Guider et contrôler les reconfigurations de systèmes à composants

Jean-François Weber

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Guider et contrôler les reconfigurations de systèmes à composants

Reconfigurations dynamiques: modélisation formelle et validation automatique

JEAN-FRANÇOIS WEBER

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

N° 2017UBFCD068

THÈSE présentée par

JEAN-FRANÇOIS WEBER

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Guider et contrôler les reconfigurations de systèmes à composants

Reconfigurations dynamiques: modélisation formelle et validation automatique

Unité de Recherche :
Département d'Informatique des Systèmes Complexes (DISC)

Soutenue publiquement le 5 octobre 2017 devant le Jury composé de :

GWEN SALAÜN	Président	Professeur HDR au LIG, Université Grenoble Alpes
RÉGINE LALEAU	Rapporteur	Professeur HDR au LACL, Université Paris-Est Créteil
PHILIPPE MERLE	Rapporteur	Chargé de recherche HDR, Inria Lille Nord Europe
OLGA KOUCHNARENKO	Directeur de thèse	Professeur HDR au DISC, Université de Bourgogne Franche-Comté
FRÉDÉRIC DADEAU	Examinateur	Maître de conférence au DISC, Université de Bourgogne Franche-Comté

REMERCIEMENTS

Avant tout, je tiens à remercier tous les membres du jury qui ont accepté d'évaluer mon travail. Merci à Gwen Salaün d'avoir présidé ce jury de thèse. Merci également à Régine Laleau et Philippe Merle pour m'avoir fait l'honneur de rapporter cette thèse.

Je voudrais remercier tout spécialement Olga Kouchnarenko de m'avoir fait confiance pour mener à bien ce travail de thèse malgré le fait que j'ai une activité professionnelle à plein temps. Olga Kouchnarenko a été disponible au delà de mes attentes les plus optimistes ; en semaine et le week-end, le jour et la nuit, en cours d'année universitaire et durant les vacances.

Je souhaite aussi remercier Jacques Julliand qui m'a suggéré de m'orienter vers un doctorat au moment où je pensais avoir déjà manqué cette opportunité, Pierre-Cyrille Héam et Frédéric Dadeau pour leur soutien, ainsi que Jean-Michel Hufflen pour l'aide spontanée qu'il m'a proposée à plusieurs reprises, sans oublier Julien Bourgeois, Benoît Piranda, Julien Dormoy, Abderrahim Ait Wakrime et Hadrien Bride.

Un (autre) grand merci à Philippe Merle sans qui l'intégration de FraSCAti dans notre implémentation n'aurait jamais pu avoir lieu (en tout cas, en si peu de temps) ; il a su trouver la disponibilité pour m'assister dans cette tâche malgré son agenda chargé.

Merci aussi à Brigitte Bataillard et Dominique Ménétrier qui ont toujours su me guider et m'assister dans le monde obscur des formalités administratives, à Oscar Carrillo et Hassan Mountasir pour l'organisation des TP Fractal et à Laurent Steck pour son support technique.

Je remercie les étudiants dont j'ai suivi le travail avec Olga Kouchnarenko : Grégori Tirsatine pour son travail sur les parseurs d'expressions de logique temporelle, Alexis Picard et Nicolas Lemasson pour leur travail sur l'interface graphique de notre implémentation, Clément Hosotte et Florian Facchini pour leur projet de Master 2 concernant les reconfigurations dynamiques, Sonia Crétin et Mathieu Robert pour le brouillage des traces d'adaptation et Martin Bérenger pour le projet de grammaires de graphes pour systèmes reconfigurables.

Enfin, je remercie Philippe Lutz de l'École Doctorale SPIM ainsi qu'Alexandrine Vieillard, Sébastien Pasteur et Samuel Gaston Amet pour leur bienveillance et leur aide.

SOMMAIRE

INTRODUCTION

Dans le monde actuel, les systèmes embarqués traditionnels sont de plus en plus remplacés par des systèmes cyber-physiques, ou CPS (*Cyber-Physical Systems*). Intuitivement, de tels systèmes peuvent être vus comme des réseaux de systèmes embarqués où un grand nombre de composants logiciels sont déployés au sein d'environnements physiques [Du et al., 2016, Lee, 2008]. Comme leurs environnements sont susceptibles de changer, ces systèmes doivent pouvoir évoluer au cours du temps tout en continuant à fonctionner correctement. Bien sûr, ces évolutions doivent permettre aux systèmes de s'améliorer (d'un point de vue quantitatif et/ou qualitatif) ou, tout du moins, de minimiser d'éventuelles dégradations. Il est donc nécessaire de pouvoir vérifier ces systèmes durant leurs exécutions afin de guider au mieux leurs évolutions pour qu'ils puissent rester adaptés à leurs environnements.

Les systèmes à composants sont constitués d'un ensemble de "briques de base" (i.e., des composants) pouvant être assemblées pour construire des applications. Ces systèmes peuvent être modifiés (e.g., ajout ou suppression de composants) au moyen de reconfigurations, dites dynamiques, sans devoir être arrêtés ou cesser de fonctionner. Lorsque de telles reconfigurations sont effectuées dans le but de permettre au système concerné de s'adapter à son environnement, on parle d'adaptation.

Dans le cadre de l'adaptation des systèmes à composants, les reconfigurations dynamiques ne peuvent pas être effectuées à n'importe quel moment ou de n'importe quelle façon. La fiabilité de ces reconfigurations doit permettre de garantir qu'elles soient effectuées comme prévu ou bien prévoir un mécanisme de retour en arrière (*rollback*) en cas de problème. De plus, certaines propriétés de sûreté doivent être maintenues malgré l'application de reconfigurations dynamiques ; par exemple, on n'imagine pas que le système de navigation d'un avion puisse cesser de fonctionner suite à l'application d'une reconfiguration dynamique. La sécurité doit aussi être prise en compte dans le cadre des reconfigurations dynamiques afin qu'elles ne puissent être effectuées que par les entités autorisées à les appliquer. En outre, dans le contexte actuel de consommation d'énergie responsable, les reconfigurations dynamiques peuvent permettre d'économiser de l'énergie en retirant des composants non nécessaires au fonctionnement du système. Ces composants pourraient bien sûr être rajoutés dynamiquement s'ils s'avéraient à nouveau nécessaires. Enfin, ces reconfigurations doivent aussi être appliquées dans le respect des usagers ; si des passagers sont présents dans un véhicule autonome, la température intérieure devrait être maintenue à un niveau confortable même si cela consomme de l'énergie.

L'adaptation des systèmes à composants via l'application de reconfigurations dynamiques se doit donc de respecter des propriétés de sûreté, d'être fiable et sécurisée tout en respectant l'environnent et les usagers. C'est pourquoi nous nous intéressons au guidage et au contrôle des reconfigurations des systèmes à composants dans leur évolution au sein de leur environnement. Nous devons aussi déterminer comment vérifier les propriétés de tels systèmes lors de leurs reconfigurations et trouver les moyens de

tester les mécanismes de l'adaptation et les reconfigurations dynamiques. La vérification à l'exécution est une technique utilisée pour s'assurer de la correction du comportement d'un système au cours de son exécution. Il est aussi envisageable d'utiliser cette technique pour valider la satisfaction de certaines propriétés du système. Ainsi, en fonction de l'évaluation de ces propriétés, il est possible de déterminer comment guider l'évolution d'un système pour contrôler son adaptation vis-à-vis de son environnement. Ce concept de l'adaptation est au cœur de nos travaux sur les systèmes à composants. Nous allons d'abord établir le contexte scientifique, les concepts et les idées qui vont nous permettre, dans un second temps, d'exposer nos problématiques et contributions. Ensuite, nous détaillerons l'organisation de ce mémoire en présentant le contenu de chaque chapitre. Enfin, nous indiquerons où nos diverses contributions ont été publiées.

CONTEXTE SCIENTIFIQUE

Dans cette section, nous introduisons les concepts et idées qui nous permettrons de pouvoir exposer les problématiques liées au travail présenté dans ce document. Plusieurs concepts se sont succédés, au cours des dernières décennies, pour la conception et le développement d'applications. Les langages machines, très performants mais ne permettant pas de séparer le développement en diverses unités ré-utilisables, ont été délaissés au profit des langages procéduraux, puis des langages objets, qui autorisent une plus grande modularité et une réutilisation du code. Comme il est difficile de mettre en évidence les liens entre les différents objets au sein d'une application lorsque leur nombre devient important, les architectures logicielles à base de composants sont de plus en plus préférées aux approches purement objet.

Les systèmes à composants sont des systèmes modulaires composés de divers composants pouvant êtres assemblés pour former des applications. On peut faire évoluer de tels systèmes via des opérations de reconfiguration permettant, par exemple, de supprimer ou d'ajouter un composant. Lorsque ces opérations de reconfiguration peuvent être effectuées sans avoir à arrêter le système, on parle de reconfigurations dynamiques. Ainsi, il est possible de modifier les systèmes à composants en cours d'exécution afin de leur permettre de s'adapter à leur environnement.

De telles adaptations sont déterminées par des règles, regroupées en politiques, ayant la possibilité de vérifier des propriétés du système durant sont exécution afin de pouvoir juger de la pertinence des actions à effectuer. Ces politiques d'adaptation peuvent s'avérer complexes à mettre en place, c'est pourquoi il est vital de pouvoir les tester. Nous commençons par nous intéresser aux idées de composants logiciels et de systèmes à composants, puis nous explicitons le concept de vérification pour ces systèmes. Enfin, nous présentons les notions de politique d'adaptation et de test dans le cadre des systèmes à composants.

COMPOSANTS LOGICIELS ET SYSTÈMES À COMPOSANTS

Un composant est généralement décrit comme un élément, d'une structure plus complexe, pouvant être intégré dans différentes applications. Afin qu'ils soient utilisables, les composants doivent exposer les fonctionnalités qu'ils fournissent mais aussi les fonctionnalités dont ils ont besoin pour s'exécuter.

Les concepts de modularité et de ré-utilisabilité permettent de structurer et de faciliter le développement d'applications basées sur les composants. La modularité permet d'extraire les différentes fonctionnalités nécessaires à la construction d'une application et de chacune les encapsuler au sein de briques logicielles. Celles-ci peuvent alors être assemblées pour construire l'application désirée. Le concept de ré-utilisabilité permet de pouvoir utiliser la même brique dans différentes applications.

On peut distinguer les composants primitifs qui encapsulent du code métier et les composants composites qui contiennent des sous-composants (primitifs ou composites). Ainsi, plusieurs composants peuvent être assemblés et liés au moyen de liens logiques pour former un système à composants. Un tel assemblage représente une configuration du système à composants. Intuitivement, si cette configuration correspond à un assemblage de composants capable de s'interconnecter entres-eux et de fonctionner sans erreur, on dit qu'elle est consistante.

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. Cependant, l'application d'une reconfiguration à une configuration donnée ne garantit pas que la configuration résultante sera consistante.

Considérons un ensemble de configurations et un ensemble de reconfigurations permettant d'évoluer d'une configuration à une autre. On peut ainsi, comme dans [Dormoy, 2011], définir un modèle de système à composants comme l'ensemble des séquences de configurations pouvant débuter par une configuration choisie dans un ensemble de configurations initiales pour évoluer de configuration en configuration au moyen des reconfigurations qu'il est possible d'effectuer.

Fractal [Bruneton et al., 2004, Bruneton et al., 2006] et FraSCAti [Seinturier et al., 2009, Seinturier et al., 2012] sont des modèles à composants supportant l'utilisation de reconfigurations dynamiques, qui permettent de reconfigurer un système à composants durant son exécution.

Dans le cadre du développement classique utilisant la programmation objet, les choix concernant l'architecture d'une application sont fait au moment de la conception. Le fait de pouvoir utiliser des reconfigurations dynamiques pour pouvoir modifier (sans avoir à le stopper) un système durant son exécution permet de différer de tels choix au moment de l'exécution. Ainsi, il est possible de faire aussi évoluer le système pour qu'il reste adapté à son environnement lorsque celui-ci évolue.

De telles possibilités posent de nouveaux problèmes auxquels nous nous efforcerons de répondre dans ce mémoire de thèse :

— Quelle(s) reconfiguration(s) effectuer et à quel moment ?

— Comment déterminer lorsque le système n'est plus (suffisamment) adapté à son environnement ?

— Comment s'assurer que l'application de reconfiguration(s) ne va pas nuire au bon fonctionnement du système ?

— Que faire si le système cesse de fonctionner comme prévu ?

VÉRIFICATION DES SYSTÈMES À COMPOSANTS

Le but de la vérification d'un système est de s'assurer que sa spécification vérifie les propriétés qu'il doit respecter. Il est donc possible de procéder à la vérification d'un système avant qu'il ne soit implémenté car la vérification est effectuée en utilisant seulement une spécification. Deux approches formelles de vérification peuvent être distinguées : la vérification algorithmique (*model-checking*) et la vérification par preuve (*theorem proving*). Le *model-checking* consiste à explorer de manière exhaustive l'ensemble des états du système, afin de s'assurer que celui-ci satisfait une certaine propriété. Le principe de la vérification par preuve consiste à prouver mathématiquement la validité d'un ensemble de formules (nommées obligations de preuve) engendrées par une spécification formelle décrivant le système et ses propriétés. Ces deux approches peuvent être combinées comme c'est le cas dans [Lanoix et al., 2011].

Dans le cadre du *model-checking* appliqué au modèle à composants Fractal [Bruneton et al., 2006], les travaux de [Merle et al., 2008, Tiberghien et al., 2010] ont permis de représenter ce modèle de façon à pouvoir utiliser Alloy [Jackson, 2012], un logiciel de *model-checking* (ou *model-checker*). Le *model-checker* CADP [1] [Garavel et al., 2007, Garavel et al., 2011, Garavel et al., 2013] a pu aussi être utilisé pour vérifier des systèmes à composants basés sur Fractal dans [Barros et al., 2007]. Nous pouvons aussi citer les travaux [Simonot et al., 2008] qui utilisent Focal [Dubois et al., 2006] pour vérifier de tels systèmes à composants.

Ces approches utilisent les fonctionnalités d'outils existants mais elles ne sont pas toutes bien adaptées aux concepts de composants logiciels, c'est pourquoi d'autres approches donnent la possibilité d'enrichir les spécifications des composants avec des éléments et des concepts qui seront ensuite vérifiés. Nous pouvons citer ConFract [Collet et al., 2005, Collet et al., 2007] qui est une extension du modèle à composants Fractal vérifiant des Pré et Post-conditions [Hoare, 1969]. Le paradigme d'hypothèses/garanties [Misra et al., 1981, Lamport, 1983] est utilisé pour effectuer la vérification compositionnelle des systèmes à composants dans [Andrade et al., 2002]. Enfin, l'approche proposée dans [Chouali et al., 2010] utilise des automates d'interfaces pour spécifier des contrats au niveau comportemental.

Dans le cadre des systèmes à composants supportant les reconfigurations dynamiques, les travaux proposés dans [David et al., 2009] définissent une extension du modèle à composants Fractal dans laquelle il est possible de spécifier des invariants sur la structure du modèle à composants. Cette approche vérifie que ces invariants architecturaux sont préservés lors de l'exécution des reconfigurations dynamiques.

Afin d'exprimer des contraintes architecturales et temporelles sur des systèmes à composants, la logique FTPL a été introduite dans [Dormoy et al., 2012a]. Cette logique qui se base sur les travaux de Dwyers sur les schémas de spécification [Dwyer et al., 1999], permet d'évaluer des propriétés temporelles dans $\mathbb{B}_2$ (en utilisant les valeurs 'faux" et "vrai") sur des traces d'exécution. On notera que la structure composite des systèmes à composants se prête à une approche décentralisée ; l'évaluation décentralisée de formules LTL présentée dans [Bauer et al., 2012] en est un exemple.

La vérification à l'exécution [Havelund et al., 2005, Bauer et al., 2010] est une technique permettant de s'assurer de manière efficace de la satisfaction d'exigences par un système lors de son exécution. Ceci nécessite l'évaluation de propriétés à

1. http://www.inrialpes.fr/vasy/cadp/

l'exécution, ce qui implique de considérer des traces d'exécution incomplètes. C'est le cas dans [Dormoy et al., 2011] où des propriétés temporelles sont évaluées dans $\mathbb{B}_4$, en utilisant les valeurs 'faux" et "vrai", d'une part et "potentiellement faux" et "potentiellement vrai", d'autre part, pour la potentielle (non) satisfaction d'une propriété.

POLITIQUES D'ADAPTATION ET TEST DE SYSTÈMES À COMPOSANTS

Dans le cadre des systèmes à composants, les politiques d'adaptation peuvent être vues comme un ensemble de règles pouvant guider et contrôler les reconfigurations dynamiques. Pour un système à composants donné, ces règles contiennent des propriétés concernant le système et son environnement. En se basant, au cours de l'exécution du système, sur l'évaluation de ces propriétés on peut alors déterminer comment le système devrait évoluer. De telles règles peuvent s'avérer difficiles à élaborer dans la cas de systèmes à composants de grandes tailles. C'est pourquoi il est nécessaire de pouvoir les tester.

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. Des informations sur l'environnement du système peuvent être obtenues en utilisant des capteurs externes (du point de vue du système) de façon à détecter des changements. Ces changements de l'environnement correspondent à des événements externes pouvant être pris en compte par les propriétés utilisées par les règles incluses dans les politiques d'adaptation. Un modèle de système à composants pouvant évoluer sur la base de politiques d'adaptation utilisant des schémas temporels (basés sur la logique qMEDL [Gonnord et al., 2009]) prenant en compte des événements externes a été introduit dans [Dormoy et al., 2010].

L'enforcement d'une propriété à l'exécution [Schneider, 2000, Falcone et al., 2008, Ligatti et al., 2009, Delaval et al., 2010] consiste à utiliser la vérification des propriétés à l'exécution pour essayer d'anticiper l'apparition d'une exécution non désirée qui pourrait rendre le système défaillant. Cette approche est en fait une extension de l'approche de vérification à l'exécution qui permet de répondre au besoin de prévenir et de corriger un comportement non-désiré. La réflexion à l'exécution [Bauer et al., 2008] est une approche utilisant aussi la vérification de propriétés lors de l'exécution du système. Cette méthode a non seulement pour but d'observer le comportement du système mais aussi de donner un diagnostic de l'exécution. Lorsque le diagnostic est assez détaillé et non-ambigü, celui-ci peut être utilisé pour faire réagir le système en exécutant des commandes, telles que des reconfigurations dynamiques par exemple, pour rétablir le système dans un état bien défini. Ainsi, en complément des propriétés utilisées par les règles composant les politiques d'adaptation, on peut définir d'autres propriétés que l'on peut *a*) préserver par le mécanisme d'enforcement, ou *b*) utiliser, via la réflexion, pour déclencher des actions correctrices lorsque ces propriétés ne sont pas satisfaites.

Lorsqu'on cherche à s'assurer que l'implémentation d'un système vérifie les propriétés qu'il doit respecter en analysant un ensemble d'exécutions passées, on parle de test plutôt que de vérification. Les premiers travaux théoriques sur le test logiciel datent des années 1970 avec l'apparition des notions de tests "fiables" [Howden, 1976] (*reliable*) et "idéaux" [Goodenough et al., 1975] (*ideal*) qui permettent d'établir que le test ne peut que montrer la présence d'erreur, mais jamais leur absence [Dijkstra, 1970].

Le concept d'hypothèse de test, qui définit les pré-suppositions faites sur le système à tester, a été formalisé dans [Bernot et al., 1991]. Le test de système 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 [Weyuker, 1998]. En effet, bien que chaque composant ait pu être testé indépendamment avec succès, leur combinaison peut engendrer des comportements imprévus.

D'après [Bertolino, 2007], il subsiste deux défis à relever dans le cadre du test de système à composants. Le premier est un défi technique qui consiste à trouver les moyens de pouvoir tester les composants au sein des différents contextes dans lesquels ils peuvent être déployés. Le second défi, plus théorique, est de pouvoir obtenir davantage d'informations sur les composants à tester que ce qui est proposé par les différents modèles de systèmes à composants. Ces informations additionnelles pourraient être utilisées pour la génération de test. Il s'agirait, par exemple, de tests intégrés dans les composants eux-mêmes (*built-in testing*) ou de données permettant l'élaboration de tests pertinents.

PROBLÉMATIQUES ET CONTRIBUTIONS

Dans cette section, nous présentons les problématiques auxquelles nous souhaitons répondre. Ensuite, nous détaillons nos contributions en indiquant les problématiques adressées.

PROBLÉMATIQUES

Le but du travail présenté dans ce document est de pouvoir guider et contrôler les systèmes à composants dans leur évolution.

Le modèle de système à composants présenté dans [Léger, 2009, Léger et al., 2010, Dormoy, 2011] peut évoluer d'une configuration à une autre au moyen d'opérations de reconfiguration pouvant être des séquences de reconfigurations primitives. De plus lors de l'évolution vers une configuration donnée, celle-ci peut ne pas être dans un état consistant. C'est pourquoi nous proposons la problématique suivante :

(1) "*Définir un modèle permettant de garantir la consistance des configurations "cibles" sans se limiter à des reconfigurations définies comme des séquences de reconfigurations primitives.*"

La logique FTPL introduite dans [Dormoy et al., 2012a] permet d'exprimer des contraintes architecturales et temporelles des systèmes à composants. Cette logique repose sur des schémas de spécifications [Dwyer et al., 1999] et permet d'exprimer des propriétés en prenant en compte le déclenchement des reconfigurations dynamiques générées par le système. Ces propriétés peuvent alors porter sur la passé ou sur le futur de l'exécution. La logique proposée permet d'exprimer des exigences sur l'enchainement des reconfigurations dynamiques d'un système à composants qui doivent être vérifiées pendant l'exécution. Dans [Dormoy et al., 2011], des propriétés temporelles sont évaluées à l'exécution de façon centralisées. On notera que, souvent, l'évaluation

de propriétés temporelles à l'exécution nécessite de maintenir un long historique des informations concernant les évaluation passées car elles peuvent être nécessaires pour les évaluations futures. De plus, dans [Dormoy et al., 2010], des politiques d'adaptation utilisant des schémas temporels (basés sur la logique qMEDL) prennent en compte des événements externes. Malheureusement, qMEDL ne permet pas d'exprimer de contraintes architecturales sur des systèmes à composants. Il serait souhaitable d'avoir une seule et même logique pour définir des politiques d'adaptation et exprimer des contraintes architecturales, c'est la raison pour laquelle nous proposons cette seconde problématique :

(2) "*Choisir ou établir une logique temporelle prenant en compte des événements internes et externes afin d'exprimer des politiques d'adaptation et des contraintes architecturales ; évaluer, de façon centralisée ou décentralisée, les propriétés basées sur cette logique à l'exécution sur des traces incomplètes, sans avoir à maintenir un long historique ; et intégrer ces propriétés dans des politiques d'adaptation*."

La problématique suivante concerne la mise en place d'une implémentation du modèle mentionné ci-dessus :

(3) "*Développer une application permettant de contrôler et guider les reconfigurations au moyen de politiques d'adaptation pour un système à composants basé sur Fractal ou sur un autre modèle à composants ; et utiliser cette application sur différents cas d'étude.*"

Enfin, cette dernière problématique concerne le test de notre modèle :

(4) "*Avoir la possibilité de tester le modèle à composants reconfigurable en testant des politiques d'adaptation ; et en simulant le modèle de reconfiguration, ainsi que l'évaluation décentralisée de propriétés temporelles.*"

Nous présentons ci-dessous ces problématiques de façon plus structurée. La numérotation écrite en caractères gras sera utilisée par la suite pour désigner des éléments de ces problématiques.

(1) Définir un modèle permettant
  - **(1.1)** de garantir la consistance des configurations "cibles" ; et
  - **(1.2)** de ne pas se limiter à des reconfigurations définies comme des séquences de reconfigurations primitives.

(2) Choisir ou établir une logique temporelle
  - **(2.1)** prenant en compte des événements internes et externes afin d'exprimer des politiques d'adaptation et des contraintes architecturales ;
  - (2.2) telle que l'on puisse évaluer les propriétés basées sur cette logique
    - **(2.2.1)** à l'exécution sur des traces incomplètes,
    - **(2.2.2)** sans avoir à maintenir un long historique, et
    - **(2.2.3)** de façon décentralisée ; et

(2.3) intégrer ces propriétés dans des politiques d'adaptation.

(3) Développer une application permettant

(3.1) de contrôler et guider les reconfigurations au moyen de politiques d'adaptation pour un système à composants basé sur Fractal ou sur un autre modèle à composants ; et

(3.2) utiliser cette application sur différents cas d'étude.

(4) Avoir la possibilité de tester notre modèle en

(4.1) testant des politiques d'adaptation ; et en

(4.2) simulant

(4.2.1) notre modèle de reconfiguration, ainsi que

(4.2.2) l'évaluation décentralisée de propriétés temporelles.

CONTRIBUTIONS

Nous présentons nos contributions en indiquant, au moyen de la numérotation précédemment établie, à quels éléments des problématiques ci-dessus elles correspondent.

Nous définissons un modèle abstrait en utilisant la logique du premier ordre. Ceci permet d'introduire la notion de reconfiguration gardée afin de garantir la consistance des configurations d'un système à composants sous la condition que les configurations initiales de notre modèle soient consistantes. Cette propagation de la consistance est étendue à un modèle interprété **(1.1)**.

Ces reconfigurations gardées utilisent les opérations de reconfiguration primitives en tant que "briques de base" pour construire des reconfigurations non-primitives impliquant des constructions, non seulement, séquentielles, mais aussi alternatives ou répétitives **(1.2)**.

Nous étendons la logique FTPL introduite dans [Dormoy et al., 2012a] avec des événements externes. Ceci permet de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité **(2.1)**.

Nous définissons la sémantique FTPL (étendue aux événements externes) en utilisant les quatre valeurs de $\mathbb{B}_4$ pour l'évaluation de propriétés FTPL à l'exécution sur des traces d'exécution incomplètes **(2.2.1)**. De plus, afin de pouvoir évaluer ces propriétés à l'exécution sans avoir à maintenir un historique des évaluations, nous introduisons une méthode d'évaluation progressive permettant dans la plupart des cas d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente **(2.2.2)**.

Nous proposons une méthode pour l'évaluation décentralisée de propriétés FTPL en utilisant les notions de progression et d'urgence. Une fonction de progression permet de réécrire une formule représentant l'évaluation d'une propriété à une configuration donnée de façon qu'elle soit toujours pertinente à la configuration suivante. Dans ce contexte, nous décomposons les formules en sous-formules auxquelles nous attribuons un niveau d'urgence pour déterminer l'ordre dans lequel les évaluer. Nous posons aussi formellement le problème de l'évaluation décentralisée et proposons un algorithme pour le résoudre ainsi que des résultats de correction et de terminaison **(2.2.3)**.

Nous définissons des politiques d'adaptation intégrant la logique FTPL **(2.3)**. Nous proposons aussi une relation de simulation sur les modèles de systèmes à composants reconfigurables et nous définissons le problème de l'adaptation. En utilisant cette relation, nous établissons que le problème de l'adaptation est semi-décidable et présentons un algorithme pour l'enforcement des politiques d'adaptation de systèmes à composants reconfigurables. De plus, nous décrivons l'utilisation du fuzzing [Takanen et al., 2008], une technique de test que nous appliquons au cas spécifique du test de politiques d'adaptation **(4.1)**.

Nous décrivons notre implémentation permettant de guider et contrôler les reconfigurations de systèmes à composants via des politiques d'adaptation basées sur la logique temporelle FTPL. Cette implémentation supporte les modèles à composants Fractal et FraSCAti **(3.1)**. Nous présentons les différentes entités qui composent notre implémentation et détaillons leur fonctionnement ainsi qu'un résultat de conformité architecturale en nous basant sur le modèle interprété précédemment défini. Ensuite, nous montrons une partie des informations contenues dans les logs générés par notre implémentation et la façon dont elles sont utilisées pour produire les sorties pour notre interface graphique et pour la génération de tests basés sur le fuzzing. Enfin, nous décrivons l'intégration de la fonctionnalité de fuzzing permettant de tester des politiques d'adaptation **(4.1)**.

Nous appliquons les politiques d'adaptation que nous avons décrites précédemment à un cas d'étude en utilisant notre implémentation qui permet de contrôler et guider les reconfigurations de systèmes à composants en utilisant les mécanismes d'enforcement et de réflexion pour obtenir les résultats attendus **(3.2)**. Deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti sont aussi présentés **(3.2)**. De plus notre modèle a été implémenté avec l'outil de transformation de graphes GROOVE [Ghamarian et al., 2012]. Une expérimentation de cette implémentation est présentée **(4.2.1)**. En outre, nous décrivons la façon dont nous simulons l'évaluation décentralisée sous GROOVE **(4.2.2)**. Nous montrons comment nous représentons les propriétés FTPL sous GROOVE et présentons l'implémentation (toujours sous GROOVE) de l'algorithme permettant l'évaluation décentralisée de propriétés FTPL **(4.2.2)**.

ORGANISATION

La figure 1 représente la structure du présent document (sans tenir compte des chapitres 1 et 2 qui proposent respectivement une introduction aux systèmes à composants et un recueil des définitions et notations utilisées). Sur cette figure, les flèches pleines indiquent l'ordre de lecture des chapitres de ce mémoire, tandis que les flèches discontinues montrent les chemins possibles pour une lecture cohérente. Il est ainsi possible de lire dans l'ordre les chapitres 3, 5, 6, 8 et 10 et de revenir plus tard à la lecture des chapitres 4, 7 et 9. La suite du document est organisée en quatre parties de la façon suivante :

— La partie I présente le contexte scientifique et définit les notions et notations utilisées dans ce document,

— la partie II décrit notre modèle de reconfiguration, définit les reconfigurations gardées et étend la logique FTPL pour supporter les événements externes,

— la partie III présente l'évaluation centralisée et décentralisée des propriétés FTPL à l'exécution ainsi que leur intégration dans des politiques d'adaptation,

— La partie IV décrit notre implémentation ainsi que plusieurs cas d'étude et simulations, et

— la conclusion nous permet de dresser un bilan et d'envisager les perspectives des travaux présentés dans ce mémoire.

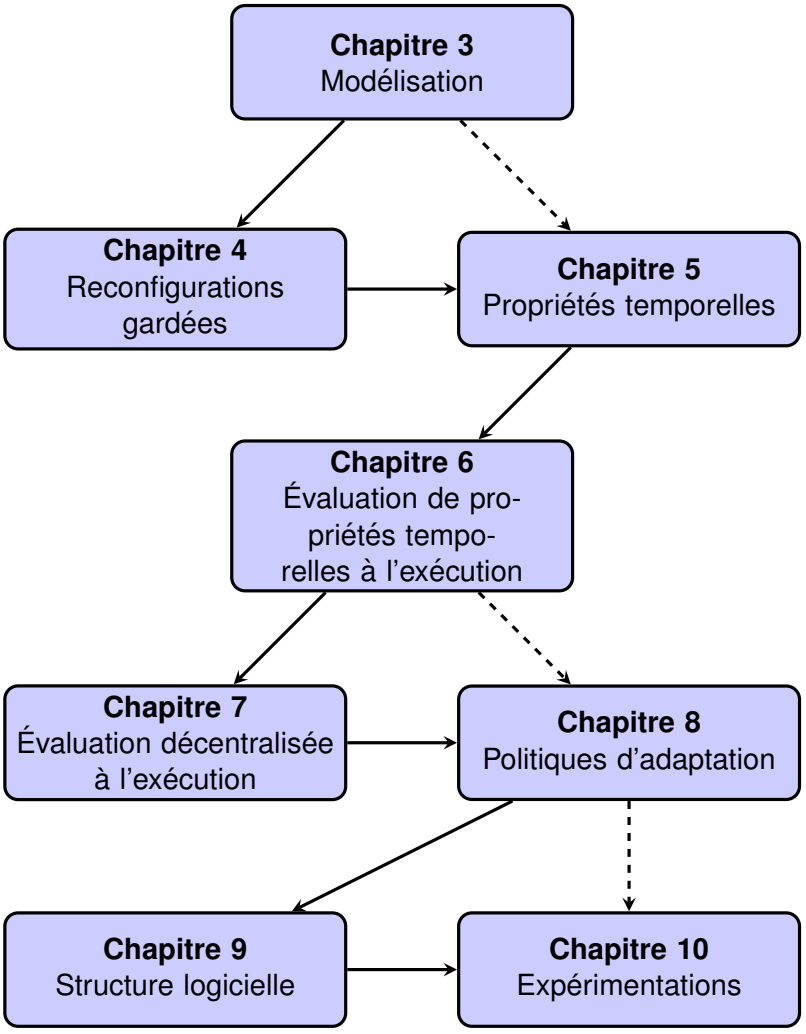


FIGURE 1 – Structure du mémoire

PARTIE I : CONTEXTE SCIENTIFIQUE ET PRÉLIMINAIRES

Dans le chapitre 1 nous introduisons le concept de composant en explicitant la notion d'architectures orientées composants, puis nous exposons l'idée de reconfiguration dans le cadre des systèmes à composants. Pour finir, nous présentons les deux implémentations de systèmes à composants que nous utilisons pour illustrer les contributions de cette thèse ; c'est-à-dire les modèles à composants Fractal [Bruneton et al., 2004, Bruneton et al., 2006] et FraSCAti [Seinturier et al., 2009, Seinturier et al., 2012].

Le chapitre 2 est consacré aux définitions et notations utilisées dans la suite du document. Nous nous intéressons à la spécification et la vérification des systèmes ainsi qu'à l'expression de leurs propriétés.

PARTIE II : MODÉLISATION DES RECONFIGURATIONS DYNAMIQUES

Au chapitre 3, nous proposons, comme dans [Dormoy, 2011] de décrire la sémantique d'un système à composants en utilisant un système de transition. Pour ce faire, nous présentons dans un premier temps, pour servir de base aux exemples utilisés, le composant composite **Location** utilisé pour le positionnement du Cycab, un véhicule autonome. Ensuite, nous définissons une sémantique opérationnelle permettant de décrire l'architecture des systèmes à composants ainsi que les reconfigurations primitives qui sont des opérations d'évolution basiques permettant de passer d'une configuration architecturale à une autre pour définir les systèmes à composants reconfigurables primitifs. Enfin, nous abordons la notion de reconfiguration non primitive pour définir les systèmes à composants reconfigurables (non primitifs).

Nous proposons dans le chapitre 4 de décrire des contraintes sur les éléments et relations des configurations de systèmes à composants reconfigurables afin de pouvoir garantir que le système à composants considéré soit dans un état permettant son bon fonctionnement. En outre, en nous basant sur les pré/post-conditions des opérations primitives de reconfiguration, nous introduisons les reconfigurations gardées qui permettent d'utiliser les opérations primitives en tant que "briques de base" pour construire des reconfigurations non-primitives impliquant des constructions, non seulement, séquentielles, mais aussi alternatives ou répétitives. Ensuite, nous montrons que l'utilisation de reconfigurations gardées garantit la consistance des configurations d'un système à composants sous la condition que les configurations initiales soient consistantes. Enfin, nous définissons un modèle interprété en partant du modèle abstrait présenté au chapitre 3.

Le chapitre 5 présente la logique FTPL introduite dans [Dormoy et al., 2012a] qui est utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permet pas la prise en compte d'événements externes. C'est pourquoi nous étendons cette logique avec des événements externes afin de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité. Nous présentons la syntaxe de la logique FTPL étendue aux événements externes, puis nous décrivons sa sémantique et la notion de satisfaction d'une propriété FTPL par un système à composants.

PARTIE III : VÉRIFICATION ET TEST DES RECONFIGURATIONS DYNAMIQUES

Nous proposons au chapitre 6, après avoir introduit le concept général d'évaluation à l'exécution, une logique à quatre valeurs ("faux", "potentiellement faux", "potentiellement vrai" et "vrai") pour évaluer les propriétés FTPL à l'exécution sur des traces d'exécution incomplètes. En outre, pour pouvoir évaluer les propriétés FTPL à l'exécution sans avoir à maintenir un historique des évaluations, nous introduisons une méthode d'évaluation progressive des propriétés FTPL permettant dans la plupart des cas d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente.

En s'inspirant de l'évaluation décentralisée de formules LTL présentée dans [Bauer et al., 2012], nous proposons au chapitre 7 une méthode pour l'évaluation décentralisée de propriétés FTPL. Nous définissons dans un premier temps les conventions utilisées pour l'évaluation décentralisée. Ensuite, nous présentons les notions de progression et d'urgence. Une fonction de progression nous permet de réécrire une formule représentant l'évaluation d'une propriété FTPL à une configuration donnée de façon qu'elle soit toujours pertinente à la configuration suivante. Dans le contexte de l'évaluation décentralisée, nous décomposons les formules en sous-formules auxquelles nous attribuons un niveau d'urgence pour déterminer l'ordre dans lequel les évaluer. Enfin, nous posons formellement le problème de l'évaluation décentralisée et proposons un algorithme pour le résoudre ainsi que des résultats de correction et de terminaison.

Le chapitre 8 montre comment nous intégrons la logique FTPL étendue aux événements externes définie au chapitre 5 dans des politiques d'adaptation que nous définissons. Nous proposons aussi une relation de simulation sur les modèles de systèmes à composants reconfigurables et définissons le problème de l'adaptation. En utilisant cette relation, nous établissons que le problème de l'adaptation est semi-décidable. De plus, nous fournissons un algorithme pour l'implémentation de politiques d'adaptation pour l'enforcement de propriétés FTPL sur des systèmes à composants reconfigurables. Enfin, nous présentons l'usage du fuzzing pour le test de politiques d'adaptation.

PARTIE IV : VALIDATION EXPÉRIMENTALE

Le chapitre 9 présente l'implémentation que nous avons développée pour les reconfigurations dynamiques de systèmes à composants guidées par des politiques d'adaptation basées sur des schémas temporels. Dans un premier temps, nous décrivons notre implémentation, avec les entités qui la composent et leur fonctionnement, puis nous énonçons un résultat de conformité architecturale. Ensuite, nous détaillons une partie des informations contenues dans les logs générés par notre implémentation et comment nous les utilisons pour en extraire les éléments permettant de produire une interface graphique. Enfin nous présentons l'intégration de la fonctionnalité de fuzzing permettant de tester des politiques d'adaptation.

Au chapitre 10, nous montrons comment l'exécution de notre implémentation permet de contrôler et guider les reconfigurations du composant **Location**, présenté au chapitre 3, en utilisant les mécanismes d'enforcement et de réflexion. Deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti sont présentés. Le premier modélise des machines virtuelles au sein d'un environnement *cloud* et le second s'inspire du serveur HTTP *Comanche Http Server* utilisé dans [Chauvel et al., 2009]. De plus notre modèle a été implémenté avec l'outil de transformation de graphe GROOVE [Ghamarian et al., 2012]. Une expérimentation de cette implémentation utilisant comme cas d'étude une machine virtuelle de l'environnement *cloud* est présentée. Enfin, nous décrivons la façon dont nous simulons l'évaluation décentralisée sous GROOVE. Nous montrons comment nous représentons les propriétés FTPL sous GROOVE et présentons l'implémentation sous GROOVE de l'algorithme établi pour l'évaluation décentralisée de propriétés FTPL à l'exécution sur le modèle du système à composants **Location**.

CONCLUSION ET PERSPECTIVES

Dans cette partie, nous dressons le bilan et présentons les perspectives du travail réalisé au cours de cette thèse.

PUBLICATIONS

Les travaux présentés dans cette thèse ont été soutenus en partie par le projet Labex ACTION, ANR-11-LABEX-0001-01. Par ailleurs, les différentes contributions de cette thèse, listées ci-dessous, ont été publiées dans une revue nationale et différentes conférences internationales :

— Les contributions des chapitres 5 et 6 et une partie de celles des chapitres 8, 9 et 10 ont été publiées dans [Kouchnarenko et al., 2014a].

— Les contributions du chapitre 7 et une partie de celles des chapitres 9 et 10 ont été publiées dans [Kouchnarenko et al., 2014b].

— Les contributions du chapitre 4 et une partie de celles des chapitres 9 et 10 ont été publiées dans [Kouchnarenko et al., 2015].

— Les contributions du chapitres 3 et une partie des contributions des chapitres 9 et 10 ont été publiées dans [Kouchnarenko et al., 2016].

— Une partie des contributions des chapitres 8 et 9 a été publiée dans [Weber, 2016].

I

CONTEXTE SCIENTIFIQUE ET PRÉLIMINAIRES

1

LES SYSTÈMES À COMPOSANTS

Bien que le terme de composants avait déjà été utilisé dans les années 1960 pour désigner des fragments de code, ce concept utilisé dans le cadre de l'ingénierie logicielle à base de composants est relativement récent. Intuitivement, cette technique de développement utilise un ensemble de briques logicielles (i.e., des composants) pouvant être assemblées pour construire une application. Cette méthodologie repose sur des composants réutilisables et favorise la mise en place de systèmes modulaires qui peuvent être appréhendés et développés plus simplement. Dans ce chapitre, nous introduisons le concept de composant en explicitant la notion d'architectures orientées composants, puis nous exposons l'idée de reconfiguration dans le cadre des systèmes à composants. Pour finir, nous présentons les deux implémentations de systèmes à composants que nous utilisons pour illustrer les contributions de cette thèse ; c'est-à-dire les modèles à composants Fractal et FraSCAti.

Sommaire

1.1/ ARCHITECTURES ORIENTÉES COMPOSANTS

Les architectures orientées composants sont souvent mentionnées en utilisant les acronymes CBD (*Component-Based software Development*), CBSE (*Component-Based Software Engineering*) [Szyperski, 1995, Council et al., 2001] ou SCA (Service Component Architecture). Les principaux avantages de ces approches résident dans la modularité rendue possible par l'utilisation de composants et dans la ré-utilisabilité de ceux-ci. Ces aspects permettent notamment de faciliter le développement de logiciels complexes. Dans cette section, nous détaillons le concept de composant, puis nous nous intéresserons aux architectures logicielles basées sur les composants. Enfin, pour finir, nous présentons brièvement quelques approches existantes de modèles à composants.

1.1.1/ LE CONCEPT DE COMPOSANT

L'examen de l'évolution des concepts utilisés par les technologies de l'information tend à montrer une constante volonté de simplification du développement au cours de laquelle les concepts manipulés sont de plus en plus abstraits au fur et à mesure que la complexité des systèmes augmente. La programmation pour les systèmes complexes et distribués a rendu nécessaire un type d'architecture logicielle basée sur des entités autonomes pouvant interagir avec leur environnement et entre-elles. Il est à noter que le terme d'architecture logicielle, bien qu'utilisé depuis les années 1960 [Naur et al., 1969], ne s'est imposé qu'à partir des années 1990 [Kruchten et al., 2006].

La figure 1.1 extraite de [Ait Wakrime, 2015] montre l'évolution des architectures logicielles depuis les années 1960 qui ont vu l'émergence d'une architecture basée sur la programmation structurée [Warnier et al., 1974]. Par la suite, l'architecture basée sur la décomposition fonctionnelle et celle basée sur le mode de communication client-serveur [Dahl et al., 1972, Shapiro, 1969] [1] sont apparues. L'architecture 3-tiers, qui est une extension de l'architecture client-serveur [Eckerson, 1995], a vu le jour durant la décennie 1980-1990.

Au fil des années, les recherches académiques ont fait apparaître des architectures distribuées orientées objets [Clarke et al., 1999, Nierstrasz, 1995]. La programmation objet a été une évolution majeure de la programmation procédurale en permettant un niveau d'abstraction plus élevé et davantage de modularité, notamment grâce aux notions d'héritage et de polymorphisme. Enfin, la technologie des architectures logicielles à base de composants est progressivement devenue plus puissante et a facilité le développement des systèmes [Council et al., 2001]. En effet, l'approche objet n'apparaissait pas suffisamment abstraite et souffrait du manque de mécanismes permettant la composition des objets [Meyer, 1988]. Il s'avérait difficile de mettre en évidence les liens entre les différents objets au sein d'une application lorsque leur nombre devenait important. En bref, la programmation objet était encore trop proche du code pour permettre le développement aisé d'applications tirant pleinement partie de concepts tels que la réutilisation et la substitution d'objets.

Les composants peuvent être considérés comme des boîtes noires (*black boxes*) dont l'implémentation ou la structure interne n'est pas connue a priori. Ceci favorise la réutilisation, l'interchangeabilité et la séparation des préoccupations (*separation of*

1. [Shapiro, 1969] est aussi connu en tant que RFC 4 (https://www.rfc-editor.org/rfc/rfc4.txt)

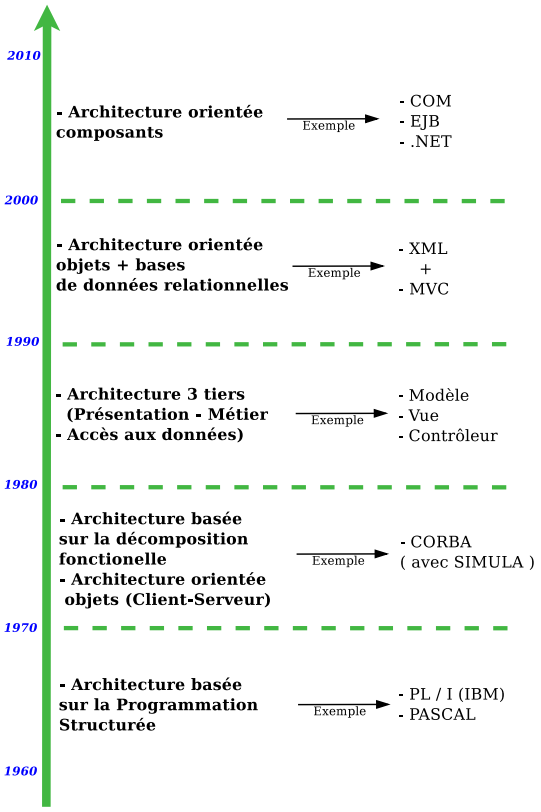


FIGURE 1.1 – Historique des architectures logicielles (extrait de [Ait Wakrime, 2015])

concerns). On notera qu'il peut aussi être possible, suivant les informations dont on dispose, de considérer les composants comme des boîtes blanches (*white boxes* — dans le cas où l'on connaît la totalité de l'implémentation et de la structure interne) ou grises (*grey boxes* — dans le cas où l'on ne dispose que d'une partie des informations).

Il n'existe pas de définition unique d'un composant et la plupart de celles qu'on trouve dans la littérature sont intuitives. La définition donnée par Microsoft [Microsoft Corporation, 1995] est très générale :

> *" . . . a piece of compiled software, which is offering a service . . . "*

D'Souza et Wills, dans [D'Souza et al., 1998], proposent une définition plus précise ; les auteurs insistent sur la notion de composition entre composants qui doit pouvoir se faire sans avoir à effectuer le moindre changement au niveau des composants eux-mêmes :

> *"A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger."*

La définition donnée par Sametinger dans [Sametinger, 1997] contient la notion de réutilisabilité et mentionne que les interfaces (des composants) doivent être clairement définies :

"Reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status."

C'est aussi le cas de la définition qui se trouve dans la spécification UML 1.3 (OMG 1999) [Rayner et al., 2005] :

"A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files."

Enfin, la définition la plus utilisée est celle de Szyperski dans [Szyperski, 1995] :

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

Nous pouvons nous inspirer de ces définitions (en particulier de la dernière) pour exprimer trois propriétés des composants :

— Les composants sont des unités de composition qui peuvent être assemblées pour former une application,
— les services fournis ainsi que les dépendances requises par un composant sont exprimés sous la forme de contrats explicites, et
— les composants sont indépendants entres-eux et peuvent donc être déployés indépendamment.

On peut donc concevoir un composant comme un élément architectural, encapsulant une partie métier d'une application, qui fournit des services et peut requérir des dépendances.

Lors de l'assemblage de plusieurs composants, des mécanismes de communication apparaissent avec la spécification de contrats entre les composants. Ces contrat peuvent s'avérer plus complexes que ceux utilisés dans le cas de la programmation objet mais apportent une modularité plus fine tout en facilitant la réutilisation des éléments d'un système. Cette modularité permet aussi de renforcer l'encapsulation des parties métier d'une application au sein de composants. Ces notions de modularité et d'encapsulation permettent de rendre un assemblage de composants adaptable à son environnement. Nous détaillerons cet aspect dans la section 1.2.

1.1.2/ ARCHITECTURES LOGICIELLES BASÉES SUR LES COMPOSANTS

Nous avons vu qu'il n'y a pas de définition unique de ce qu'est un composant ; c'est pourquoi les modèles à composants sont souvent définis en fonction d'un domaine particulier. Nous pouvons néanmoins définir les principaux concepts communs à la plupart des applications utilisant des composants. Dans cette section, nous présentons les notions de composants primitifs et composites, puis 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.

1.1.2.1/ 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 figure1.2 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 C2 et C5.

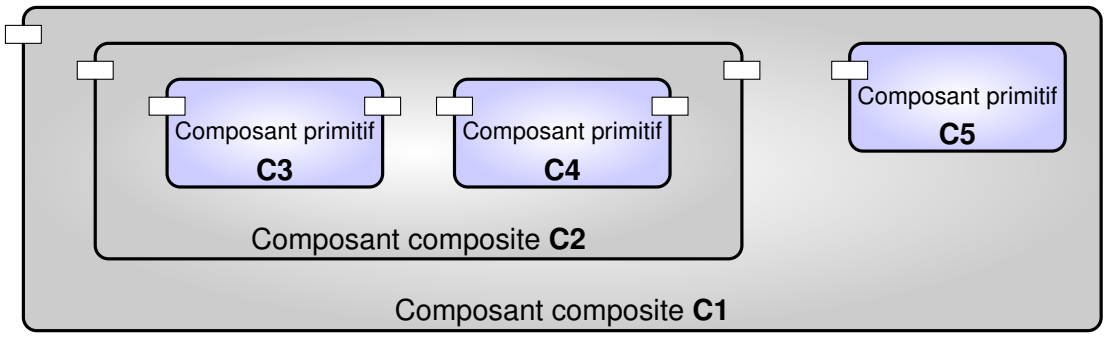


FIGURE 1.2 – Exemple de composant composite

Ainsi, les composants composites encapsulent des composants primitifs ou d'autres composants composites, ce qui permet de masquer certains détails dans une architecture logicielle. La notion de modularité en est renforcée puisque ce concept permet la réutilisation directe d'assemblages de composants. Ces composants composites peuvent alors être "mis sur une étagère" selon le principe de COTS (*Component Off The Shelf*) qui permet la mise à disposition de composants prêts à l'emploi. Cet aspect lié aux composants facilite aussi la compréhension du système en permettant de considérer le système à un haut niveau d'abstraction.

1.1.2.2/ INTERFACES

Les interfaces d'un composant sont les seuls points d'accès du programme encapsulé dans ce composant depuis ou vers l'extérieur. Les seules informations nécessaires à l'utilisation du composant sont les spécifications de ses interfaces ; cela peut prendre la forme d'une description des propriétés et dépendances fonctionnelles ou des services offerts par le composant [Bergner et al., 1998]. 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 1.3 illustre cette notion d'interface.

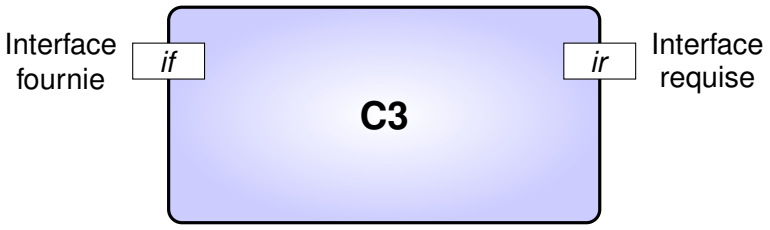


FIGURE 1.3 – Interfaces fournie et requise d'un composant

D'après [Oussalah, 2005], 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. Notons que plusieurs ports peuvent être utilisés par un même service.

Les différents 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 [Beugnard et al., 1999]. 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 [Hoare, 1969] 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 [Hoare, 1978] ; et

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

1.1.2.3/ ASSEMBLAGES DE COMPOSANTS

Afin de former 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. Bien sûr, 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 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-composants 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 1.4 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.

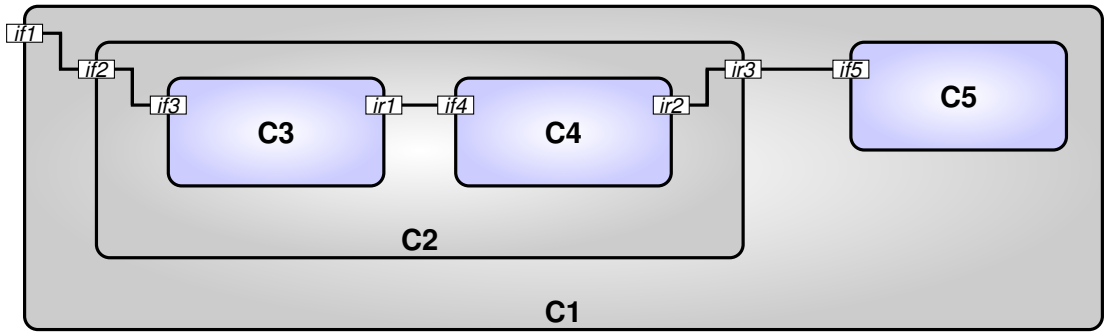


FIGURE 1.4 – Assemblage de composants

L'opération d'assemblage permet de créer une brique logicielle qui peut représenter un système complet ou peut être utilisée comme un sous-composant d'un système plus complexe. Il est bien sûr possible de réutiliser ce composant composite dans différents systèmes.

1.1.3/ QUELQUES APPROCHES DE MODÈLES À COMPOSANTS

Dans [Council et al., 2001], un composant est défini comme un élément logiciel conforme à un modèle à composants qui définit des standards de composition et d'interaction. Bien que le concept de modèle à composants permette de s'affranchir du langage de programmation, il n'existe pas d'unification des modèles à composants. C'est pourquoi l'exécution d'un modèle à composants nécessite une implémentation qui fournit un environnement dédié à son exécution.

Les deux implémentations que nous utilisons pour illustrer les contributions de cette thèse sont les modèles à composants Fractal et FraSCAti que nous présenterons respectivement dans les sections 1.3 et 1.4. Nous mentionnons brièvement ci-dessous quelques autres approches de modèles à composants.

EJB : Les EJB [DeMichiel et al., 2006] (*Entreprise Java Beans*) sont 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 : CCM [Wang et al., 2001] (*CORBA Component Model*) est un modèle à composants standardisé par l'OMG (*Object Management Group*). CCM supporte des 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 réceptacles) ainsi que des sources et des puits d'événements pour le support de services asynchrones.

Les services Web : L'architecture SOA [Papazoglou et al., 2007] (*Service Oriented Architecture*) utilise les technologies Web afin d'établir et de gérer les communications entre ses composants appelés services Web. SOAP [2] (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 [3] (*Web Services Description Language*), BPEL [4] (*Business Process Execution Language*) et WSCL [5] (*Web Services Conversation Language*) sont utilisés pour décrire les caractéristiques de services Web.

.NET : La plateforme .NET [6] 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 [7], 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.

2. https://www.w3.org/TR/soap
3. http://www.w3.org/TR/wsdl
4. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
5. http://www.w3.org/TR/wscl10
6. http://www.microsoft.com/net
7. http://www.osgi.org

L'architecture d'une application repose sur des paquets (*bundles*) contenant des services qui proposent des fonctionnalités et un ensemble d'interfaces ; une application OSGi est représentée par un ensemble de services qui sont connectés.

BIP : BIP [Basu et al., 2006] (*Behaviour Interaction Priority*) est un modèle basé sur trois couches : *a*) "comportement" (*behaviour*) qui représente la couche spécifiant l'ensemble des transitions, *b*) "interaction" qui est la couche intermédiaire contenant l'ensemble des connecteurs décrivant les transitions qui représentent les comportements, et *c*) "priorité" (*priority*) qui représente l'ensemble des règles de priorité décrivant les politiques d'ordonnancement des interactions.

La norme IEC 61499 : La norme IEC 61499 [Vyatkin, 2011] décrit un modèle distribué de programmation des systèmes automatisés pour diviser les différentes parties d'un processus d'automatisation industrielle et de contrôle de machines complexes en modules fonctionnels. Dans ce modèle, où les composants représentent des blocs fonctionnels, il n'existe pas de liens pour réaliser leur l'assemblage ; les événements et les données sont transmises entre composants au moyen d'une pile.

1.2/ MODIFICATION DYNAMIQUE DES SYSTÈMES À COMPOSANTS

Un système complexe a besoin d'évoluer au cours du temps et il est préférable de limiter l'indisponibilité due à ses évolutions. Nous avons explicité dans la section précédente la notion de composant. Ces composants logiciels peuvent être primitifs (dans ce cas ils encapsulent du code métier) ou composites (et contiennent, dans ce cas, des sous-composants). Ils fournissent des services via leurs interfaces fournies et satisfont leurs dépendances au moyen de leurs interfaces requises. On peut construire un système à composants en assemblant des composants et en liant leurs interfaces au moyen de liens de type "client-serveur" ou délégation. Les reconfigurations dynamiques permettent la modification de l'architecture d'un système à composants pendant son exécution. Leur but est pouvoir rendre un système à composants adaptatif à son environnement tout en minimisant son indisponibilité. Dans cette section, nous décrivons le concept de reconfiguration dynamique, puis nous présentons les systèmes réflexifs et adaptatifs. Enfin nous nous intéressons aux modèles de boucle de contrôle et de boucle MAPE utilisés dans le cadre de systèmes logiciels adaptatifs.

1.2.1/ 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 [Oreizy et al., 1998b].

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 être 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 [Oreizy et al., 1998a] ; 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'interopérabilité 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.

Dans la section suivante, nous présentons la façon dont les reconfigurations dynamiques peuvent s'appliquer à deux types de systèmes (les systèmes réflexifs et les systèmes adaptatifs).

1.2.2/ SYSTÈMES RÉFLEXIFS ET ADAPTATIFS

Dans cette section, nous nous intéressons en particulier à deux types de systèmes : les systèmes réflexifs et les systèmes adaptatifs. Nous présentons chacun de ces systèmes et la façon dont les reconfigurations dynamiques peuvent leurs être appliquées.

Systèmes réflexifs : Un système réflexif [Smith, 1984] est un système capable de raisonner et d'agir sur lui-même [Kiczales et al., 1991]. Un tel système doit donc pouvoir s'observer et se modifier, ce qui correspond respectivement aux capacités d'introspection et d'intercession.

Un système réflexif comporte nécessairement deux niveaux :

— un niveau de base désignant le code implémentant le système, et
— un méta-niveau représentant le code implémentant la réflexion qui peut se décliner en deux types de réflexion :
  — la réflexion structurelle qui permet d'observer et de manipuler la façon dont le logiciel est organisé (e.g., l'observation de la valeur d'une variable peut amener à sa modification), et

— la réflexion comportementale qui s'intéresse à l'exécution de programme (e.g., l'interception d'un appel de fonction pour surcharger son exécution).

La réflexivité est utilisée en programmation objets mais elle est aussi applicable aux modèles à composants [Cazzola et al., 1998]. Dans un modèle à composants réflexif, le niveau de base correspond aux composants applicatifs dotés d'interfaces de réification au méta-niveau. Ces interfaces sont invoquées pendant l'exécution pour observer et modifier la structure et le comportement des composants.

Systèmes adaptatifs : 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 ils se trouve. Pour pouvoir considérer une opération de reconfiguration comme une adaptation, il est nécessaire que le système ait pu "bénéficier" de la reconfiguration d'un point de vue quantitatif et/ou qualitatif. Ce "bénéfice" se mesure selon des critères propres au type du système (e.g., la fiabilité ou la qualité de service).

Un système adaptatif se modifie de manière autonome en fonction du contexte ; si cette modification n'est pas autonome, le système est dit adaptable. D'après [David, 2005], les systèmes adaptatifs nécessitent 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.

Les reconfigurations dynamiques peuvent permettre de répondre au besoin d'opération pour modifier et adapter le système. La section suivant présente le modèle de boucle de contrôle ainsi que le modèle de référence de boucle MAPE utilisés dans le cadre de systèmes logiciels adaptatifs.

1.2.3/ Boucle de contrôle et boucle MAPE

Une particularité des systèmes adaptatifs réside dans le fait que la prise de certaines décisions, habituellement effectuée 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 ainsi qu'un autre modèle de référence : la boucle MAPE. Ces modèles, tous deux utilisés dans le cadre de systèmes logiciels adaptatifs, sont présentés dans cette section.

Le modèle de boucle de contrôle [Dobson et al., 2006, Cheng et al., 2009, de Lemos et al., 2013] représenté figure 1.5 est composé de quatre activités détaillées ci-dessous :

— **Collecte :** Cette activité 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" (*user context*) de l'application.

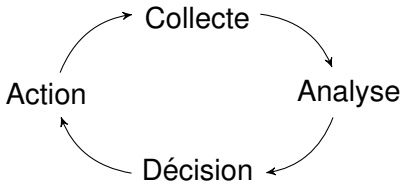


FIGURE 1.5 – Activités d'une boucle de contrôle

— **Analyse :** Les informations collectées sont ensuite analysées en utilisant des raisonnements basés sur l'inférence, mais aussi sur des méthodes moins certaines (e.g., probabilité ou logique floue). Il est aussi possible d'utiliser un ensemble de règles ou de politiques dans la phase d'analyse ou bien des modèles économiques, voire la théorie des jeux. 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 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 consiste à mettre en œuvre la décision prise au cours de la phase précédente en s'appuyant sur des mécanisme propre au système adaptatif lui-même. Cette action et son résultat doivent être enregistrés dans un journal applicatif (*application log*) et, suivant l'impact de la reconfiguration, les utilisateurs ou administrateurs de l'application peuvent être notifiés.

Le modèle de référence de boucle MAPE-K [IBM Corporation, 2006] (*MAPE-K loop reference model*) aussi connu sous le nom de boucle MAPE (*MAPE loop*) a été proposé par IBM pour guider la conception de systèmes adaptatifs [Kephart et al., 2003].

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 de cinq éléments (*Monitor*, *Analyzer*, *Planner*, *Executor*, *Knowledge manager*) représentés figure 1.6. 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.

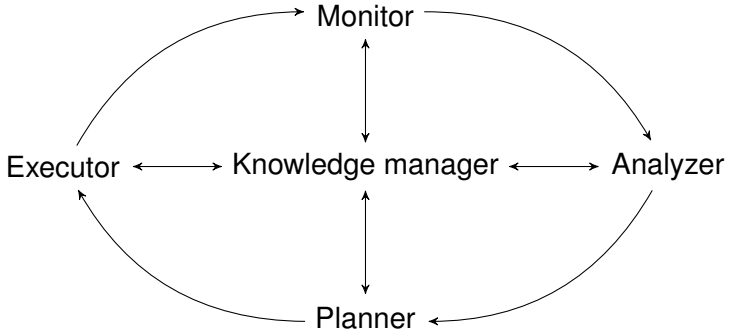


FIGURE 1.6 – Boucle MAPE

Intuitivement, le moniteur (*Monitor*), l'analyseur (*Analyzer*), le planificateur (*Planner*) et l'exécuteur (*Executor*) effectuent respectivement des tâches similaires à celles des activités de collecte, d'analyse, de décision et d'action du modèle de la boucle de contrôle. La différence fondamentale réside dans le rôle du gestionnaire de connaissance (*Knowledge manager*) qui est un élément contenant (a minima) les informations collectées par le moniteur (*Monitor*). Ces informations peuvent être consultées, mises à jours et complétées par tous les éléments de la boucle MAPE.

L'autre différence principale entre la boucle de contrôle et la boucle MAPE est le fait les activités de la boucle MAPE sont effectuées par des éléments distincts, ce qui n'est pas forcément le cas de la boucle de contrôle.

1.3/ LE MODÈLE À COMPOSANTS FRACTAL

Dans cette section, nous présentons le modèle à composants Fractal [Bruneton et al., 2004, Bruneton et al., 2006] que nous utilisons pour illustrer les contributions de cette thèse. Il a été développé par France Télécom R&D et distribué dans le cadre du consortium Objectweb. Fractal est un modèle hiérarchique qui permet la définition, le contrôle et la reconfiguration dynamique d'un système à composants ainsi qu'une séparation des préoccupations fonctionnelles et non-fonctionnelles. Les principaux atouts de ce modèle sont :

— les composants composites qui permettent aux composants de contenir des sous-composants ;

— les composants partagés qui permettent à un composant d'être le sous-composant de plusieurs autres composants ;

— les capacités d'introspection qui permettent d'observer et de contrôler un système en cours d'exécution ; et

— les capacités de reconfiguration qui permettent de déployer et de reconfigurer dynamiquement un système.

1.3.1/ COMPOSANTS FRACTAL

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 interfaces des composants peuvent alors être de deux types suivant qu'elles soient liées au contenu ou à la membrane : les interfaces métier et les interfaces de contrôle. Les interfaces métier correspondent aux points d'accès au composant. Elles correspondent aux interfaces requises et fournies dont nous avons parlé précédemment. Les interfaces de contrôle, quant à elles, 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 1.7, issue du tutorial "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'un composant **Client** et d'un composant **Server**.

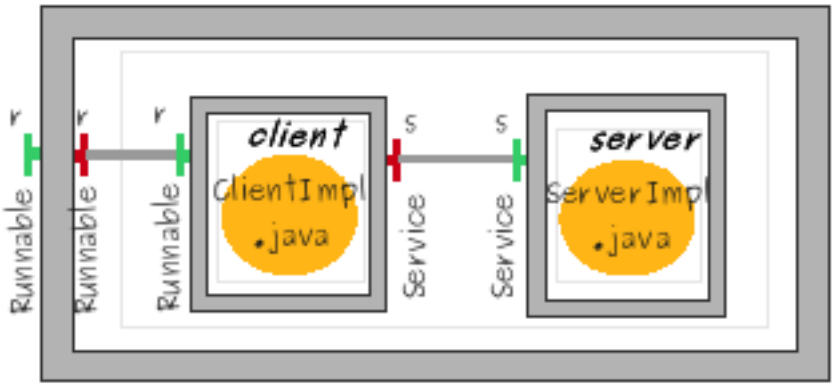


FIGURE 1.7 – Composant Helloworld en Fractal

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 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. Ces contrôleurs soutiennent le caractère dynamique des composants Fractal et permettent facilement de reconfigurer dynamiquement l'architecture d'une application en cours d'exécution. 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.

Fractal ADL (*Architecture Definition Language*) permet de définir, au moyen d'un fichier au format XML, la façon dont un composant composite est assemblé. La figure 1.8 (issue, comme la figure 1.7, du tutorial "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.

8. Par exemple, dans le cadre de l'implémentation de Fractal utilisant Java, le type d'une interface correspond à une interface Java.
9. http://fractal.ow2.org/fractal-distribution/helloworld-julia-fraclet/user-doc.html

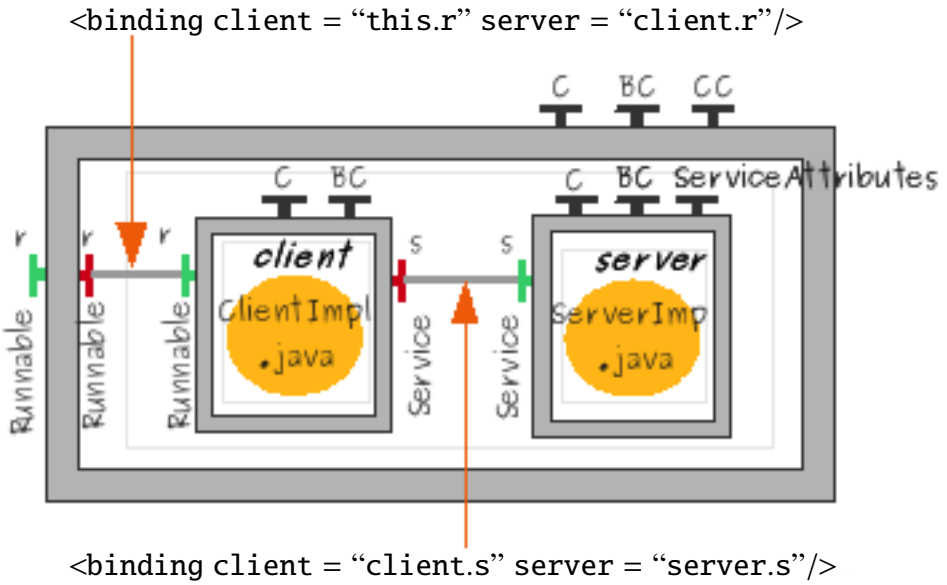


FIGURE 1.8 – Spécification de liens entre interfaces en utilisant Fractal ADL

1.3.2/ 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 [Bruneton et al., 2004] 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 1.8 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 implémentations 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 [Fassino et al., 2002] et Cécilia [12] pour l'implémentation de systèmes embarqués et de systèmes d'exploitation, .NET (FracNet) ou encore smalltalk (FracTalk).

1.4/ LE MODÈLE À COMPOSANTS FRASCATI

Dans cette section, nous présentons la plateforme FraSCAti [Seinturier et al., 2009, Seinturier et al., 2012], un des modèles à composants que nous utilisons pour illustrer les contributions de cette thèse. FraSCAti étant une implémentation de la spécification

10. http://fractal.ow2.org/aokell/index.html
11. http://proactive.inria.fr/
12. http://fractal.objectweb.org/cecilia-site/current/index.html

SCA, nous commençons par présenter cette spécification avant de décrire la façon dont FraSCAti implémente SCA. Enfin, nous comparons cette implémentation à d'autres implémentations de SCA.

1.4.1/ LA SPÉCIFICATION SCA

La collaboration OSOA (*Open Service Oriented Architecture Collaboration*) a regroupé plusieurs acteurs des technologies de l'information dans le but de définir un modèle pour construire et exécuter des applications distribuées de type "architectures orientées service" ou SOA (*Service Oriented Architecture*) indépendamment du langage de programmation utilisé. Les spécifications des premiers modèles à composants (qui avaient été proposées par quelques-uns des fondateurs de l'OSOA) ont servi de base pour le premier ensemble de spécifications SCA (*Service Component Architecture*) qui est cohérent avec les définitions et principes des modèles à composants précédemment énoncés.

Le cœur 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 [Beisiegel et al., 2007]. Ainsi, comme le suggère [Tamura, 2012], 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 e.g., Web services, SOAP, REST, RPC, RMI, etc.).

Cette spécification a été implémentée par plusieurs acteurs des technologies de l'information. On peut citer les implémentations suivantes : WebSphere Application Server for SCA [13] par IBM, Apache Tuscany [14], Fabric3 [15] et bien sûr FraSCAti [16].

La spécification SCA est à présent maintenue par OASIS [17] sous le nom de Open CSA (*Open Composite Services Architecture*).

Enfin, ajoutons que SCA est aussi un des fondements de paradigmes tels que l'"informatique orientée service" [Papazoglou et al., 2007] (*Service Oriented Computing*) et de l'"informatique en nuage" [Merle et al., 2011] (*Cloud Computing*).

1.4.2/ L'IMPLÉMENTATION DE SCA PAR FRASCATI

FraSCAti est une implémentation *open source* de la spécification SCA pour le développement et l'exécution d'applications SCA distribuées. Elle se présente sous la forme d'une pile logicielle intermédiaire (*middleware software stack*) qui, en plus des fonctionnalités SCA standard, est capable de réflexion. Ainsi, ces capacités d'introspection et d'intercession (permettant des reconfigurations primitives e.g., ajout/suppression de composants ou de liens) peuvent être utilisées par toute application SCA exécutée au sein de FraSCAti.

La figure 1.9, extraite de [Seinturier et al., 2012], montre l'architecture de FraSCAti vue

13. https://www.ibm.com/support/knowledgecenter/SSAW57_8.0.0/com.ibm.websphere.nd.doc/info/ae/ae/csca_overview.html
14. http://tuscany.apache.org/
15. http://www.fabric3.org/
16. http://wiki.ow2.org/frascati
17. http://www.oasis-opencsa.org

comme une pile SCA composées de quatre niveaux : noyau (*kernel*), personnalité (*personality*), exécution (*run-time*) et non-fonctionnel (*non-Functional*). Ces niveaux sont décrit ci-dessous :

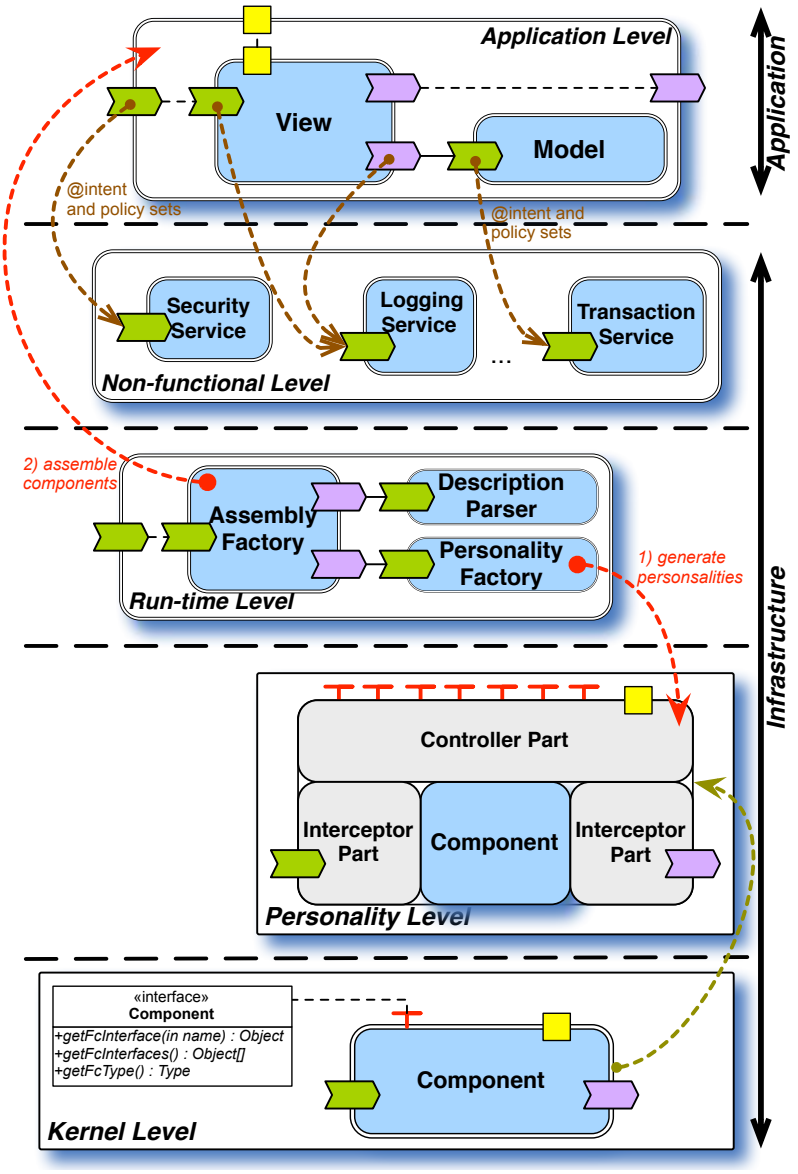


FIGURE 1.9 – Architecture de la plateforme FraSCAti (extrait de [Seinturier et al., 2012])

Niveau noyau : Ce niveau (*Kernel Level* au bas de la figure 1.9) est basé sur le modèle à composant Fractal présenté en section 1.3. Les capacités de réflexion de Fractal permettent d'opérer directement sur la structure des composants en utilisant les contrôleurs mis à disposition par Fractal et supportent les reconfigurations dynamiques.

Niveau personnalité : Comme le montre la figure 1.9, ce niveau (situé au-dessus du niveau noyau) permet l'ajout aux composants noyaux de deux parties intercepteur (*interceptor parts*) — l'une pour l'interface fournie, l'autre pour l'interface requise — et d'une

partie contrôleur (*controller part*). Les intercepteurs sont utilisés pour modifier ou étendre le comportement des invocations de services, tandis que le contrôleur permet la configuration de différentes “facettes” de la “personnalité” du composant, comme par exemple son cycle de vie ou la gestion de liens.

Niveau exécution : Ce niveau (situé au-dessus du niveau personnalité) est responsable de l'instanciation et de l'exécution dans la plateforme d'exécution des composants (composites) SCA. L'analyseur de description (*Description Parser*) charge et vérifie le fichier qui spécifie les composants (composites) de l'application SCA et crée la structure d'exécution correspondante. L'usine d'assemblage (*Assembly Factory*) créé l'assemblage de composants correspondant à la structure d'exécution créée par l'analyseur de description. L'usine de personnalité (*Personality Factory*) ajoute les parties contrôleur et intercepteur à chaque composant en accord avec les spécification du fichier chargé par l'analyseur de description.

Niveau non-fonctionnel : Ce niveau correspond à l'implémentation de divers services permettant de satisfaire des besoins non-fonctionnels des applications exécutées. L'utilisation de ces services peut être paramétrée via des annotations spécifiques dans le fichier chargé par l'analyseur de description du niveau exécution.

1.4.3/ COMPARAISON AVEC D'AUTRES IMPLÉMENTATIONS DE SCA

Les mesures effectuées dans [Seinturier et al., 2009] montrent que la plateforme FraSCAti est capable de performances comparables ou même meilleures que celles de Tuscany [18], la plateforme de référence du monde SCA. De plus, Tuscany n'offre pas de fonctionnalité de reconfiguration dynamique, contrairement à FraSCAti.

L'implémentation de WebSphere Application Server V8.5 for SCA [19] par IBM ne permet pas non plus de reconfigurations dynamiques.

Fabric3 [20] permet des reconfigurations dynamiques en se basant sur ces capacités d'introspection [Romero, 2011], mais contrairement à FraSCAti, ne fournit pas de solution où tous les éléments de la plateforme sont implémentés suivant la spécification SCA.

1.5/ CONCLUSION

Dans ce chapitre, nous avons introduit le concept de composant en explicitant la notion d'architectures orientées composants, puis nous avons exposé l'idée de reconfiguration dans le cadre des systèmes à composants. Pour finir, nous avons présenté les deux implémentations de systèmes à composants que nous utilisons pour illustrer les contributions de cette thèse ; il s'agit des modèles à composants Fractal et FraSCAti. Dans le chapitre suivant, nous exposons les définitions et notations utilisées dans la suite de ce mémoire.

18. http://tuscany.apache.org/
19. https://www.ibm.com/support/knowledgecenter/SSAW57_8.5.5/com.ibm.websphere.nd.doc/ae/csca_overview.html
20. http://www.fabric3.org/

2

PRÉLIMINAIRES

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

2.1/ 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.

2.1.1/ ENSEMBLES

Pour tout ensemble E, nous notons :

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

— $\mathcal{P}(E)$ ou 2^E, l'ensemble des parties de E,

— $\varnothing$, l'ensemble vide.

Pour deux ensembles E et F, nous notons :

— $E \backslash F$, 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,

Sommaire

— $\bar{E}^F$, le complémentaire de E dans F, i.e., $F \backslash E$.

Pour tout entier n, nous notons :

— $E_1 \times \cdots \times E_n$ ou $\prod_{i=1}^{n} E_i$, le produit cartésien des ensembles $E_1, \ldots, E_n$ défini comme l'ensemble $\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$,

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

2.1.2/ RELATIONS ET FONCTIONS

Une relation (binaire) d'un ensemble E dans un ensemble F est une partie $\mathcal{R}$ de $E \times F$. Soient $e \in E$, $f \in F$ d'une part et $\mathcal{R} \subseteq (E \times F)$, une relation de E dans 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, f est une image de e par $\mathcal{R}$ et e est un antécédent de f par $\mathcal{R}$. La relation réciproque $\mathcal{R}^{-1}$ est définie par l'ensemble $\{(f, e) \in F \times E \mid (e, f) \in \mathcal{R}\}$.

Définition 1 : Domaine et codomaine d'une relation

Soit $\mathcal{R}$ une relation de E dans 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 :

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

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

$$codom(\mathcal{R}) = \{f \in F \mid \exists e.(e \in E \wedge (e, f) \in \mathcal{R})\}$$

Définition 2 : Composition de deux relations

Soient $\mathcal{R} \subseteq E \times F$ et $\mathcal{R}' \subseteq F \times G$ deux relations respectivement de E dans F et de F dans G. La composition de $\mathcal{R}$ et $\mathcal{R}'$, notée $\mathcal{R} \circ \mathcal{R}'$ est une relation $\mathcal{R}'' \subseteq E \times G$ définie par :

$$\{(e, g) \in E \times G \mid \exists f.(f \in F \wedge (e, f) \in \mathcal{R} \wedge (f, g) \in \mathcal{R}')\}$$

Définition 3 : 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 $\forall e_1, e_2.(e_1, e_2 \in E \wedge e_1\mathcal{R}e_2 \Rightarrow e_2\mathcal{R}e_1)$

— $\mathcal{R}$ est antisymétrique si $\forall e_1, e_2.(e_1, e_2 \in E \wedge e_1\mathcal{R}e_2 \wedge e_2\mathcal{R}e_1 \Rightarrow e_1 = e_2)$

— $\mathcal{R}$ est transitive si $\forall e_1, e_2, e_3.(e_1, e_2, e_3 \in E \wedge e_1\mathcal{R}e_2 \wedge e_2\mathcal{R}e_3 \Rightarrow e_1\mathcal{R}e_3)$

Définition 4 : Fermeture transitive d'une relation et relation acyclique

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

— R^+ est transitive,

— $R \subset R^+$,

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

Une relation est dite acyclique si

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

Définition 5 : 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 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 $(e, f) \in \varphi$.

Définition 6 : Fonction totale et partielle

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

Définition 7 : Restriction

Soient f une application de E dans F et E' un sous-ensemble de E. La restriction de f à E', notée $f_{|E'}$, est l'application de E' dans F qui à tout élément e de E' associe l'élément $f(e)$ de F.

Définition 8 : 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.

Définition 9 : Signature et profil

Une signature Sig est un couple $\langle E, F \rangle$ dans lequel E est un ensemble non vide de sortes et F est un ensemble non vide de symboles de relations, tel que $E \cap F = \emptyset$. Pour chaque élément $f \in F$, nous associons une séquence $(e_1, \ldots, e_n, e)$ d'éléments de E appelée profil de f et notée $f \subseteq e_1 \times \cdots \times e_n \times e$ ou, si f est un relation fonctionnelle, $f : e_1 \times \cdots \times e_n \to e$.

2.2/ SYSTÈMES DE TRANSITIONS

Cette section a pour but de présenter des modèles sémantiques souvent utilisés par la suite pour décrire le comportement de systèmes. Nous allons nous intéresser aux systèmes de transitions (étiquetés), aux structures de Kripke ainsi qu'aux systèmes de transitions doublement étiquetés.

Les systèmes de transitions vont nous permettre par la suite de spécifier le comportement de systèmes à composants. Nous les définissons ci-dessous.

Définition 10 : Système de transitions ST

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

- Q est un ensemble d'états,
- Q_0 est un ensemble d'états initiaux tel que $Q_0 \subseteq Q$,
- $\rightarrow \subseteq Q \times Q$ 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 système de transitions étiquetés :

Définition 11 : Système de transitions étiqueté ST_E

Un système de transitions étiqueté est un n-uplet $\langle Q, Q_0, E, \rightarrow \rangle$ 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 transitions.

Un ST_E est fini si les ensembles Q et E sont finis. Dans le cas contraire, le système est infini.

Exemple 1 : Système de transitions étiqueté

La figure 2.1 représente un ST_E a trois états q_0, q_1 et q_2 pouvant effectuer quatre actions e_1, e_2, e_3 et e_4. Les cercles représentent les états du système et les flèches orientées d'un état source vers un état destination représentent les transitions. Une transition sans état source (près du mot $start$) permet de pointer les états initiaux du système.

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

Les comportements d'un système de transition peuvent être décrits par un chemin décrivant l'ensemble des états et des transitions qui ont été exécutés.

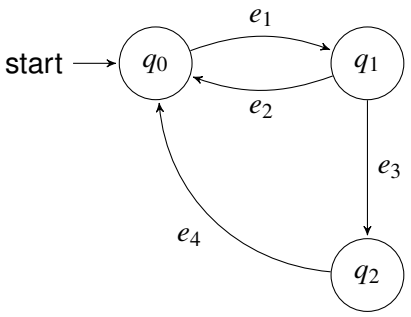


FIGURE 2.1 – Exemple d'un système de transitions étiqueté

Définition 12 : Chemin

Soit $SE = \langle Q, Q_0, E, \rightarrow \rangle$ un ST_E. Un chemin σ de SE est une séquence (potentiellement infinie) d'états $q_0, q_1, q_2, \ldots$ telle que

$$\forall i.(i > 0 \Rightarrow \exists e.(e \in E \wedge q_{i-1} \xrightarrow{e} q_i \in \rightarrow))$$

Notation : Nous notons une telle séquence $\sigma = q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \ldots q_i \xrightarrow{e_{i+1}} q_{i+1} \ldots$. Nous utilisons $\sigma(i)$ pour décrire le i-ème état du chemin σ. La notation σ_i décrit le suffixe du chemin contenant $\sigma(i), \sigma(i+1), \ldots$ et σ_i^j le segment du chemin contenant $\sigma(i), \sigma(i+1), \sigma(i+2), \ldots, \sigma(j-1), \sigma(j)$. Le segment d'un chemin est infini en longueur quand la dernière partie de ce segment est répétée infiniment souvent. Nous notons $\Sigma(SE)$ (ou Σ, s'il n'y a pas d'ambiguïté) l'ensemble des chemins d'évolution sur SE et $\Sigma^f (\subseteq \Sigma)$ l'ensemble des chemins finis sur SE. par ailleurs, nous notons $\xrightarrow{w}$ l'extension aux mots de la relation de transition $\rightarrow$, c'est-à-dire $\sigma(i) \xrightarrow{w} \sigma(i+n)$ avec $w = e_{i+1} \ldots e_{i+n}$ correspondant au chemin σ_i^{i+n} tel que $\sigma_i^{i+n} = \sigma(i) \xrightarrow{e_{i+1}} \sigma(i+1) \xrightarrow{e_{i+2}} \ldots \sigma(i+n-1) \xrightarrow{e_{i+n}} \sigma(i+n)$. Une exécution π est un chemin maximal [1] dans Σ dont le premier état $\pi(0)$ est un des états initiaux de SE : $\pi(0) \in Q_0$. $\Pi(SE)$ désigne l'ensemble de toutes les exécutions de SE.

Exemple 2 : Partie d'un chemin d'évolution

La figure 2.2 représente une partie d'un chemin d'évolution possible du système de transitions de la figure 2.1.

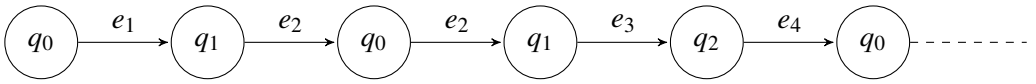


FIGURE 2.2 – Exemple d'une partie d'un chemin d'évolution

Définition 13 : Ensemble d'états atteignables

Soit $SE = \langle Q, Q_0, E, \rightarrow \rangle$ un ST_E. Pour tout état $q \in Q$, l'ensemble des états atteignables en utilisant un chemin fini depuis q est noté $\texttt{reach}(q)$. Formellement, $\texttt{reach}(q) = \{q' \in Q \mid \exists \sigma \Sigma^f, \exists i, j \in \mathbb{N}.i < j \wedge \sigma(i) = q \wedge \sigma(j) = q'\}$. En appliquant cette notion aux ensembles, on a $\texttt{reach}(Q) = \{\texttt{reach}(q) \mid q \in Q\}$.

1. Un chemin est maximal s'il est infini ou fini non prolongeable, i.e., aucune transition ne peut être effectuée depuis son dernier état.

Afin de pouvoir raisonner sur le comportement des systèmes à composants, nous estimons intéressant de retrouver des variables du systèmes (toutes ou une partie) au niveau du modèle utilisé.

Nous considérons $V = \{v_1, \ldots, v_n\}$ un ensemble fini de variables $v_1, \ldots, v_n$ et leurs domaines respectifs $\mathbb{D}_1, \ldots \mathbb{D}_n$. Nous considérons AP_V comme un ensemble de propositions atomiques qui sont formées à partir de V définit formellement par $AP_V \stackrel{\text{def}}{=} \{v_i = d_i \mid v_i \in V \wedge d_i \in \mathbb{D}_i\}$. Dans ce cadre, une structure de Kripke est un système de transitions dans lequel chaque état est étiqueté par l'ensemble des propositions atomiques qui sont valides dans cet état.

Définition 14 : Structure de Kripke ST_K

Une structure de Kripke est un n-uplet $\langle Q, Q_0, \rightarrow, V, l \rangle$ où :

- Q est un ensemble d'états,
- Q_0 est un ensemble d'états initiaux tel que $Q_0 \subseteq Q$,
- $\rightarrow \subseteq Q \times Q$ est une relation de transitions,
- V est un ensemble de variables,
- $l : Q \rightarrow 2^{AP_V}$ est une fonction d'interprétation d'états qui associe à chaque état du système l'ensemble des propositions atomiques.

Pour éviter les blocages, la relation de transition doit être totale [Clarke et al., 1999]. La structure de Kripke peut également être finie ou infinie.

Exemple 3 : Structure de Kripke

La figure 2.3 représente une structure de Kripke dans laquelle une variable $a \in \{0, 1, 2\}$ est prise en compte. Les états contiennent la valeur courante de la variable a.

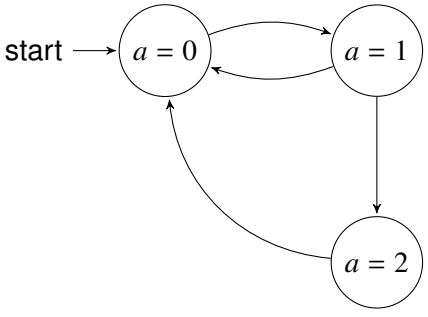


FIGURE 2.3 – Exemple d'un structure de Kripke

Comme dans [De Nicola et al., 1995, Lanoix, 2005, Dormoy, 2011], afin de bénéficier des avantages des deux approches, nous pouvons combiner l'interprétation des états avec l'étiquetage des transitions.

Définition 15 : Système de transitions doublement étiqueté ST_{2E}

Un système de transition doublement étiqueté est un n-uplet $\langle Q, Q_0, E, \rightarrow, V, l\rangle$ 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 transitions,
- V est un ensemble de variables,
- $l : Q \rightarrow 2^{AP_V}$ est une fonction d'interprétation des états qui associe à chaque état du système un ensemble de propositions atomiques.

Exemple 4 : Système de transition doublement étiqueté

La figure 2.4 représente un système de transition doublement étiqueté combinant le système de transition de la figure 2.1 et la structure de Kripke de la figure 2.3.

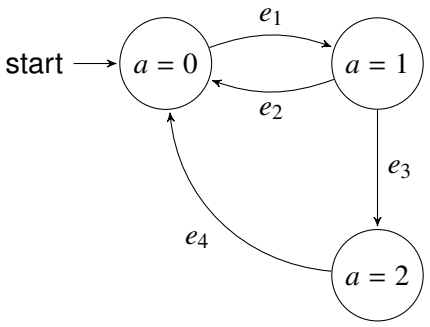


FIGURE 2.4 – Exemple d'un système de transition doublement étiqueté

2.3/ SPÉCIFICATION DES PROPRIÉTÉS DES SYSTÈMES

Cette section porte sur les propriétés des systèmes pour lesquelles nous distinguons, d'une part, les propriétés statiques spécifiées par des invariants ou des formules de la logique du premier ordre et, d'autre part, les propriétés temporelles spécifiées grâce à des logiques temporelles.

2.3.1/ PROPRIÉTÉS D'INVARIANCE

Les propriétés d'invariance permettent d'exprimer des propriétés dites statiques qui doivent être vérifiées dans tous les états du système considéré. Pour cela, nous considérons l'ensemble $SP_V \stackrel{\text{def}}{=} \{sp_0, sp_1, \ldots\}$ de formules d'états formées à partir d'un ensemble de propositions atomiques AP_V. La grammaire de SP_V est formellement définie par :

$$sp_1, sp_2 ::= ap \mid sp_1 \vee sp_2 \mid \neg sp_1 \text{ où } ap \in AP_V$$

Définition 16 : Satisfaction d'une formule d'état

Une formule d'état $sp \in SP_V$ est satisfaite par un état q d'un système de transitions doublement étiqueté $S = \langle Q, Q_0, E, \rightarrow, V, l \rangle$, noté $q \models sp$, si

— $q \models ap$ si $ap \in l(q)$,

— $q \models \neg sp$ s'il n'est pas vrai que $q \models sp$,

— $q \models sp_1 \vee sp_2$ si $q \models sp_1$ ou si $q \models sp_2$.

Une formule d'état $sp \in SP_V$ est alors un invariant satisfait par S, noté $S \models sp$, si

$$\forall q.(q \in Q \Rightarrow q \models sp)$$

Remarque : La sémantique des opérateurs $\wedge$, $\Rightarrow$ et $\Leftrightarrow$ peut être déduite grâce aux règles de réécriture suivantes : $sp_1 \wedge sp_2 \equiv \neg(\neg sp_1 \vee \neg sp_2)$, $sp_1 \Rightarrow sp_2 \equiv \neg sp_1 \vee sp_2$ et $sp_1 \Leftrightarrow sp_2 \equiv (sp_1 \Rightarrow sp_2) \wedge (sp_2 \Rightarrow sp_1)$.

Exemple 5 : Invariant

Dans l'exemple du système de transition doublement étiqueté de la figure 2.4, nous considérons, par exemple, deux autres variables $b, c \in V$. Un invariant pourrait être que lorsque la variable a est égale à 2 alors b et c seraient aussi égales à 2. L'invariant ainsi spécifié serait :

$$(a = 2) \Rightarrow (b = 2 \wedge c = 2)$$

2.3.2/ LOGIQUE DU PREMIER ORDRE

Les propriétés d'invariance proposées dans la section précédente ne sont pas assez expressives pour exprimer, par exemple, des contraintes sur une architecture à composants comme nous envisageons de le faire dans la suite de ce document. C'est pourquoi, nous nous intéressons, dans cette section, à la logique du premier ordre qui permet de spécifier des propriétés sur les ensembles et les relations.

2.3.2.1/ SYNTAXE

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

Définition 17 : Langage du premier ordre

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

- Les connecteurs logiques $\wedge, \vee, \neg, \Rightarrow$,
- Les quantificateurs $\exists, \forall$,
- Un ensemble infini dénombrable de variables $V = \{v_1, v_2, v_3, \ldots\}$,
- Un ensemble dénombrable éventuellement vide de symboles de relations $\mathcal{S}_\mathcal{R} = \{r_1, r_2, r_3, \ldots\}$ 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 $\mathcal{S}_\mathcal{F} = \{f_1, f_2, f_3, \ldots\}$ 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 $\mathcal{S}_C = \{c_1, c_2, c_3, \ldots\}$ (si c_1 est une constante alors nous définissons que $ar(c_1) = 0$),
- Un ensemble dénombrable éventuellement vide de symboles de prédicats $\mathcal{S}_\mathcal{P} = \{P, Q, R, \ldots\}$ 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 18 : Termes

L'ensemble des termes, notés $\mathcal{T} = \{t_1, t_2, t_3, \ldots\}$ est défini inductivement par :

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

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

Définition 19 : Formules

Soient $t_1, \ldots, t_n$ des termes et P un symbole de prédicat tel que $ar(P) = n$, $P(t_1, \ldots, t_n)$ est une formule atomique. L'ensemble $\mathcal{F} = \{A, B, C, \ldots\}$ 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.

Nous nous intéressons à présent à la notion de variables libres d'un terme ou d'une formule qui correspond à l'ensemble des variables apparaissant dans ce terme ou cette formule sans être capturés par un quantificateur.

Définition 20 : Variable libre et liée

L'ensemble des variables libres d'un terme t ou d'une formule A, noté $fv(t)$ ou $fv(A)$ est défini inductivement par :

- $fv(v) = \{v\}$ si v est une variable,
- $fv(f(t_1, \ldots, t_n)) = fv(t_1) \cup \cdots \cup fv(t_n)$,
- $fv(P(t_1, \ldots, t_n)) = fv(t_1) \cup \cdots \cup fv(t_n)$,
- $fv(\neg A) = fv(A)$,
- $fv(A \wedge A') = fv(A \vee A') = fv(A \Rightarrow A') = fv(A) \cup fv(A')$,
- $fv(\exists v.A) = fv(\forall v.A) = fv(A) \backslash \{v\}$.

A l'inverse, l'ensemble des variables liées à une formule A, noté $bv(A)$, est l'ensemble des variables qui ne sont pas libres dans A.

2.3.2.2/ 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 21 : 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, $I(f) : E^n \to E$,
- 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, $I(P) : E^n \to \mathbb{B}$.

Une fois l'interprétation I définie, nous définissons une affectation qui donne la valeur des variables libres dans les formules.

Définition 22 : Affectation

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

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

Nous nous intéressons à la sémantique des termes dans laquelle l'interprétation sert à évaluer les variables.

Définition 23 : Sémantique des termes

La valeur d'un terme t par rapport à une structure d'interprétation $\mathcal{SI} = (E, I)$ et à une affectation ρ, notée $[t]_{\mathcal{SI},\rho}$ est définie inductivement par :

- $[v]_{\mathcal{SI},\rho} = \rho(v)$ si v est une variable,
- $[c]_{\mathcal{SI},\rho} = I(c)$ si c est une constante,
- $[f(t_1, \ldots, t_n)]_{\mathcal{SI},\rho} = I(f)([t_1]_{\mathcal{SI},\rho}, \ldots, [t_n]_{\mathcal{SI},\rho})$ si f est un symbole de fonction.

Nous définissions maintenant la sémantique des formules. Dans cette définition, nous utilisons l'indice $bool$ sur certains des opérateurs pour exprimer que ceux-ci se rapportent aux opérateurs booléens et non aux opérateurs sur les ensembles.

Définition 24 : Sémantique des formules

La valeur de vérité d'une formule A par rapport à une structure d'interprétation $\mathcal{SI} = (E, I)$ et à une affectation ρ, notée $[A]_{\mathcal{SI},\rho}$ est définie inductivement par :

- $[P(t_1, \ldots, t_n)]_{\mathcal{SI},\rho} = I(P)([t_1]_{\mathcal{SI},\rho}, \ldots, [t_n]_{\mathcal{SI},\rho})$ si P est un symbole de prédicat,
- $[\neg A]_{\mathcal{SI},\rho} = \neg_{bool}[A]_{\mathcal{SI},\rho}$,
- $[A \wedge A']_{\mathcal{SI},\rho} = [A]_{\mathcal{SI},\rho} \wedge_{bool} [A']_{\mathcal{SI},\rho}$,
- $[A \vee A']_{\mathcal{SI},\rho} = [A]_{\mathcal{SI},\rho} \vee_{bool} [A']_{\mathcal{SI},\rho}$,
- $[A \Rightarrow A']_{\mathcal{SI},\rho} = [A]_{\mathcal{SI},\rho} \Rightarrow_{bool} [A']_{\mathcal{SI},\rho}$,
- $[\forall v.A]_{\mathcal{SI},\rho} = \top$ si $\forall e.(e \in E \Rightarrow [A]_{\mathcal{SI},\rho[v:=e]} = \top)$,
- $[\exists v.A]_{\mathcal{SI},\rho} = \top$ si $\exists e.(e \in E \wedge [A]_{\mathcal{SI},\rho[v:=e]} = \top)$.

Nous nous intéressons maintenant à satisfiabilité et à la validité d'une formule. Soient $\mathcal{SI} = (E, I)$ une structure d'interprétation et A une formule sur $\mathcal{SI}$. Nous disons que A est satisfiable dans $\mathcal{SI}$ s'il existe une valuation ρ telle que $[A]_{\mathcal{SI},\rho} = \top$; A est insatisfiable dans le cas contraire, i.e., si A est évalué à $\bot$ dans toute interprétation et toute affectation. A est valide dans $\mathcal{SI}$ si pour tout valuation ρ, $[A]_{\mathcal{SI},\rho} = \top$. Dans ce cas, nous disons que $\mathcal{SI}$ est un modèle de A et nous notons $\mathcal{SI} \models A$. L'ensemble des modèles de A est noté $mod(A)$.

2.3.3/ LOGIQUES TEMPORELLES

Les invariants et les formules de la logique du premier ordre présentés ci-dessus ne sont pas suffisants pour exprimer des 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 [Clarke et al., 1999]. Elles peuvent être distinguées en fonction des critères suivants :

- les logiques faisant référence aux états :
  - les logiques linéaires telles que la logique temporelle linéaire LTL [Manna et al., 1992] (pour Linear Temporal Logic) et le μ-calcul linéaire [Vardi, 1988] expriment des propriétés sur les séquences d'exécution du modèle,

  - les logiques arborescentes telles que CTL [Clarke et al., 1986] et CTL* [Emerson et al., 1986] (pour Computation Tree Logic) expriment des propriétés sur les arbres d'exécution du modèle,

- les logiques faisant référence aux actions telles que la logique temporelle des actions TLA [Lamport, 1996] (pour Temporal Logic of Actions) ou le μ-calcul modal [Kozen, 1983].

Nous nous intéressons plus particulièrement dans le cadre de nos travaux à la logique temporelle linéaire propositionnelle PLTL (pour Propositional Linear Temporal Logic) pour décrire des propriétés temporelles.

Définition 25 : Syntaxe de PLTL

Une formule de la PLTL est donnée par la grammaire suivante :

$$\phi, \phi' ::= ap \mid \neg\phi \mid \phi \vee \phi' \mid \bigcirc\phi \mid \phi \,\mathcal{U}\phi'$$

La formule $\bigcirc\phi$ (**next**) indique que ϕ est vraie dans l'état suivant de l'exécution. La formule $\phi \,\mathcal{U}\phi'$ (ϕ **until** ϕ') signifie que ϕ est vraie jusqu'à ce que ϕ' soit vraie. Les formules logiques habituelles $\phi \wedge \phi'$, $\phi \Rightarrow \phi'$ et $\phi \Leftrightarrow \phi'$ peuvent être utilisées ainsi que les combinateurs temporels $\Diamond$ (**eventually**), $\Box$ (**always**) et $\mathcal{W}$ (**unless**) suivants :

- $\Diamond\phi := \top \,\mathcal{U}\phi$,
- $\Box\phi := \neg(\Diamond\neg\phi)$,
- $\phi\mathcal{W}\phi' := (\phi \,\mathcal{U}\phi') \vee \Box\phi$.

La formule $\Diamond\phi$ indique que ϕ est fatalement vraie dans un état futur de l'exécution. La formule $\Box\phi$ indique que ϕ est toujours vraie dans tous les états futurs de l'exécution. La formule $\phi\mathcal{W}\phi'$ signifie que ϕ est vraie dans tous les états futurs jusqu'à un éventuel état où ϕ' est vraie.

La satisfaction d'une formule PLTL est donnée classiquement pour une structure de Kripke mais nous pouvons la donner pour un système de transitions doublement étiqueté car il est possible de le ramener à une structure de Kripke par projection. Soit $S = \langle Q, Q_0, E, \rightarrow, V, l\rangle$ un système de transitions doublement étiqueté, alors $\mathrm{SK}(S) = \langle Q, Q_0, \rightarrow', V, l\rangle$ est la structure de Kripke associée à S avec $\rightarrow' \subseteq Q \times Q$ étant une projection de $\rightarrow \subseteq Q \times E \times Q$ sur $Q \times Q$ définie par :

$$\forall e \in E.((q, e, q') \in \rightarrow \Rightarrow (q, q') \in \rightarrow')$$

La satisfaction d'une formule PLTL ϕ est définie inductivement sur la structure de ϕ.

Définition 26 : Sémantique de PLTL

Soient S un système de transitions doublement étiqueté, π une exécution de S et $|\pi|$ désigne la longueur de π. La formule PLTL ϕ est satisfaite sur le i-ème état de l'exécution π $(i \geq 0)$, noté $\pi(i) \models \phi$, si

- $\pi(i) \models ap$ si $ap \in l(\pi(i))$,
- $\pi(i) \models \neg\phi$ si $\neg(\pi(i) \models \phi)$,
- $\pi(i) \models \phi \vee \phi'$ si $\pi(i) \models \phi$ ou $\pi(i) \models \phi'$,
- $\pi(i) \models \bigcirc\phi$ si $\pi(i+1) \models \phi$,
- $\pi(i) \models \phi \,\mathcal{U}\phi'$ s'il existe j tel que $i \leq j \leq |\pi|$ et $\pi(j) \models \phi'$ et pour tout k tel que $i \leq k < j$,$\pi(k) \models \phi$.

Une formule PLTL ϕ est satisfaite par π, noté $\pi \models \phi$ si $\forall i \in \mathbb{N}.\pi(i) \models \phi$; une telle formule ϕ est satisfaite par S, noté $S \models \phi$ si

$$\forall \pi.(\pi \in \Pi(S) \Rightarrow \pi \models \phi)$$

Le lecteur intéressé par la complexité des algorithmes de vérification pourra, par exemple, consulter [Clarke et al., 1999] ou [Bérard et al., 2013].

Exemple 6 : Propriétés exprimées avec PLTL

Voici deux exemples de propriétés exprimées avec PLTL :

- Une première propriété exprime la propriété d'invariance de l'exemple 5 en PLTL :
$$\Box((a = 2) \Rightarrow (b = 2 \wedge c = 2))$$
- Une deuxième propriété PLTL exprime, sur l'exemple de la figure 2.3, que quand la variable a est égale à 1, dans l'état suivant cette variable doit être égale à 2 :
$$\Box((a = 1) \Rightarrow \bigcirc(a = 2))$$

II

MODÉLISATION DES RECONFIGURATIONS DYNAMIQUES

3

MODÉLISATION

Nous proposons, comme dans [Dormoy, 2011] de décrire la sémantique d'un système à composants en utilisant un système de transition. Pour ce faire, nous présentons dans un premier temps, pour servir de base aux exemples utilisés, le composant composite **Location** utilisé pour le positionnement du Cycab, un véhicule autonome. Ensuite, nous définissons une sémantique opérationnelle permettant de décrire l'architecture des systèmes à composants ainsi que les reconfigurations primitives qui sont des opérations d'évolution basiques permettant de passer d'une configuration architecturale à une autre pour définir les systèmes à composants reconfigurables primitifs. Enfin, nous abordons la notion de reconfiguration non primitive pour définir les systèmes à composants reconfigurables (non primitifs) avant de conclure ce chapitre.

3.1/ LE COMPOSANT **LOCATION**

Les systèmes de positionnement représentent une partie critique pour le bon fonctionnement des véhicules autonomes. Afin d'illustrer notre modèle, nous proposons en guise d'exemple un système à composants permettant à un véhicule autonome d'obtenir sa

Sommaire

position. Ce système composite, **Location**, est inspiré du système de positionnent du Cycab [Baille et al., 1999], un véhicule autonome développé par l'équipe IMARA (Informatique Mathématiques et Automatique pour la Route Automatisée) d'Inria dans le cadre du projet CityMobil [1].

Dans cette section, nous présentons brièvement les systèmes de géolocalisation puis nous nous intéresserons plus particulièrement au Cycab avant d'expliciter le composant **Location** qui nous servira d'exemple pour illustrer notre modèle.

3.1.1/ LES SYSTÈMES DE GÉOLOCALISATION

Un système de géolocalisation a pour fonction de fournir la position d'un objet à l'aide de coordonnées géographiques. Les principales techniques pouvant être utilisées pour obtenir une telle position utilisent des satellites ou un réseau Wi-Fi pour obtenir une précision de l'ordre de quelques mètres. Il est aussi possible de géolocaliser un terminal mobile en utilisant le réseau GMS, mais dans ce cas la précision est moindre (de l'ordre de la centaine de mètres). L'utilisation de l'adresse ip d'un terminal peut aussi fournir sa position mais avec seulement une précision de l'ordre de la ville. Il est aussi possible d'employer d'autres systèmes utilisant un réseau Bluetooth, des senseurs RFID ou des points de repères prédéfinis.

Parmi les systèmes de positionnement par satellite, aussi appelés GNSS (Global Navigation Satellite System), nous pouvons citer GPS, GALILEO ou GLONASS. De nos jours, presque tous les téléphones mobiles sont équipés de récepteurs GNSS. Néanmoins les GNSS ne sont utilisables que dans certaines conditions ; par exemple, il est souvent impossible d'obtenir une position fiable à l'intérieur d'un immeuble.

C'est pourquoi des solutions alternatives ont vu le jour. Il est possible, par exemple, d'établir une relation entre l'intensité du signal émis par une borne Wi-Fi et la distance physique de laquelle on se trouve de cette borne. Dans ce cas, l'idée consiste à se baser sur des infrastructures existantes et des points d'accès Wi-Fi fixes.

Il est fréquent d'utiliser différentes techniques en privilégiant celle qui apportera la précision souhaitée. La plupart des téléphones utilisent un récepteur GNSS, mais sont aussi capables d'obtenir leur position en utilisant le réseau GSM ou des bornes d'accès Wi-FI.

3.1.2/ PRÉSENTATION DU CYCAB

Le Cycab est un véhicule électrique autonome développé par Inria Rhône-Alpes. Il s'agit d'une première version d'un système de transport urbain de véhicules autonomes accessibles en libre service. Ce système est basée sur l'idée de convoi de véhicules autonomes (voir Figure 3.1) conçus pour des zones où la circulation automobile est limitée, comme les hyper-centres urbains ou les aéroports. Un terminal permet à l'utilisateur d'indiquer sa destination et peut fournir des renseignements touristiques ou commerciaux en fonction du contexte.

1. http://www.citymobil-project.eu/

La figure 3.2 extraite du dépliant[2] du Cycab, illustre les principales caractéristiques de son fonctionnement. Un système GPS différentiel permet d'obtenir une précision supérieure à celle obtenue par des systèmes GPS classiques. Le positionnement d'un véhicule autonome en mouvement étant critique, l'utilisation de plus d'un système de positionnement est nécessaire. C'est pourquoi le Cycab est aussi équipé d'un système de positionnement utilisant les réseaux Wi-Fi.

FIGURE 3.1 – Convoi de Cycabs

Dans le cadre de nos travaux nous nous intéressons au composant de géolocalisation du Cycab qui fonctionne en utilisant le système de positionnement GPS, Wi-Fi ou une combinaison GPS+Wi-Fi.

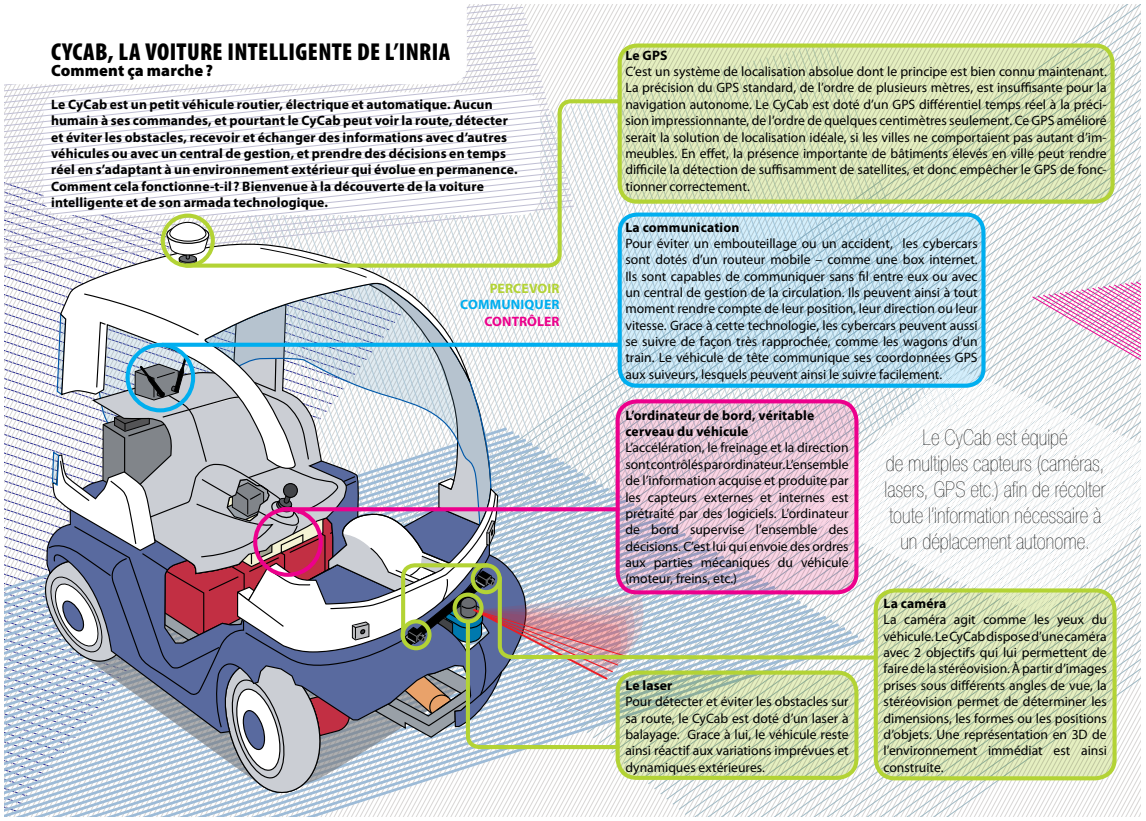


FIGURE 3.2 – Extrait du dépliant du Cycab

3.1.3/ LE COMPOSANT COMPOSITE **LOCATION**

La figure 3.3 représente une vue abstraite d'une version simplifiée du composant composite de géolocalisation du Cycab. Ce composant composite, nommé **Location**, contient

2. http://www.inria.fr/medias/actualites/innovation/documents-pdf/depliant-cybercar

les quatre composants primitifs **Controller**, **Merger**, **GPS** et **Wi-Fi**. Cette version simplifiée a été initialement implémentée [Dormoy et al., 2010] en utilisant le modèle de système à composants Fractal. Nous avons par la suite [Kouchnarenko et al., 2015] implémenté cette même version avec le modèle de système à composants FraSCAti.

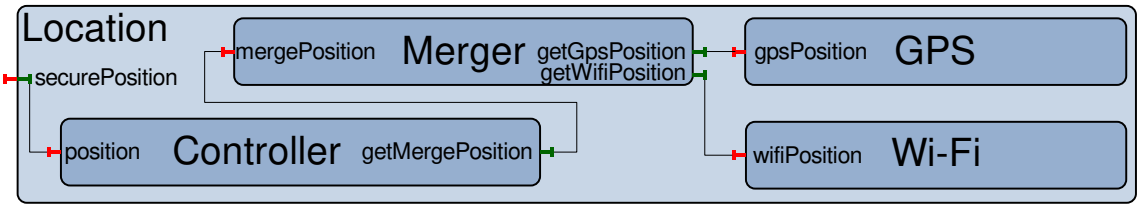


FIGURE 3.3 – Le composant de géolocalisation **Location**

Les composants **GPS** et **Wi-Fi** sont des systèmes logiciels de géolocalisation permettant, respectivement, de collecter les données reçues des systèmes de positionnement GPS et Wi-Fi. Le composant **Merger** fusionne les données obtenues des composants **GPS** et **Wi-Fi** au moyen d'un algorithme particulier permettant d'obtenir un niveau de confiance relatif à la valeur fusionnée. Ce niveau de confiance ne doit pas baisser entre deux opérations de géolocalisation successives. Enfin, le composant **Controller** effectue des requêtes auprès du composant **Merger** et accuse réceptions des données obtenues.

Les composants **GPS** et **Wi-Fi** ont chacun une interface fournie, respectivement nommées $gpsPostion$ et $wifiPostion$, de type $Position$.

Le composant **Merger** contient deux interfaces requises de type $Position$, $getGpsPosition$ et $getWifiPosition$, utilisées respectivement pour requêter des positions auprès des composants **GPS** et **Wi-Fi**. En outre, le composant **Merger** possède aussi une interface fournie de type $MergePosition$ permettant de fournir une position combinée assortie d'un niveau de confiance qui est représenté par un entier positif ou nul. Le niveau de confiance courant est un attribut (ou un paramètre) du composant **Merger** appelé $Trust$. Le composant **Merger** possède deux autres attributs entiers, $GpsPowerUsage$ et $WifiPowerUsage$, représentant respectivement le coût énergétique du fonctionnement des composants **GPS** et **Wi-Fi**.

Le composant **Controller** possède une interface requise $getMergePosition$ de type $MergePosition$, lui permettant de requêter une position fusionnée auprès du composant **Merger**. Il contient un attribut entier, nommé $Power$, qui est un pourcentage représentant le niveau de charge de la batterie utilisée pour le fonctionnement du composant **Location**. Enfin, le composant **Controller** possède une interface fournie $position$, de type $SecurePosition$ permettant de fournir une position combinée assortie d'un niveau de confiance et du pourcentage d'énergie disponible.

Le composant composite **Location** contient les quatre composants primitifs **Controller**, **Merger**, **GPS** et **Wi-Fi** ainsi qu'une interface fournie $securePosition$, de type $SecurePosition$. Les requêtes effectuées sur cette interface, sont transmises (le terme exact est "déléguées") à l'interface $position$ du composant **Controller**.

Au cours du fonctionnement du Cycab, lorsque le niveau de charge de la batterie devient bas, il peut être nécessaire de désactiver ou d'enlever un ou plusieurs composants pour économiser de l'énergie. Dans ce cas, si l'on souhaite supprimer le composant **Wi-Fi**, on peut effectuer les actions suivantes : *a*) arrêt du composant **Location** (ce qui a pour conséquence l'arrêt de ses composants, à savoir : **Controller**, **Merger**, **GPS** et **Wi-Fi**),

b) suppression du lien entre l'interface fournie $wifiPostion$ du composant **Wi-Fi** et l'interface requise $getWifiPosition$ du composant **Merger**, *c*) suppression du composant **Wi-Fi**, *d*) redémarrage du composant **Location**, (ce qui a pour conséquence le démarrage de ses sous-composants, à savoir : **Controller**, **Merger**, et **GPS**).

Si, par exemple, une fois la batterie rechargée, on souhaite rajouter le composant **Wi-Fi**, on peut effectuer les actions suivantes : *a*) ajout du composant **Wi-Fi**, *b*) création du lien entre l'interface fournie $wifiPostion$ du composant **Wi-Fi** et l'interface requise $getWifiPosition$ du composant **Merger** *c*) démarrage du composant **Wi-Fi**.

On notera qu'un composant est soit démarré (état $started$) ou arrêté (état $stopped$) ; bien sûr on peut imaginer des états intermédiaires (e.g., "démarrage en cours" ou "en cours d'arrêt", etc.), mais les états $started$ et $stopped$ permettent de décrire de façon satisfaisante les composants issus des modèles à composant Fractal et FraSCAti.

Dans le même ordre d'idée, on peut imposer à une interface requise d'être toujours liée (à une autre interface) lorsque le composant qui la contient doit être démarré. Néanmoins, il peut être nécessaire qu'un composant puisse être démarré en ayant un ou plusieurs interfaces requises non liées ; c'est le cas du composant **Merger** si, comme évoqué ci-dessus, on supprime le composant **Wi-Fi**. Dans ce cas, si on autorise une interface requise à ne pas être liée lorsque le composant qui la contient est démarré, on lui affecte la contingence $optional$. Dans le cas contraire, on affecte la contingence $mandatory$ à une interface requise dont le composant qui la contient ne doit pas démarrer tant que cette interface n'est pas liée à une autre interface (c'est le comportement par défaut avec Fractal).

3.2/ SÉMANTIQUE OPÉRATIONNELLE

Nous définissons à présent la sémantique opérationnelle permettant de décrire l'architecture des systèmes à composants ainsi que les reconfigurations primitives qui sont des opérations d'évolution basiques permettant de passer d'une configuration architecturale à une autre pour définir les systèmes à composants reconfigurables primitifs.

3.2.1/ ARCHITECTURE DES SYSTÈMES À COMPOSANTS

Afin de définir l'architecture d'un système à composants, nous définissons un modèle architectural basé sur les différents éléments d'un système à composants donné et sur les relations entre ces éléments. Le modèle architectural utilisé dans ce mémoire de thèse est proche de celui défini dans [Dormoy, 2011, Dormoy et al., 2012a], lui-même inspiré de la représentation basée sur les graphes dans [Léger, 2009, Léger et al., 2010] concernant le modèle à composants Fractal.

Bien que le modèle architectural défini ci-dessous soit tout à fait adapté à la description de composants Fractal, il peut aussi décrire d'autres modèles de systèmes à composants comme, par exemple, FraSCAti. Les légères différences de ce modèle par rapport à [Dormoy, 2011, Dormoy et al., 2012a] seront explicitées au fur et à mesure de leurs apparitions. Nous commençons par définir la notion de configuration qui correspond à une signature (voir définition 9) dans laquelle l'ensemble de sortes correspond aux éléments architecturaux, et l'ensemble de symboles de fonctions correspond aux relations architecturales.

Définition 27 : Configuration

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

— $Elem$ est un ensemble d'éléments architecturaux,

— Rel est un ensemble de symboles de relations (fonctionnelles ou non).

Dans notre modèle, les éléments architecturaux sont les entités centrales du système à composants. Ce sont les composants, les interfaces (fournies ou requises), les paramètres et les types.

3.2.1.1/ ÉLÉMENTS ARCHITECTURAUX

Définition 28 : Éléments architecturaux

L'ensemble des éléments architecturaux $Elem$ est défini par :

$$Elem = \{Components, IProvided, IRequired, Parameters, ITypes, PTypes, Contingencies, States, Values\}$$

où $Components$, $IProvided$, $IRequired$, $Parameters$, $ITypes$, $PTypes$, $Contingencies$, $States$ et $Values$ sont des ensembles de différents éléments architecturaux explicités ci-dessous.

— $Components$ est un ensemble non vide d'entités centrales, c'est-à-dire les composants,

— $IProvided$ est un ensemble d'interfaces fournies,

— $IRequired$ est un ensemble d'interfaces requises,

— $Parameters$ est un ensemble de paramètres,

— $ITypes = \{iType_0, iType_1, \ldots, iType_n\}$ est un ensemble fini de types d'interfaces,

— $PTypes = \{pType_0, pType_1, \ldots, pType_m\}$ est un ensemble fini de types de paramètres [3],

— $Contingencies$ est un ensemble fini contenant les valeurs possibles de la contingence d'une interface requise,

— $States$ est un ensemble fini contenant les valeurs possibles de l'état d'un composant,

— $Values \subseteq \bigcup_{i=0}^{m} \{v \mid v \in pType_i\}$ est un sous-ensemble de l'ensemble de toutes les valeurs pouvant être prise par les paramètres de type $t \in PTypes$.

Il est à noter que les éléments architecturaux $Contingencies$, $States$ et $Values$ n'étaient pas des éléments de $Elem$ dans [Dormoy, 2011]. La raison de cet ajout réside dans le fait que ces éléments sont utilisés pour définir le profil (voir définition 9) des relations architecturales contenues dans l'ensemble Rel ; Si ces éléments ne faisaient pas partie de $Elem$, la configuration $\langle Elem, Rel \rangle$ ne serait pas une signature au sens de la définition 9.

L'exemple 7 illustre l'ensemble $Elem$ de la définition 28 pour le composant **Location** défini précédemment dans la section 3.1.3. Les éléments architecturaux $Components$, $IProvided$

3. D'autres modèles de systèmes à composants utilisent le terme "attribut" pour désigner ce que nous appelons "paramètre".

et $IRequired$ peuvent se déduire directement de la représentation, figure 3.3, du composant **Location**. Les éléments $Parameters$, $ITypes$, $PTypes$ et $Values$ sont obtenus en examinant les classes Java instanciées par les composants primitifs du composant **Location**. Les élements $Contingencies$ et $States$ dépendent respectivement des différentes contingences permises sur les interfaces requises par le modèle de système à composant utilisé pour implémenter le composant **Location** et des états que ce modèle permet aux composants de prendre.

Exemple 7 : Éléments architecturaux du composant Location

$$
\begin{array}{lcl}
Components & = & \{controller, gps, location, merger, wifi\} \\
IProvided & = & \{gpsPosition, mergePosition, position, securePosition, wifiPosition\} \\
IRequired & = & \{getGpsPosition, getMergePosition, getWifiPosition\} \\
Parameters & = & \{GpsPowerUsage, Power, Trust, WifiPowerUsage\} \\
ITypes & = & \{MergePosition, Position, SecurePosition\} \\
PTypes & = & \{int\} \\
Contingencies & = & \{optional, mandatory\} \\
States & = & \{started, stopped\} \\
Values & = & \mathbb{Z}
\end{array}
$$

3.2.1.2/ RELATIONS ARCHITECTURALES

Après avoir défini les éléments architecturaux, nous devons spécifier les relations entre ces éléments. Nous définissons ci-dessous l'ensemble Rel des relations entre les éléments architecturaux.

Définition 29 : Relations architecturales

L'ensemble de symboles de relations architecturales Rel est défini par :

$$
\begin{aligned}
Rel = \{&IProvidedType, IRequiredType, Provider, Requirer, Contingency, ParameterType, \\
&Definer, Parent, Descendant, Binding, DelegateProv, DelegateReq, State, Value\}
\end{aligned}
$$

où $IProvidedType$, $IRequiredType$, $Provider$, $Requirer$, $Contingency$, $ParameterType$, $Definer$, $Parent$, $Descendant$, $Binding$, $DelegateProv$, $DelegateReq$, $State$ et $Value$ sont des relations de $Elem \times Elem$ explicitées ci-dessous.

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

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

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

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

— $Contingency : IRequired \rightarrow Contingencies$ est une fonction totale qui associe à chaque interface requise à la valeur de contingence lui correspondant,

— $ParameterType : Parameters \rightarrow PTypes$ est une fonction totale qui associe à chaque paramètre un type de paramètre,

— $Definer : Parameters \rightarrow Components$ est une fonction totale qui associe à chaque paramètre le composant auquel il appartient,

— $Parent \subseteq Components \times Components$ est une relation qui lie un sous-composant au composant qui le contient [4],

— $Descendant \subseteq Components \times Components$ est une relation qui lie un composant aux sous-composants qu'il contient [5],

— $Binding \subseteq IProvided \times IRequired$ est une relation qui est utilisée pour lier entres-elles des interfaces fournies et requises,

— $DelegateProv : IProvided \rightarrow IProvided$ est une fonction partielle et injective qui spécifie la délégation entre une interface fournie d'un sous-composant et une interface fournie du composant qui le contient,

— $DelegateReq : IRequired \rightarrow IRequired$ est une fonction partielle et injective qui spécifie la délégation entre une interface requise d'un sous-composant et une interface requise du composant qui le contient,

— $State : Components \rightarrow States$ est une fonction totale qui indique pour chaque composant son état,

— $Value : Parameters \rightarrow Values$ est une fonction totale qui donne la valeur courante de chaque paramètre.

Précisons que les symboles de relations de Rel on été légèrement modifiés par rapport à [Dormoy, 2011] pour garantir le fait que la configuration $\langle Elem, Rel \rangle$ ne soit une signature au sens de la définition 9. Au delà de ces changement mineurs, deux modifications légèrement plus significatives ont été effectuées : *a*) d'une part, la relation $Binding$ qui était précédemment une relation fonctionnelle ne l'est plus pour pouvoir autoriser plusieurs interfaces requises à être liées à une interface fournie (comportement classique d'une relation client-serveur) et *b*) d'autre part, la relation fonctionnelle $Descendant$ a été ajoutée à Rel.

La figure 3.4 est une représentation sous forme de graphe des différents éléments et relations permettant de spécifier une architecture d'un système à composants. Les rectangles représentent les élément architecturaux et les flèches les relations. On notera qu'une relation fonctionnelle (par exemple, $State$) est représente par une flèche unidirectionnelle ($\rightarrow$) partant de l'ensemble représentant le domaine de la fonction, tandis qu'une relation non fonctionnelle (par exemple, $Parent$) est représentée par une flèche bidirectionnelle ($\leftrightarrow$).

L'exemple 8 illustre l'ensemble Rel de la définition 29 pour le composant **Location** défini précédemment dans la section 3.1.3. Les relations $Provider$, $Requirer$, $Parent$, $Descendant$, $Binding$, $DelegateProv$ et $DelegateReq$ peuvent se déduire directement de la représentation, figure 3.3, du composant **Location**. Les relations $IProvidedType$, $IRequiredType$, $ParameterType$ et $Definer$ sont obtenues en examinant les classes Java instanciées par les composants primitifs du composant **Location**. La relation $Contingency$ dépend de la spécification utilisée pour mettre en place le composant **Location**. Enfin

4. Pour tout $(p, q) \in Parent$, on dit que q a un sous-composant p, i.e., p est un descendant de q. On notera que $Parent$ n'est pas une relation fonctionnelle pour pouvoir décrire des composants partagés (étant des sous-composants de plusieurs composites) qui peuvent avoir plus d'un parent.

5. Pour tout $(p, q) \in Parent$ et $(q, c) \in Parent$, on dit que p est un descendant de q et que p et q sont des descendants de c, i.e., $\{(q, p), (c, q), (c, p)\} \subseteq Descendant$. Comme $Parent$, $Descendant$ n'est pas une relation fonctionnelle.

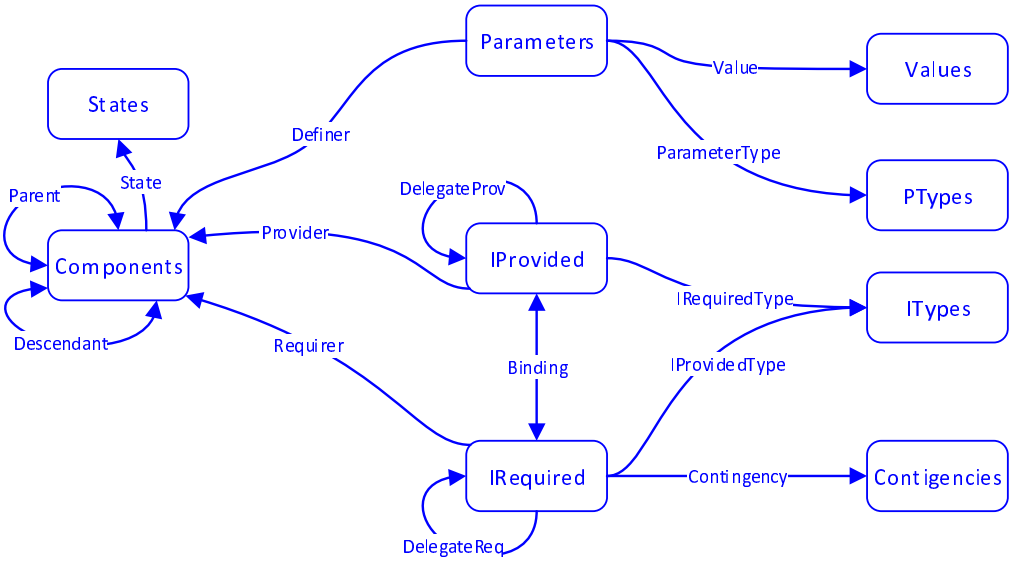


FIGURE 3.4 – Éléments et relations architecturales

les relations $State$ et $Value$, qui peuvent changer durant l'exécution du système à composants, doivent être construites à partir des résultats de requêtes sur le modèle de système à composants utilisé (Fractal ou FraSCAti dans notre cas).

Exemple 8 : Relations architecturales du composant Location

$$
\begin{array}{lcl}
IProvidedType & = & \{gpsPosition \mapsto Position, mergePosition \mapsto MergePosition, \\
 & & position \mapsto SecurePosition, securePosition \mapsto SecurePosition, \\
 & & wifiPosition \mapsto Position\} \\
IRequiredType & = & \{getGpsPosition \mapsto Position, getMergePosition \mapsto MergePosition, \\
 & & getWifiPosition \mapsto Position\} \\
Provider & = & \{gpsPosition \mapsto gps, mergePosition \mapsto merger, position \mapsto controller, \\
 & & securePosition \mapsto location, wifiPosition \mapsto wifi\} \\
Requirer & = & \{getGpsPosition \mapsto merger, getMergePosition \mapsto controller, \\
 & & getWifiPosition \mapsto merger\} \\
Contingency & = & \{getGpsPosition \mapsto optional, getMergePosition \mapsto mandatory, \\
 & & getWifiPosition \mapsto optional\} \\
ParameterType & = & \{GpsPowerUsage \mapsto int, Power \mapsto int, Trust \mapsto int, \\
 & & WifiPowerUsage \mapsto int\} \\
Definer & = & \{GpsPowerUsage \mapsto merger, Power \mapsto controller, Trust \mapsto merger, \\
 & & WifiPowerUsage \mapsto merger\} \\
Parent & = & \{(controller, location), (gps, location), (merger, location), \\
 & & (wifi, location)\} \\
Descendant & = & \{(location, controller), (location, gps), (location, merger), \\
 & & (location, wifi)\} \\
Binding & = & \{(gpsPosition, getGpsPosition), (mergePosition, getMergePosition), \\
 & & (wifiPosition, getWifiPosition)\} \\
DelegateProv & = & \{position \mapsto securePosition\} \\
DelegateReq & = & \varnothing \\
State & = & \{controller \mapsto started, gps \mapsto started, location \mapsto started, \\
 & & merger \mapsto started, wifi \mapsto started\} \\
Value & = & \{GpsPowerUsage \mapsto 3, Power \mapsto 95, Trust \mapsto 0, \\
 & & WifiPowerUsage \mapsto 2\}
\end{array}
$$

3.2.1.3/ SIMPLIFICATIONS POSSIBLES

Il est possible de simplifier la notation décrite ci-dessus en s'inspirant de [Kouchnarenko et al., 2014a].

On peut considérer les simplifications ou réécritures suivantes, où $\uplus$ est l'opérateur d'union disjointe :

— $Interfaces = IProvided \uplus IRequired$,

— $Containers = Interfaces \uplus Parameters$,

— $Types = ITypes \uplus PTypes$.

Ainsi, on peut définir (ou redéfinir) les relations ci-dessous en faisant abstraction des éléments $Contigencies$, $States$ et $Values$.

— $Container : Containers \rightarrow Components$ est une fonction totale permettant de spécifier le composant contenant une interface ou un paramètre donné,

— $ContainerType : Containers \rightarrow Types$ est une fonction totale qui associe un type à chaque interface et chaque paramètre,

— $Contingency : IRequired \rightarrow \{mandatory, optional\}$ est une fonction totale qui indique la contingence de chaque interface requise,

— $Delegate : Interface \rightarrow Interface$ est une fonction partielle qui exprime les liens de délégation (formellement, $Delegate = DelegateProv \uplus DelegateReq$),

— $State : Components \rightarrow \{started, stopped\}$ est une fonction totale qui donne l'état de chaque composant instancié,

— $Value : Parameters \rightarrow \{v \mid \exists t \in PTypes.\ v \in t\}$ est une fonction totale qui donne la valeur de chaque paramètre.

On peut aussi faire abstraction de la relation $Descendant$ qui est formellement la relation inverse de la fermeture transitive [6] de $Parent$, i.e., $Parent^{+-1}$, et peut être déduite de $Parent$.

La figure 3.5 est une représentation sous forme simplifiée des différents éléments et relations permettant de spécifier une architecture d'un système à composants. Les conventions utilisées sont les mêmes que celles de la figure 3.4. Les éléments et relations déjà présents dans la figure 3.4 sont grisés s'il ne sont plus utilisées, sinon ils restent inchangés. Les ajouts par rapport à la figure 3.4 sont représentés en vert.

L'exemple 9 illustre la configuration $c = \langle Elem, Rel \rangle$ pour le composant **Location** défini précédemment dans la section 3.1.3 en utilisant les éléments et relations simplifiés définis ci-dessus. Pour une meilleure visibilité, nous utilisons la notation $x \mapsto y$ pour un élément (x, y) d'une relation fonctionnelle.

6. La notion de fermeture transitive n'est utilisée ici que pour simplifier les notations ; en effet nous utilisons par la suite la logique du premier ordre pour définir la sémantique des opérations de reconfiguration. La fermeture transitive ne pouvant être exprimée par la logique du premier ordre, nous n'utilisons pas cette notion d'un point de vue formel.

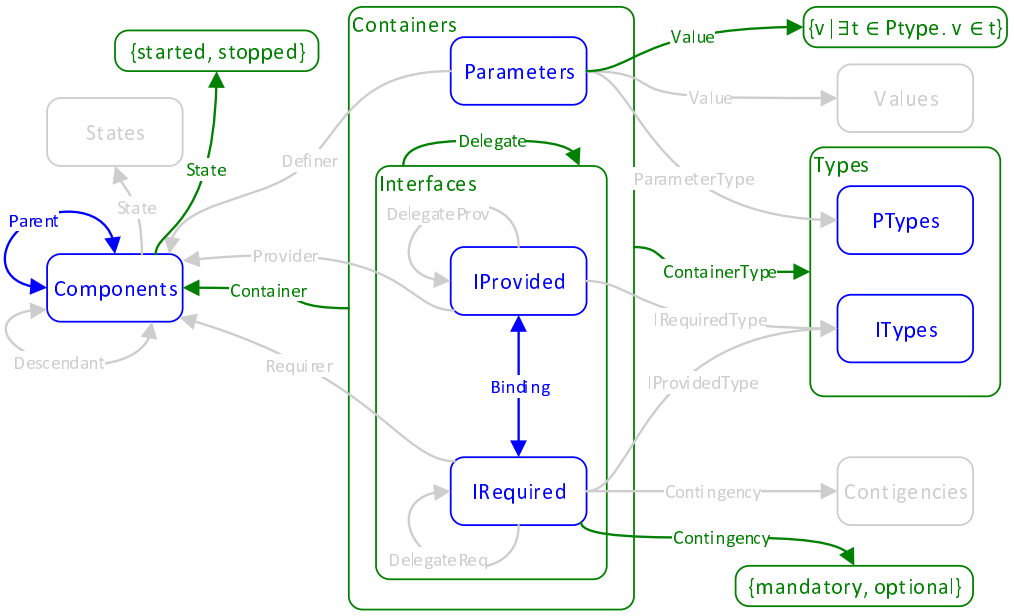


FIGURE 3.5 – Éléments et relations architecturales (représentation simplifiée)

Exemple 9 : Configuration simplifiée du composant Location

$$
\begin{array}{lcl}
Components & = & \{controller, gps, location, merger, wifi\} \\
IProvided & = & \{gpsPosition, mergePosition, position, securePosition, wifiPosition\} \\
IRequired & = & \{getGpsPosition, getMergePosition, getWifiPosition\} \\
Parameters & = & \{GpsPowerUsage, Power, Trust, WifiPowerUsage\} \\
Types & = & \{MergePosition, Position, SecurePosition, int\} \\
Container & = & \{gpsPosition \mapsto gps, mergePosition \mapsto merger, position \mapsto controller, \\
 & & securePosition \mapsto location, wifiPosition \mapsto wifi, \\
 & & getGpsPosition \mapsto merger, getMergePosition \mapsto controller, \\
 & & getWifiPosition \mapsto merger, GpsPowerUsage \mapsto merger, \\
 & & Power \mapsto controller, Trust \mapsto merger, WifiPowerUsage \mapsto merger\} \\
ContainerType & = & \{getGpsPosition \mapsto Position, getMergePosition \mapsto MergePosition, \\
 & & getWifiPosition \mapsto Position, gpsPosition \mapsto Position, \\
 & & mergePosition \mapsto MergePosition, position \mapsto SecurePosition, \\
 & & securePosition \mapsto SecurePosition, wifiPosition \mapsto Position, \\
 & & GpsPowerUsage \mapsto int, Power \mapsto int, Trust \mapsto int, \\
 & & WifiPowerUsage \mapsto int\} \\
Contingency & = & \{getGpsPosition \mapsto optional, getMergePosition \mapsto mandatory, \\
 & & getWifiPosition \mapsto optional\} \\
Parent & = & \{(controller, location), (gps, location), (merger, location), \\
 & & (wifi, location)\} \\
Binding & = & \{(gpsPosition, getGpsPosition), (mergePosition, getMergePosition), \\
 & & (wifiPosition, getWifiPosition)\} \\
Delegate & = & \{position \mapsto securePosition\} \\
State & = & \{controller \mapsto started, gps \mapsto started, location \mapsto started, \\
 & & merger \mapsto started, wifi \mapsto started\} \\
Value & = & \{GpsPowerUsage \mapsto 3, Power \mapsto 95, Trust \mapsto 0, \\
 & & WifiPowerUsage \mapsto 2\}
\end{array}
$$

3.2.2/ SÉMANTIQUE DES RECONFIGURATIONS

En utilisant les définitions précédentes, nous pouvons construire les différentes configurations d'un système à composants. Les systèmes à composants qui nous intéressent étant reconfigurables, notre modèle doit pouvoir les supporter et prendre en compte leurs reconfigurations. Nous définissons d'abord les opérations d'évolution permettant de passer d'une configuration à une autre, puis nous introduisons les propriétés de configuration qui vont nous permettre d'exprimer des contraintes architecturales sur des configurations. Enfin, nous définissons les systèmes à composants reconfigurables primitifs en termes de systèmes de transitions permettant de modéliser le comportement de ces systèmes au niveau des reconfigurations.

3.2.2.1/ RECONFIGURATIONS ET OPÉRATIONS D'ÉVOLUTION

Les reconfigurations que nous considérons consistent à passer d'une configuration à une autre. Parmi les évolutions possibles d'un système à composants, nous avons les opérations suivantes, appelées opérations primitives :

— l'instanciation ou la destruction d'un composant, respectivement notées `new` et `destroy`,

— l'ajout ou la suppression d'un composant, respectivement notés `add` et `remove`,

— le démarrage ou l'arrêt d'un composant, respectivement notés `start` et `stop`,

— l'ajout ou la suppression d'un lien entre deux interfaces de composants, respectivement notés `bind` et `unbind`,

— la modification de la valeur d'un paramètre d'un composant, notée `update`.

Notation : Les opérations primitives peuvent être écrites en utilisant les notations suivantes.

— Pour exprimer la création, la destruction, le démarrage ou l'arrêt d'un composant $comp$, nous écrivons respectivement "`new`($comp$)", "`destroy`($comp$)", "`start`($comp$)" ou "`stop`($comp$)".

— Pour l'ajout d'un composant $comp$ dans un composant $comp_p$, on note "`add`($comp_p, comp$)", tandis que pour la suppression d'un composant $comp$ ayant pour parent un composant $comp_p$, on note "`remove`($comp_p, comp$)".

— Pour lier l'interface fournie int_1 d'un composant à l'interface requise int_2 d'un autre (ou du même) composant ou pour créer une relation de délégation entre deux interfaces fournies (ou requises) où int_2 est l'interface du composant parent et int_1 est celle du sous-composant , on écrit "`bind`(int_1, int_2)". Pour supprimer une telle liaison, on écrit "`unbind`(int_1, int_2)" (ou bien "`unbind`(int_1)" ou "`unbind`(int_2)" s'il n'y a pas d'ambiguïté).

— Pour mettre à jour un paramètre $param$, si la valeur qu'on souhaite lui affecter est val, on note "`update`($param, val$)".

Remarque : Lorsqu'on utilise les notations ci-dessus dans le cadre d'une implémentation utilisant des composants instanciés via le modèle à composants Fractal ou FraSACti, il peut être difficile de distinguer les interfaces de deux composants distincts qui portent des

interfaces ayant le même nom. C'est pourquoi nous utilisons une notation alternative, présentée ci-dessous, qui permet de lever les ambiguïtés éventuelles. La partie de la notation entre crochets ("[" et "]") peut être omise si cela ne crée pas d'ambiguïté.

— Pour exprimer la création, la destruction, le démarrage ou l'arrêt d'un composant $comp$, nous écrivons respectivement "$comp$ `new`", "$comp$ `destroy`", "$comp$ `start`" ou "$comp$ `stop`".

— Pour l'ajout d'un composant $comp$ dans un composant $comp_p$, on note "$comp_p$ `add` $comp$", tandis que pour la suppression d'un composant $comp$ ayant pour parent un composant $comp_p$, on note "$comp_p$ `remove` $comp$".

— Pour lier l'interface requise (resp. fournie) int_1 d'un composant $comp_1$ à l'interface fournie (resp. requise) int_2 d'un composant $comp_2$, on écrit "[$comp_1$] `bind` int_1 [$comp_2$] int_2". Pour supprimer une telle liaison, on écrit "[$comp_1$] `unbind` int_1 [$comp_2$] [int_2]".

— pour créer ou supprimer une relation de délégation entre deux interfaces fournies (ou requises), où int_p est l'interface du composant parent $comp_p$ et int_c est celle du sous-composant $comp_c$, on note respectivement "[$comp_c$] `bind` int_c [$comp_p$] int_p" ou "[$comp_c$] `unbind` int_c".

— Pour mettre à jour un paramètre $param$ du composant $comp$, si le type de $param$ est $type$ et la valeur qu'on souhaite lui affecté est val, on note "[$comp$] `update` [$type$] $param$ val".

Les reconfigurations ne sont pas les seules opérations permettant de faire évoluer une architecture à composants. Les exécutions normales des différents composants (appelées opérations d'exécution) changent aussi l'architecture en modifiant, par exemple, les valeurs de paramètres. Cet aspect des systèmes à composants reconfigurables génère un nombre infini de configurations pour un système donné. En effet, les paramètres pouvant être des entiers, ils peuvent prendre un nombre infini de valeurs.

En considérant le modèle donné dans la section 3.2.1, une opération d'évolution fait évoluer l'architecture, avec une opération primitive ou avec une opération d'exécution, en transformant une configuration $c = \langle Elem, Rel \rangle$ en une autre configuration $c' = \langle Elem', Rel' \rangle$. Parmi les opérations d'évolution, nous nous intéressons principalement aux opérations primitives de reconfiguration. C'est pourquoi les opérations d'exécution sont toutes représentées par une opération générique notée run.

Définition 30 : Opérations d'évolution

L'ensemble des opérations d'évolution $\mathcal{OP}_{run}$ est défini par :

$$\mathcal{OP}_{run} = \mathcal{OP} \cup \{run\}$$

où

— $\mathcal{OP} = \{op_0, op_1, op_2, \ldots\}$ est un ensemble fini d'opérations primitives,

— run est une action représentant une ou plusieurs opérations d'exécution.

L'exemple 10 décrit l'ensemble des opérations d'évolution utilisables dans le cas du composant **Location** (voir figure 3.3). L'ensemble $\mathcal{OP}$ des opérations primitives contient les opérations nécessaires à l'ajout ou la suppression des composants **GPS** et **Wi-Fi** ainsi qu'à la création ou destruction des liens entre leurs interfaces et celles du composant

Merger. On notera aussi la présence d'opérations de type "`update`" qui permet d'affecter les valeurs 0 ou 100 à l'attribut $Power$.

Exemple 10 : Opérations d'évolution du composant Location

L'ensemble des opérations d'évolution $\mathcal{OP}_{run}$ du composant **Location** est défini par $\mathcal{OP}_{run} = \mathcal{OP} \cup \{run\}$ où

$$\begin{aligned}\mathcal{OP} = \{&\texttt{add}(location, gps), \texttt{add}(location, wifi), \texttt{bind}(gpsPosition, getGpsPosition),\\ &\texttt{bind}(wifiPosition, getWifiPosition), \texttt{remove}(gps), \texttt{remove}(wifi),\\ &\texttt{start}(gps), \texttt{start}(controller), \texttt{start}(location), \texttt{start}(merger), \texttt{start}(wifi),\\ &\texttt{stop}(gps), \texttt{stop}(controller), \texttt{stop}(location), \texttt{stop}(merger), \texttt{stop}(wifi),\\ &\texttt{unbind}(getGpsPosition), \texttt{unbind}(getWifiPosition), \texttt{update}(Power, 0),\\ &\texttt{update}(Power, 100)\}\end{aligned}$$

3.2.2.2/ PROPRIÉTÉS DE CONFIGURATION

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.

Définition 31 : Propriété de configuration

L'ensemble des propriétés de configuration $CP = \{cp_0, cp_1, cp_2, \ldots\}$ est défini comme un ensemble de formules de la logique du premier ordre formées sur :

- Un ensemble de constantes : $\mathcal{S}_C = \{\top, \bot\}$,
- Un ensemble infini dénombrable de variables : $\mathcal{V} = \{x, y, z, \ldots\}$,
- Un ensemble de symboles de fonctions :

$$\begin{aligned}\mathcal{S}_{\mathcal{F}} = \{&IProvidedType, IRequiredType, Provider, Requirer, Contingency,\\ &ParameterType, Definer, DelegateProv, DelegateReq, State, Value\}\end{aligned}$$

- Un ensemble de symboles de relations :
$$\mathcal{S}_{\mathcal{R}} = \{Parent, Descendant, Binding\},$$
- Un ensemble de symboles de prédicats qui est l'ensemble de base pour les formules logiques du premier ordre : $\mathcal{S}_{\mathcal{P}} = \{\in, =, \ldots\}$,
- Des connecteurs logiques $\wedge, \vee, \neg, \Rightarrow$,
- Des quantificateurs $\exists, \forall$.

Nous pouvons à présent préciser la sémantique des relations (fonctionnelles ou non) et des prédicats.

Étant donné le domaine $Elem$ (voir définition 28), la fonction d'interprétation I associant des fonctions, des relations (non fonctionnelles) et des prédicats sur $Elem$ aux symboles de fonctions, de relations et de prédicats de la syntaxe des propriétés de configuration est définie comme suit :

- Pour chaque $f \in \mathcal{S}_{\mathcal{F}}$, $I(f)$ est définie de la même façon que dans la définition 29,

— Pour chaque $r \in \mathcal{S}_{\mathcal{R}}$, $I(r)$ est définie de la même façon que dans la définition 29,

— Pour chaque $p \in \mathcal{S}_{\mathcal{P}}$, $I(p)$ est définie comme dans la logique du premier ordre [Hamilton, 1988].

L'exemple 11 montre une propriété de configuration exprimant le fait que deux composants soient connectés via leurs interfaces.

Exemple 11 : Propriété de configuration du composant Location

La propriété de configuration suivante (nommée $gpsConnected$) exprime le fait que les sous-composants **Merger** et **GPS** du composant **Location** (voir figure 3.3) sont connectés, i.e., qu'une interface fournie du composant **GPS** est liée à une interface requise du composant **Merger**.

$$\exists (int_p, int_r) \in IProvided \times IRequired.(Provider(int_p) = gps \\ \wedge\ Requirer(int_r) = merger \wedge Binding(int_p, int_r))$$

3.2.2.3/ SYSTÈME À COMPOSANTS RECONFIGURABLE PRIMITIF

Nous définissons à présent la notion de système à composants reconfigurable primitif en terme de systèmes de transitions. Les états du système de transitions représentent les configurations du système à composants et les transitions représentent les opérations d'évolution du système qui permettent de passer d'une configuration à une autre. Ce système est qualifié de primitif car l'ensemble des opérations d'évolution est composé d'opérations primitives. De plus, ce système de transitions est étiqueté avec les propriétés de configurations valides dans chaque état.

Définition 32 : Système à composants reconfigurable primitif

Un système à composants reconfigurable primitif est défini par le système de transition doublement étiqueté $SP = \langle C, C^0, \mathcal{OP}_{run}, \rightarrow, l \rangle$ où :

— $C = \{c, c_1, c_2, \ldots\}$ est un ensemble de configurations,

— $C^0 \subseteq C$ est un ensemble de configurations initiales,

— $\mathcal{OP}_{run}$ est un ensemble fini d'opérations d'évolutions,

— $\rightarrow \subseteq C \times \mathcal{OP}_{run} \times C$ est la relation de reconfiguration,

— $l : C \rightarrow CP$ est une fonction, associant à chaque reconfiguration $c \in C$ un décor $l(c)$ qui est la plus grande conjonction d'éléments cp de CP évaluée à $\top$ pour c.

En utilisant la fonction l d'un système à composants reconfigurable primitif, il est possible de savoir si une configuration satisfait une propriété de configuration.

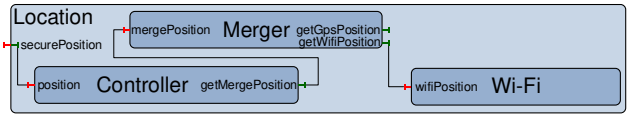
Définition 33 : Satisfaction d'une propriété de configuration par une configuration

Soit un système à composants reconfigurable primitif $SP = \langle C, C^0, \mathcal{OP}_{run}, \rightarrow, l \rangle$. Une configuration $c \in C$ satisfait une proposition de configuration $cp \in CP$, noté $c \models cp$, si

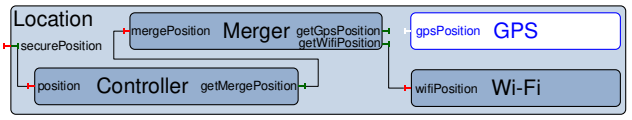
$$l(c) \Rightarrow cp$$

Dans le cas contraire, on note $c \not\models cp$.

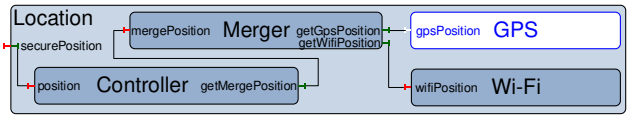
L'exemple 12 illustre un chemin d'évolution du composant **Location** qui démarrage à la configuration c_0 où l'on considère que le composant **GPS** n'est pas présent, comme c'est le cas figure 3.6(a). L'opération d'exécution permettant de passer de la configuration c_0 à la configuration c_1 étant une exécution normale du système, elle est représentée par l'opération générique run. Le passage de la configuration c_1 à c_2, représentée figure 3.6(b) s'effectue via la transition "$ope_a = \texttt{add}(gps)$" qui permet l'ajout du composant **GPS** arrêté, i.e., dans l'état $stopped$. L'opération primitive "$ope_b = \texttt{bind}(gpsPosition, getGpsPosition)$" permet de passer à la configuration suivante, c_3 où apparait la liaison du composant **GPS** au composant **Merger** ; comme l'illustre la figure 3.6(c). Enfin, nous passons de la configuration c_3 à c_4 (voir figure 3.6(d)) en démarrant le composant **GPS** au moyen de l'opération primitive "$ope_s = \texttt{start}(gps)$". Le passage de la configuration c_4 à c_5 (toute comme la transition de l'état c_0 à c_1) est représenté par l'opération générique run.



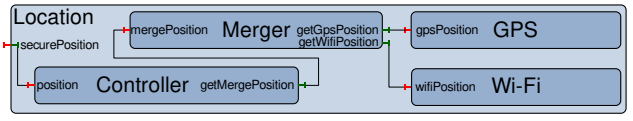
(a) Composant **GPS** absent



(b) Composant **GPS** arrêté et déconnecté



(c) Composant **GPS** arrêté et connecté



(d) Composant **GPS** démarré et connecté

FIGURE 3.6 – Évolution du composant **Location** au cours de l'ajout du sous-composant **GPS**

Exemple 12 : Chemin d'évolution du composant Location

La figure 3.7 représente un chemin d'évolution possible du composant **Location** où les opérations primitives ope_a, ope_b et ope_s sont définis ci-dessous.

- $ope_a = \texttt{add}(gps)$,
- $ope_b = \texttt{bind}(gpsPosition, getGpsPosition)$,
- $ope_s = \texttt{start}(gps)$.

Remarque : La propriété de configuration $gpsConnected$ définie dans l'exemple 11 n'est pas valide sur les configuration c_0 à c_2 correspondant aux figures 3.6(a) à 3.6(b) ; for-

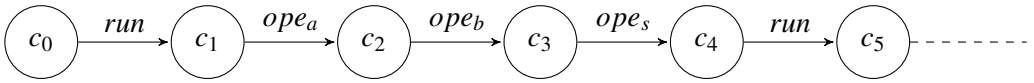


FIGURE 3.7 – Exemple d'un chemin d'évolution du composant **Location**

mellement : $c_i \not\models gpsConnected$, pour $0 \leq i \leq 2$. En revanche, les configurations c_3 à c_5 (figures 3.6(c) à 3.6(d)) satisfont la propriété de configuration $gpsConnected$; formellement : $c_i \models gpsConnected$, pour $3 \leq i \leq 5$.

Les opérations primitives peuvent être exprimées en termes de pré-conditions et post-conditions utilisant des propriétés de configuration. La table 3.1 représente les pré-conditions et post-conditions de l'opération primitive `add` qui consiste en l'ajout en un composant parent, $comp_p$, d'un composant enfant, $comp_c$. On représente les éléments et relations d'une configuration à l'état courant en utilisant les noms d'ensembles ($Elem$) et de relations (Rel), e.g., $Components$ ou $Parent$, tandis que ces mêmes éléments et relations à l'état suivant sont primés, e.g., $Components'$ ou $Parent'$. Lorsqu'un ensemble ne change pas, e.g., $Components' = Components$, il ne sera simplement pas mentionné dans la post-condition.

En guise de pré-condition, on s'assure que $comp_p$ et $comp_c$ soient bien des éléments distincts de $Components$, que le composant parent, $comp_p$, n'a pas de paramètre et que le composant parent, $comp_p$ n'est pas lui-même un descendant de $comp_c$.

La post-condition consiste, d'une part, à ajouter le couple $(comp_c, comp_p)$ à la relation $Parent$. D'autre part, on ajoute à $Descendant$ le couple $(comp_p, comp_c)$ ainsi que les ensembles *a*) $\{(c, comp_c) \mid \forall c \in Components, (c, comp_p) \in Descendant\}$, signifiant que tout composant ayant $comp_p$ comme descendant doit aussi avoir $comp_c$ comme descendant ; *b*) $\{(comp_p, c) \mid \forall c \in Components, (comp_c, c) \in Descendant\}$, afin de s'assurer que tout descendant de $comp_c$ est aussi descendant de $comp_p$ et *c*) $\{(c, c') \mid \forall c, c' \in Components, (c, comp_p) \in Descendant \wedge (comp_c, c') \in Descendant\}$, pour décrire le fait que tout composant ayant $comp_p$ comme descendant doit aussi avoir comme descendants ceux de $comp_c$.

TABLE 3.1 : Opération primitive `add` : pré/post-conditions

Opération	`add`$(comp_p, comp_c)$
Pré-condition	$comp_p, comp_c \in Components \wedge comp_c \neq comp_p$ $\wedge\ (comp_c, comp_p) \notin Descendant \wedge\ \forall p \in Parameters.(comp_p, p) \notin Definer$
Post-condition	$Parent' = Parent \cup \{comp_c, comp_p\}$ $Descendant' = Descendant \cup \{(comp_p, comp_c)\} \cup$ $\{(c, comp_c) \mid \forall c \in Components, (c, comp_p) \in Descendant\} \cup$ $\{(comp_p, c) \mid \forall c \in Components, (comp_c, c) \in Descendant\} \cup$ $\{(c, c') \mid \forall c, c' \in Components, (c, comp_p) \in Descendant$ $\wedge\ (comp_c, c') \in Descendant\}$

Les pré-conditions et post-conditions de toutes opérations primitives (`new`, `destroy`, `add`, `remove`, `start`, `stop`, `bind`, `unbind` et `update`) sont décrites dans l'annexe A.

3.3/ SYSTÈME À COMPOSANTS RECONFIGURABLE

Dans les systèmes à composants reconfigurables, les reconfigurations dynamiques sont souvent considérées comme des séquences d'opérations primitives. Dans ce cas, il est utile de représenter des opérations non-primitives. En effet, nous pouvons considérer que pour qu'un composant soit ajouté dans le système et qu'il puisse être utilisé, il faut exécuter les opérations suivantes :

— instanciation du nouveau composant,

— ajout du nouveau composant au système,

— liaison des interfaces du nouveau composant et

— démarrage du nouveau composant.

Exemple 13 : Reconfiguration non primitive du composant Location

La figure 3.8 illustre, sur l'exemple de la figure 3.7, la notion de reconfiguration non-primitive. Nous montrons que la séquence contenant les opérations *a*) $ope_a = \texttt{add}(gps)$, *b*) $ope_b = \texttt{bind}(gpsPosition, getGpsPosition)$ et *c*) $ope_s = \texttt{start}(gps)$ permettant l'ajout du composant **GPS** [a] peut être considérée comme une reconfiguration non-primitive nommée $addgps$.

a. On considère que l'instanciation de ce composant est ici implicite.

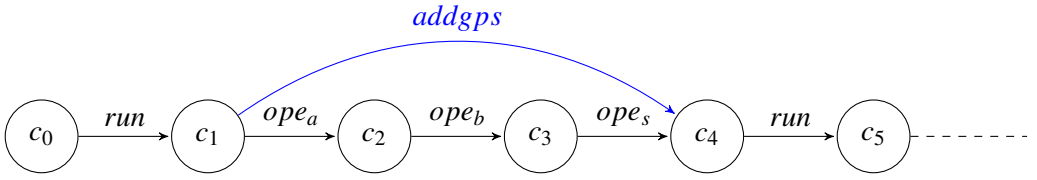


FIGURE 3.8 – Exemple de reconfiguration non primitive du composant **Location**

Définition 34 : Reconfiguration non-primitive

Soient $i, j \in \mathbb{N}$ tels que $i < j$. Soient $SP = \langle C, C^0, \mathcal{OP}_{run}, \rightarrow, l \rangle$ un système à composants reconfigurable primitif et σ un chemin de SP. Nous définissons une reconfiguration non-primitive comme un mot $w = op_i \dots op_j$ tel que

$$\sigma_i^j \in \Sigma(SP) \wedge \forall k.(i \leq k < j \Rightarrow op_{k+1} \in \mathcal{OP} \wedge \sigma(k) \overset{op_{k+1}}{\rightarrow} \sigma(k+1) \in \rightarrow)$$

Nous notons $\mathcal{R} = \{r, r_1, r_2, \dots\}$ l'ensemble fini des noms représentant les reconfigurations non-primitives d'un système à composants reconfigurable. Un élément de $\mathcal{R}$ est un nom d'une reconfiguration non-primitive associé à un mot w.

Remarque : Dans le mot w représentant une reconfiguration non-primitive, une opération d'exécution run ne peut pas intervenir car $op_{i+1} \in \mathcal{OP}$, w étant un mot sur $\mathcal{OP}$.

Exemple 14 : Reconfigurations non primitives du composant Location

Dans l'exemple du composant **Location**, nous utilisons les reconfigurations non-primitives suivantes :

- $addgps$ et $removegps$, pour l'ajout et la suppression du composant **GPS**,
- $addwifi$ et $removewifi$, pour l'ajout et la suppression du composant **GPS**,
- $chargeBattery$, pour configurer le niveau de la batterie à son maximum et
- $stopCycab$ pour arrêter le véhicule.

Dans le cadre des travaux de cette thèse, nous souhaitons plutôt manipuler des reconfigurations non-primitives. Il est donc intéressant de définir un système à composants reconfigurable comme un système de transitions prenant en compte les reconfigurations non-primitives. Pour cela nous modifions la définition 32 en changeant l'ensemble des noms d'actions et la relation de transition.

Définition 35 : Système à composants reconfigurable

Un système à composants reconfigurable est défini par le système de transition doublement étiqueté $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$ où :

- $C = \{c, c_1, c_2, \ldots\}$ est un ensemble de configurations,
- $C^0 \subseteq C$ est un ensemble de configurations initiales,
- $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ est un ensemble fini de noms de reconfigurations non-primitives et d'opérations d'exécution,
- $\mapsto \subseteq C \times \mathcal{R}_{run} \times C$ est la relation de reconfiguration non-primitive,
- $l : C \rightarrow CP$ est une fonction d'interprétation qui à chaque reconfiguration $c \in C$ associe un décor $l(c)$ qui est une proposition de configuration valide dans c.

Il existe donc un lien entre un système non-primitif et un système primitif *a*) si l'ensemble des configurations du premier est un sous-ensemble de l'ensemble des configurations du second et *b*) si chaque reconfiguration non-primitive du système non-primitif est associée à un mot représentant une séquence d'opérations primitives du système primitif. Dans ce cas nous disons que le système non-primitif est une vision du système primitif.

Définition 36 : Lien entre systèmes à composants reconfigurable

Soient $SP = \langle C_P, C_P^0, \mathcal{OP}_{run}, \rightarrow, l_P \rangle$ un système à composants reconfigurable primitif et $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$ un système à composants reconfigurable. S est une vision de SP si :

- $C \subseteq C_P$,
- $C^0 = C_P^0$,
- $\forall c, c' \in C, \forall r \in \mathcal{R}.(c \overset{r}{\mapsto} c' \Rightarrow \exists w.(w \in \mathcal{OP}^+ \wedge c \overset{w}{\rightarrow} c'))$ où $\overset{w}{\rightarrow}$ est l'extension aux mots de la relation de transition $\rightarrow$.

Exemple 15 : Lien de visibilité utilisant le composant Location

Nous illustrons, figure 3.9, le lien de visibilité entre un système à composants reconfigurable primitif (SP) et un autre non primitif (S) en représentant d'une part, un chemin de SP comme dans la figure 3.7 et d'autre part, le chemin correspondant à S, une vision de SP.

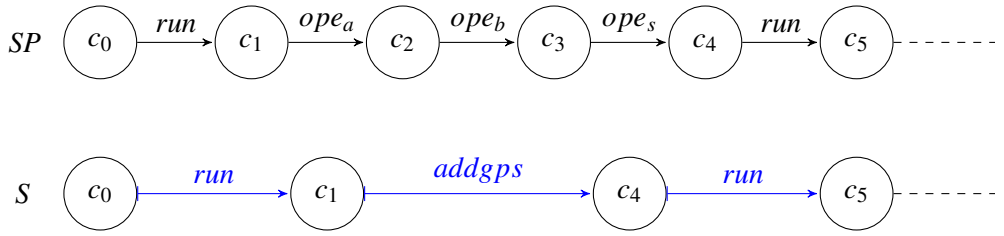


FIGURE 3.9 – Exemple de lien de visibilité utilisant le composant **Location**

3.4/ CONCLUSION

Dans ce chapitre, nous avons introduit formellement la notion de configuration en définissant les différents éléments architecturaux ainsi que les relations entre eux, puis la notion de reconfiguration permettant d'effectuer une modification architecturale sur une configuration. En utilisant les systèmes de transition, nous avons modélisé un système à composants reconfigurable primitif pour lequel les états sont des configurations et les transitions sont des reconfigurations.

Nous avons ensuite introduit la notion de reconfiguration non-primitive et nous avons défini un nouveau modèle représentant les systèmes à composants reconfigurables qui sont une vision particulière des systèmes à composants reconfigurables primitifs. Ce modèle, qui permet de donner un cadre général à la spécification des systèmes à composants reconfigurables, nous servira par la suite pour la spécification de propriétés temporelles sur ces systèmes.

Nous avons aussi présenté en détail le composant **Location** qui nous a servi à illustrer le modèle proposé.

4

RECONFIGURATIONS GARDÉES

Nous proposons dans ce chapitre de décrire des contraintes sur les éléments et relations des configurations de système à composants reconfigurables afin de pouvoir garantir que le système à composants considéré soit dans un état permettant son bon fonctionnement. En outre, en nous basant sur les pré/post-conditions des opérations primitives de reconfiguration, nous introduisons les reconfigurations gardées qui nous permettrons d'utiliser les opérations primitives en tant que "briques" pour construire des reconfigurations non-primitives impliquant des constructions, non seulement, séquentielles, mais aussi alternatives ou répétitives. Ensuite, nous montrerons que l'utilisation de reconfigurations gardées garantit la consistance des configurations d'un système à composants sous la condition que les configurations initiales soient consistantes. Enfin, nous définirons un modèle interprété en partant du modèle abstrait présenté au chapitre 3.

4.1/ CONTRAINTES DE CONSISTANCE

Parmi les propriétés de configuration, les contraintes de consistances architecturales CC, table 4.1, communes aux différents types de système à composants que nous étudions [Lanoix et al., 2011], expriment les conditions que nous souhaitons préserver pour la cohérence et le bon fonctionnement d'un système à composants.

Intuitivement,

— Un composant contient au moins une interface fournie (CC.1) ;

— un composant composite n'a pas de paramètre (CC.2) ;

Sommaire

TABLE 4.1 – Contraintes de consistance

$$\forall c \in Components.(\exists ip \in IProvided.\ Container(ip) = c) \quad \text{(CC.1)}$$

$$\forall c, c' \in Components.((c, c') \in Parent \Rightarrow \forall p \in Parameters.\ Container(p) \neq c') \ \wedge (c', c) \in Descendant \quad \text{(CC.2)}$$

$$\forall c, c', c'' \in Components.\{(c, c'), (c', c'')\} \subseteq Descendant \Rightarrow (c, c'') \in Descendant \quad \text{(CC.3)}$$

$$\forall c, c' \in Components.(c, c') \in Descendant \Rightarrow c \neq c' \wedge \begin{pmatrix} (c', c) \in Parent \vee (\exists c_1, c_2, c_3 \in Components. \\ \{(c_1, c'), (c, c_2), (c, c_3), (c_3, c')\} \subseteq Descendant \\ \wedge \{(c_1, c), (c', c_2)\} \subseteq Parent \wedge (c', c) \notin Parent) \end{pmatrix} \quad \text{(CC.4)}$$

$$\begin{matrix} \forall ip \in IProvided, \\ \forall ir \in IRequired \end{matrix} \cdot \left((ip, ir) \in Binding \Rightarrow \begin{matrix} ContainerType(ip) = ContainerType(ir) \\ \wedge\ (\ \exists\ c \in Components. \\ \{(Container(ip), c), (Container(ir), c)\} \subseteq Parent) \end{matrix} \right) \quad \text{(CC.5)}$$

$$\begin{matrix} \forall ip \in IProvided, \\ \forall ir \in IRequired, \forall id \in Interfaces \end{matrix} \cdot \left((ip, ir) \in Binding \Rightarrow \begin{matrix} Delegate(ip) \neq id \\ \wedge\ Delegate(ir) \neq id \end{matrix} \right) \quad \text{(CC.6)}$$

$$\forall i, i' \in Interfaces. \left(Delegate(i) = i' \Rightarrow \begin{matrix} \forall ip \in IProvided.\ (ip, i) \notin Binding \\ \wedge\ \forall ir \in IRequired.\ (i, ir) \notin Binding \end{matrix} \right) \quad \text{(CC.7)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in IProvided \Rightarrow i' \in IProvided) \quad \text{(CC.8)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in IRequired \Rightarrow i' \in IRequired) \quad \text{(CC.9)}$$

$$\forall i, i' \in Interfaces. \left(Delegate(i) = i' \Rightarrow \begin{matrix} ContainerType(i) = ContainerType(i') \\ \wedge\ (Container(i), Container(i')) \in Parent \end{matrix} \right) \quad \text{(CC.10)}$$

$$\forall i, i', i'' \in Interfaces. \left(\begin{matrix} (Delegate(i) = i' \wedge Delegate(i) = i'' \Rightarrow i' = i'') \\ \wedge (Delegate(i) = i'' \wedge Delegate(i') = i'' \Rightarrow i = i') \end{matrix} \right) \quad \text{(CC.11)}$$

$$\forall ir \in IRequired. \left(\begin{matrix} State(Container(ir)) = started \\ \wedge Contingency(ir) = mandatory \end{matrix} \Rightarrow \exists i \in Interfaces. \left(\begin{matrix} (i, ir) \in Binding \\ \vee Delegate(i) = ir \\ \vee Delegate(ir) = i \end{matrix} \right) \right) \quad \text{(CC.12)}$$

— les relations *Descendant* et *Parent* sont cohérentes entres elles et aucun composant ne peut descendre de lui-même (CC.3 et CC.4) [1] ;

— deux interfaces liées (par la relation fonctionnelle *binding*) doivent être de même type et appartenir a des composants ayant un parent commun (CC.5) ;

— avant de lier deux interfaces par une relation *binding*, il faut s'assurer qu'elles ne soient pas liées par une relation *delegate* à une interface d'un de leurs parents (CC.6) ;

— réciproquement, avant de lier deux interfaces par une relation *delegate*, il faut s'assurer que l'interface appartenant au sous-composant ne soit pas liée par une relation *binding* (CC.7) ;

— 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) de même type appartenant au composant parent (CC.8, CC.9, CC.10 et CC.11) et

— un composant peut être démarré (état *started*) seulement si ses interfaces requises ayant une contingence *mandatory* sont liées par une relation *binding* ou *delegate* (CC.12).

1. Voir Exemple 16 pour le détail de la contrainte de consistance (CC.4).

Exemple 16 : Détail de la contrainte de consistance (CC.4)

La contrainte de consistance CC.4) exprime le fait que pour tout couple de composants (c, c') de la relation $Descendant$, i.e., c' est un descendant de c, d'une part c est différent de c' et d'autre part

— soit c est un parent de c'

— soit il existe des composants c_1, c_2 et c_3 non nécessairement différents, tels que les couples (c_1, c'), (c, c_2), (c, c_3) et (c_3, c') appartiennent à la relation $Descendant$ et, c et c_2 sont respectivement des parents de c_1 et c'.

Cela signifie que si c' est un descendant de c on est dans l'un des cas suivants, illustrés par la figure 4.1 [a] :

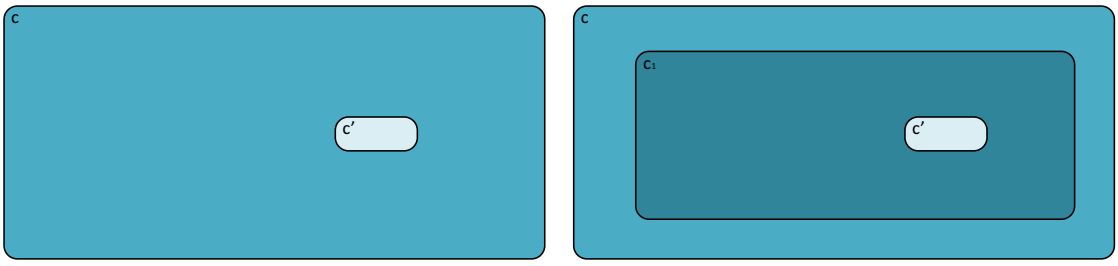
— c est un parent (une seule génération de différence) de c', comme c'est le cas figure 4.1(a) ;

— c est un grand-parent (deux générations de différence) de c' et dans ce cas $c_1 = c_2 = c_3$, voir figure 4.1(b) ;

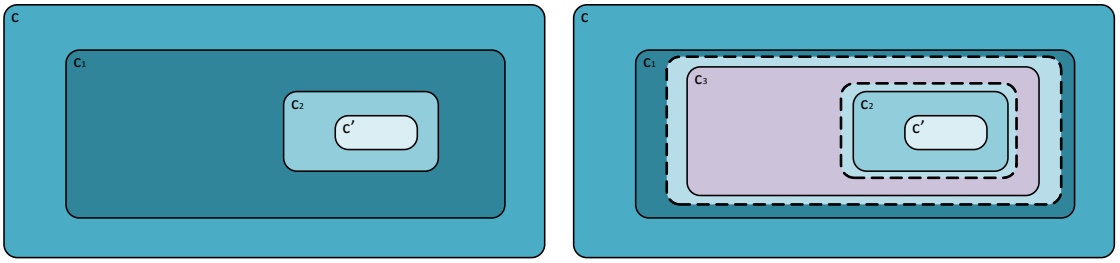
— c est un arrière grand-parent (trois générations de différence) de c' et dans ce cas $c_1 = c_3$ ou $c_2 = c_3$, voir figure 4.1(c) ; ou

— c est un ancêtre de plus de trois générations de différence avec c' et dans ce cas $c_1 \neq c_2$, $c_1 \neq c_3$ et $c_2 \neq c_3$, comme illustré figure 4.1(d).

a. Chaque forme en traits pleins représente un composant dont le nom se trouve dans le coin supérieur gauche, tandis que les formes en traits discontinus représentent une imbrication d'un nombre quelconque (pouvant être nul) de composants.



(a) c est un parent de c' (b) c est un grand-parent de c'



(c) c est un arrière grand-parent de c' (d) Plus de trois générations séparent c et c'

FIGURE 4.1 – Cas de figures possibles lorsque c' est un descendant de c

L'introduction des contraintes de consistances nous permet de définir la notion de configuration consistante.

Définition 37 : Configuration consistante

Soit $c = \langle Elem, Rel \rangle$ une configuration et CC la conjonction des contraintes de consistances. La configuration c est consistante, noté $\texttt{consistent}(c)$, si $l(c) \Rightarrow CC$. On écrit $\texttt{consistent}(C)$ si $\forall c \in C.\texttt{consistent}(c)$.

4.2/ RECONFIGURATIONS GARDÉES

En s'inspirant de la sémantique basée sur les prédicats utilisée pour les constructions de langage de programmation [Hoare, 1969] et des commandes gardées [Dijkstra, 1975], nous introduisons la notion de reconfiguration gardées.

Étant donné un système à composants reconfigurable $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$, soient $ope \in \mathcal{R}$, une opération de reconfiguration permettant d'évoluer de la configuration c à la configuration c' (noté $c \xrightarrow{ope} c'$) et R une propriété de configuration. Alors, la notation $wp(ope, R)$ décrit, comme dans [Dijkstra, 1975] la plus faible pré-condition (*weakest pre-condition*) sur la configuration courante (c dans notre cas) telle que ope puisse s'effectuer et que, le cas échéant, le résultat de ope sur c produise la configuration c' conforme à la propriété de configuration R.

Plus formellement, dans notre cas, si $l(c) \Rightarrow wp(ope, R)$ et $c \xrightarrow{ope} c'$, alors $l(c') \Rightarrow R$.

En utilisant des notations inspirées de [Dijkstra, 1975], nous proposons la grammaire de la table 4.2, ayant pour axiome <guarded reconfiguration>, pour les reconfigurations gardées.

TABLE 4.2 – Grammaire des reconfigurations gardées

<guarded reconfiguration>	::=	<guard> → <guarded list>
<guard>	::=	<boolean expression>
<guarded list>	::=	<statement> {; <statement> }
<guarded reconfiguration set>	::=	<guarded reconfiguration> {[] <guarded reconfiguration> }
<alternative construct>	::=	**if** <guarded reconfiguration set> **fi**
<repetitive construct>	::=	**do** <guarded reconfiguration set> **od**
<statement>	::=	<alternative construct> | <repetitive construct> | <ope>

Dans cette grammaire, la déclaration (<statement>) <ope> représente une opération de reconfiguration primitive, qu'on pourra aussi appeler déclaration primitive. Nous étendons l'ensemble des opérations primitives avec l'opération `skip`, qui ne produit aucun changement sur une configuration donnée. Ainsi, pour toute post-condition R, nous avons $wp(\texttt{skip}, R) = R$. De plus, comme dans [Dijkstra, 1975], la sémantique de l'opérateur ";", utilisé pour construire des séquences de déclarations, est donnée par la formule $wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$, où S_1 et S_2 sont deux déclarations.

Les ensembles de reconfigurations gardées (<guarded reconfiguration set>) sont utilisés pour définir les constructions alternatives (<alternative construct>) et répétitives (<repetitive construct>). Intuitivement, la construction alternative sélectionne pour exécution seulement les listes gardées (<guarded list>) avec une garde vraie, tandis que la construction répétitive sélectionne pour exécution les listes gardées avec une garde vraie et est répétée jusqu'à ce qu'aucune garde ne soit vraie. On notera que si un ensemble

de reconfigurations gardées contient plusieurs reconfigurations gardées, celles-ci sont séparées par l'opérateur "[]" ; l'ordre d'apparition des reconfiguration dans un ensemble de reconfigurations gardées n'a sémantiquement aucune importance.

Afin de détailler la sémantique de la construction alternative, nous représentons l'expression **if** $B_1 \rightarrow S_1 [] \ldots [] B_n \rightarrow S_n$ **fi** par IF. En utilisant BB pour représenter l'expression $(\exists i : 1 \leq i \leq n : B_i)$, on a $wp(IF, R) = BB \wedge (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R))$.

Pour la sémantique de la construction répétitive, nous représentons par DO l'expression **do** $B_1 \rightarrow S_1 [] \ldots [] B_n \rightarrow S_n$ **od**. Soit $H_0(R) = R \wedge \neg BB$ et pour tout $k > 0$, on définit $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$, ainsi, on a $wp(DO, R) = \exists k \geq 0 : H_k(R)$. Intuitivement, $H_k(R)$ est la plus faible pré-condition qui *a*) garantit la terminaison après au plus k sélections d'une liste gardée et *b*) laisse le système dans un état satisfaisant R.

4.3/ PROPAGATION DE CONSISTANCE

Soit $\mathcal{R}^G$, un ensemble fini de reconfigurations gardées instanciées en fonction du système que l'on considère de sorte que chaque reconfiguration primitive soit gardée par la conjonction de ses pré-conditions.

Proposition 1 : Propagation de consistance

Soit $S = \langle C, C^0, \mathcal{R}^G_{run}, \mapsto, l \rangle$ un système à composants reconfigurable avec $\mathcal{R}^G_{run} = \mathcal{R}^G \cup \{run\}$, on a $\texttt{consistent}(C^0) \Rightarrow \texttt{consistent}(\texttt{reach}(C^0))$.

Démonstration. Nous commençons la démonstration de cette proposition en prouvant que chaque opération primitive ope préserve les contraintes de consistance. En d'autres termes, si R_{ope} est la conjonction des post-conditions de l'opération primitive ope, on cherche à établir que $(l(c) \Rightarrow CC \wedge wp(ope, R_{ope})) \wedge c \xrightarrow{ope} c' \Rightarrow (l(c') \Rightarrow CC \wedge R_{ope})$.

Comme, par construction, ope est gardée par P_{ope}, la conjonction de ses pré-conditions, on a $P_{ope} = wp(ope, R_{ope})$. Ainsi, pour démontrer la préservation des contraintes de consistances par ope, nous devons prouver $(l(c) \Rightarrow CC \wedge P_{ope}) \wedge c \xrightarrow{ope} c' \Rightarrow (l(c') \Rightarrow CC \wedge R_{ope})$. L'opération `skip`, ne produisant aucun changement de configuration, préserve de manière évidente les contraintes de consistances. Il en est de même pour les opérations `destroy`, `stop`, `unbind` et `update` dont la sémantique (voir annexe A) n'autorise de modifications pouvant affecter les contraintes de consistance.

En revanche, la contrainte (CC.1) peut être affectée par l'opération `new`. De même, (CC.2), (CC.3) et (CC.4) peuvent être affectées par `add` ou `remove`, tandis que l'opération `start` ne peut affecter que la contrainte (CC.12). Enfin, l'opération `bind` peut avoir une influence sur les contraintes (CC.5) à (CC.11).

Nous vérifions dans l'annexe B que pour $S = \langle C, C^0, \mathcal{R}^G_{run}, \mapsto, l \rangle$, un système à composants reconfigurable, que chaque contrainte de consistance est préservée par les opérations primitives susceptibles de l'affecter, lorsque S évolue de c à c'.

A titre d'illustration, nous effectuons cette vérification pour l'opération `new`. Cette opération primitive ne peut affecter que la contrainte (CC.1). Considérons la création d'un composant $comp$ dans S au moyen de la reconfiguration primitive $\texttt{new}(comp)$, abrégé par `new` ; par hypothèse on a $(l(c) \Rightarrow CC \wedge P_{\texttt{new}}) \wedge c \xrightarrow{\texttt{new}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{new}}$.

Comme (CC.1) et $P_{\texttt{new}}$ sont satisfaites par la configuration c, on a, en utilisant les notations de la table A.1, $\forall x \in Components.(\exists ip \in IProvided.\ Container(ip) = x)$, d'une part, et $IProvided_{comp} \neq \emptyset$ d'autre part.

Par définition de l'opération primitive `new`, $Components' = Components \cup \{comp\}$ et $IProvided' = IProvided \cup IProvided_{comp}$. Ainsi, $\forall x \in Components'.(\exists ip \in IProvided'.\ Container'(ip) = x)$, ce qui signifie que (CC.1) est satisfaite par c'.

Ainsi, pour toute opération primitive ope, $(l(c) \Rightarrow CC \wedge P_{ope}) \wedge c \xrightarrow{ope} c' \Rightarrow (l(c') \Rightarrow CC \wedge R_{ope})$.

Soit $c \in \texttt{reach}(C^0)$; par définition, il existe $c_0 \in C^0$ et une suite d'opérations de $\mathcal{R}^G_{run}$ pour atteindre c. Il existe également une suite d'opérations primitives $ope_0, ope_1, \ldots, ope_{n-1}$ et un ensemble de configurations intermédiaires $C' = \{c_1, c_2, \ldots, c_{n-1}\}$ tels que $c_0 \xrightarrow{ope_0} c_1$, $c_1 \xrightarrow{ope_1} c_2$, $\ldots$, $c_{n-1} \xrightarrow{ope_{n-1}} c$, où c_i et c_{i+1} satisfont respectivement les pré-conditions et post-conditions de ope_i ($l(c_i) \Rightarrow P_{ope_i}$ et $l(c_{i+1}) \Rightarrow R_{ope_i}$), pour $0 \leq i \leq n-1$ (en identifiant c_n avec c). On notera que C' n'est pas nécessairement un sous-ensemble de C.

Si une telle suite d'opérations primitives ou C' n'existait pas, c ne serait atteignable d'aucune configuration de C^0 ($c \notin \texttt{reach}(C^0)$) ; ainsi, pour $c \in \texttt{reach}(C^0)$ une telle suite existe toujours. On notera, du fait de l'existence de cette suite, que $l(c_{i+1}) \Rightarrow R_{ope_i} \wedge P_{ope_{i+1}}$. Ainsi, du fait que chaque reconfiguration primitive ope_i ($0 \leq i < n$) préserve la consistance des configurations, pour peu que la configuration initiale $c_0 \in C^0$ soit consistante, toute configuration $c \in \texttt{reach}(C^0)$ l'est aussi. En d'autres termes, on a bien $\texttt{consistent}(C^0) \Rightarrow \texttt{consistent}(\texttt{reach}(C^0))$. □

4.4/ MODÈLE D'ARCHITECTURE INTERPRÉTÉE

Dans le modèle spécifié au chapitre 3, les reconfigurations (gardées) sont abstraites et *run* n'est pas interprété. Une sémantique formelle des systèmes à composants avec des opérations interprétées peut être obtenue en enrichissant les configurations avec l'état de la mémoire et l'effet des actions de reconfiguration (en tant qu'éléments de $\mathcal{R}_{run}$) sur la mémoire.

4.4.1/ CONFIGURATIONS ET RECONFIGURATIONS INTERPRÉTÉES

Considérons un ensemble (infini en général) $GM = \{u, \ldots\}$ d'états de mémoire partagés et un ensemble (infini en général) $LM = \{v, \ldots\}$ d'états de mémoire étant chacun local à un composant donné. Ces états de mémoire sont lus et modifiés par les reconfigurations primitives et non primitives et aussi par les actions génériques qui implémentent *run*.

Configurations interprétés. Au delà de l'interprétation des paramètres et interfaces déjà inclue dans notre modèle, l'état des composants peut être décrit plus précisément en utilisant des états locaux de mémoire. L'ensemble des états interprétés des composants est le plus petit ensemble $State_{\mathcal{I}}$ tel que, pour toute configuration $conf \in C$, si $s_1, \ldots, s_n$ sont des états non interprétés de composants $c_1, \ldots, c_n$ permettant de décrire totalement $conf$, $v_1, \ldots, v_n$ étant des éléments correspondant de LM, alors $((s_1, v_1), \ldots, (s_n, v_n))$ est dans $State_{\mathcal{I}}$. Ainsi, l'ensemble des configurations interprétées $C_{\mathcal{I}}$ est défini par $GM \times State_{\mathcal{I}}$.

Transitions interprétés. Notre assomption de base est que toute opération de reconfi-

guration primitive se termine, soit normalement avec l'effet escompté, soit en générant une erreur. De plus, nous considérons que pour toute reconfiguration primitive ope, la reconfiguration interprétée correspondante $\overline{ope}$ dispose de pré-conditions plus fortes ou équivalentes de façon que toute construction de reconfiguration gardée se comporte de façon déterminisme. Un comportement non déterministe pourra néanmoins être induit par l'intrication arbitraire des composants. Dans le cas de pré-conditions plus fortes pour les opérations interprétées que pour celles du modèle, *a*) soit on s'assure de ne pas induire dans le système interprété des comportements différents de ceux du modèle abstrait, *b*) soit on réduit le modèle en conséquence. Par exemple, si la pré-condition de l'opération interprétée `add` stipule, en plus de la pré-condition de la table 3.1, que le composant "fils" ($comp_c$) ne doit pas avoir de parent, nous considérerons un sous-ensemble de notre modèle dans lequel chaque composant ne peut avoir que zéro ou un seul parent.

Formellement, toutes les actions $ope \in \mathcal{R}_{run}$ sont interprétées comme une application $\overline{ope}$ de $GM \times LM$ dans lui-même. De plus, comme il peut exister des actions spécifiques à une interprétation donnée (pour pouvoir tester des gardes de reconfigurations gardées, par exemple), nous désignons par $\mathcal{R}_{int}$ l'ensemble contenant ces actions. Ainsi, on dit que $\mathcal{I} = (GM, LM, (\overline{ope})_{ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}})$ est une interprétation de l'ensemble sous-jacent $\mathcal{R}_{run}$ et nous désignons par $\mathcal{I}_{\mathcal{R}_{run}}$ la classe de toutes les interprétation de $\mathcal{R}_{run}$. Cette construction nous permet de définir notre modèle de reconfigurations interprété.

Définition 38 : Modèle de reconfigurations interprété.

L'interprétation de la sémantique opérationnelle des systèmes à composants est définie par le système de transitions doublement étiqueté $S_{\mathcal{I}} = \langle C_{\mathcal{I}}, C^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ où $C_{\mathcal{I}}$ est un ensemble de configurations avec les états de leur mémoire, $C^0_{\mathcal{I}}$ est l'ensemble des configurations initiales, $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}\}$, $\rightarrow_{\mathcal{I}} \subseteq C_{\mathcal{I}} \times \mathcal{R}_{run_{\mathcal{I}}} \times C_{\mathcal{I}}$ est la fonction de reconfiguration interprétée et $l_{\mathcal{I}} : C_{\mathcal{I}} \rightarrow CP$ est une fonction totale d'interprétation.

4.4.2/ INTERPRÉTATION COMPATIBLE

Afin de pouvoir établir le lien entre notre modèle de reconfiguration et un modèle interprété correspondant, nous proposons d'utiliser l'ordre partiel de la τ-simulation [Milner, 1989] en ré-étiquetant les opérations de $\mathcal{R}_{\mathcal{I}}$ par τ. Ainsi, pour $ope \in \mathcal{R} \cup \{\epsilon\}$, où ϵ désigne le mot vide, on écrit $c \overset{ope}{\Rightarrow} c'$ s'il existe $n, m \geq 0$ tels que $c \overset{\tau^n ope \tau^m}{\rightarrow} c'$.

Définition 39 : d-simulation.

Soient $S_1 = \langle C_1, C^0_1, \ldots \rangle$ et $S_2 = \langle C_2, C^0_2, \ldots \rangle$ deux modèles sur $\mathcal{R}$. Une relation binaire $\sqsubseteq_d \subseteq C_1 \times C_2$ est une d-simulation si et seulement si, pour toute opération ope de $\mathcal{R} \cup \{\epsilon\}$, $(c_1, c_2) \in \sqsubseteq_d$ implique *a*) lorsque $c_1 \overset{ope}{\Rightarrow}_1 c'_1$, il existe $c'_2 \in C_2$ tel que $c_2 \overset{ope}{\Rightarrow}_2 c'_2$ et $(c'_1, c'_2) \in \sqsubseteq_d$, et *b*) $c_1 \overset{ope}{\nRightarrow}$ implique $c_2 \overset{ope}{\nrightarrow}$.

On écrit $S_1 \sqsubseteq_d S_2$ lorsque $\forall c^0_1 \in C^0_1, \exists c^0_2 \in C^0_2 . (c^0_1, c^0_2) \in \sqsubseteq_d$.

Considérons les opérations de reconfiguration interprétée de $\mathcal{R}_{run_{\mathcal{I}}}$ et leurs homologues non interprétées de $\mathcal{R}_{run}$. Comme conséquence du ré-étiquetage des opérations de $\mathcal{R}_{int}$

par τ, nous pouvons considérer, à la notation barre horizontale[2] près, que le modèle abstrait est une τ-simulation du modèle interprété comme illustré figure 4.2.

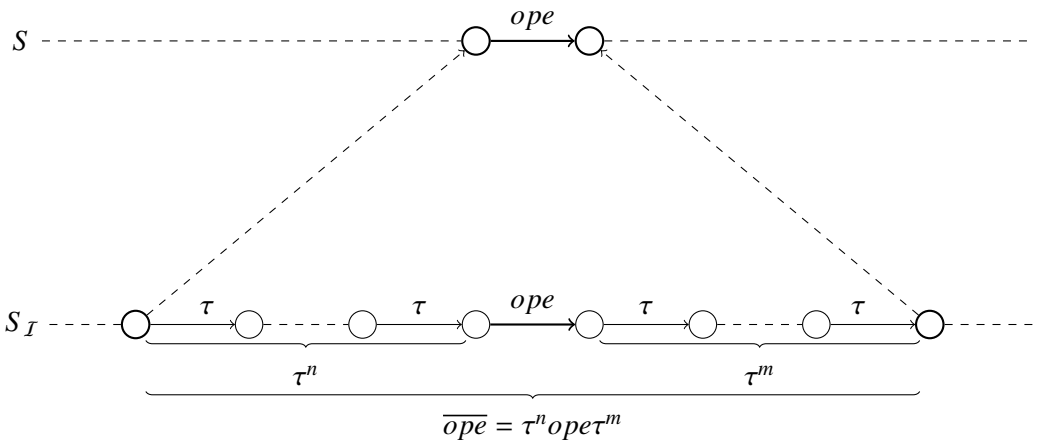


FIGURE 4.2 – τ-simulation du modèle interprété.

Théorème 1 : Compatibilité

Étant donné un système à composant S et son interprétation $S_{\mathcal{I}}$, on a $S_{\mathcal{I}} \sqsubseteq_d S$.

Démonstration. Considérons $ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}$. Comme la couverture par τ des opérations de $\mathcal{R}_{int}$ sont introduites pour évaluer les gardes de configurations gardées, elle ne forment pas de cycle infini de τ-transitions ce qui implique qu'il y a toujours une fin au cycle de τ-transitions pour laisser la place soit à la transition $\overline{ope}$, soit à la transition suivante correspondant à $ope' \in \mathcal{R}_{run} \cup \mathcal{R}_{int}$.

Par construction, toute opération de reconfiguration primitive du modèle interprété a des pré-conditions plus fortes ou équivalentes que son homologue du modèle non interprété. Ainsi, en utilisant les hypothèses sur les pré-conditions comme dans [Dijkstra, 1975], nous pouvons établir que les reconfigurations gardées composées de déclarations primitives (voir page 74) comme $G \rightarrow \overline{s}$, avec $\overline{s} \in \mathcal{R}_{run_{\mathcal{I}}} \backslash \mathcal{R}_{int}$ ont des pré-conditions plus fortes ou équivalentes que leurs déclarations primitives homologues $s \in \mathcal{R}_{run}$

En conséquence, en considérant une configuration (initiale pour la première opération de reconfiguration considéré) $c_1 \in C_{\mathcal{I}}$, il y a une configuration correspondante $c_2 \in C$ telle que si une reconfiguration gardée $G \rightarrow \overline{s}$ est appliquée à c_1, il existe une garde G', telle que $G \Rightarrow G'$ et $G' \rightarrow s$ puisse être appliquée à c_2. De plus, les configurations cibles c_1' et c_2' respectivement obtenues en appliquant $G \rightarrow \overline{s}$ et $G' \rightarrow s$ à c_1 et c_2 sont dans $\sqsubseteq_d$; car soit il existe une opération ope pouvant être appliqué (dans ce cas on reprend le même raisonnement), soit une telle opération n'existe pas (cas traité ci-dessous).

Si aucune opération ope ne peut être appliquée à $c_1 \in C_{\mathcal{I}}$ (après d'éventuelles exécutions de τ), c_1 n'est pas consistante ; en conséquence, il en est de même pour $c_2 \in C$. □

2. Notation utilisée pour les opérations interprétées comme, par exemple, $\overline{ope}$

4.4.3/ PRÉSERVATION DES PROPRIÉTÉS

Le théorème 1 peut être exploité pour la préservation de propriétés compatibles avec $\sqsubseteq_d$.

Proposition 2 : Atteignabilité.

Si une configuration c n'est pas atteignable dans S, elle n'est atteignable dans aucun $S_{\mathcal{I}}$. Inversement, si une configuration c est atteignable dans S, il existe une interprétation $\mathcal{I}$ telle que c soit atteignable dans $S_{\mathcal{I}}$.

De ce fait, des propriétés de sûreté exprimées via des propriétés de non-atteignabilités peuvent être garanties. De plus, on déduit du théorème 1 et des propositions 1 et 2 la proposition suivante.

Proposition 3 : Consistance du modèle interprété.

Soit $S_{\mathcal{I}} = \langle C_{\mathcal{I}}, C^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ un modèle interprété construit à partir du modèle abstrait $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l \rangle$. Si $S_{\mathcal{I}} \sqsubseteq_d S$, alors on a $\texttt{consistent}(C^0_{\mathcal{I}}) \Rightarrow \texttt{consistent}(\texttt{reach}(C^0_{\mathcal{I}}))$.

4.5/ CONCLUSION

Dans ce chapitre nous avons décrit des contraintes sur les éléments et relations des configurations de système à composants reconfigurables afin de pouvoir garantir que le système à composants considéré soit dans un état permettant son bon fonctionnement.

En nous basant sur les pré/post-conditions des opérations primitives de reconfiguration, nous introduisons les reconfigurations gardées qui nous permettent d'utiliser les opérations primitives en tant que "briques" pour construire des reconfigurations non-primitives impliquant des constructions, non seulement séquentielles, mais aussi alternatives ou répétitives.

Ceci nous a permis que monter que l'utilisation de reconfigurations gardées garantit la consistance des configurations d'un système à composants sous la condition que les configurations initiales soient consistantes.

Enfin, nous avons défini un modèle interprété en partant du modèle abstrait présenté au chapitre 3.

5

PROPRIÉTÉS TEMPORELLES

Un modèle de système à composants pouvant évoluer sur la base de politiques d'adaptation utilisant des schémas temporels prenant en compte des événements externes a été introduit dans [Dormoy et al., 2010]. Parallèlement, la logique FTPL introduite dans [Dormoy et al., 2012a], qui se base sur les travaux de Dwyers sur les schémas de spécification [Dwyer et al., 1999] était utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permettait pas la prise en compte d'événements externes. C'est pourquoi nous avons, dans [Kouchnarenko et al., 2014a], étendu la logique FTPL avec des événements externes afin de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité. Dans ce chapitre, nous présentons d'abord la syntaxe de la logique FTPL étendue aux événements externes, puis nous décrivons sa sémantique et la notion de satisfaction d'une propriété FTPL par un système à composants avant de conclure.

5.1/ SYNTAXE DE LA LOGIQUE FTPL

La logique FTPL introduite dans [Dormoy et al., 2012a] permet d'exprimer des propriétés sur l'architecture des configurations de systèmes à composants. Les logiques temporelles telles que LTL (voir section 2.3.3) ont un fort pouvoir d'expression mais ne permettent pas de capturer des événements. Or, les propriétés concernant les reconfigurations de systèmes à composants prennent en compte des événements internes au

Sommaire

système, comme le fait qu'une reconfiguration soit effectuée avec succès (ou non) ou tout simplement qu'une reconfiguration se soit terminée sans tenir compte du fait qu'elle ait eu lieu comme prévu ou non. C'est pourquoi la logique FTPL est basée sur les travaux de Dwyer sur les schémas de spécification [Dwyer et al., 1999] et sur ceux sur JML proposés dans [Trentelman et al., 2002].

Dans les travaux de Dwyer [Dwyer et al., 1999], les auteurs ont constaté que certaines propriétés, simples à exprimer en langage naturel, sont difficiles à décrire avec une logique temporelle. A partir de ces constatations, la notion de schémas de spécification a été proposée dans le cadre du projet Bandera [1]. Une des motivations du projet était de faciliter l'écriture de propriétés temporelles grâce à des schémas, certes moins expressifs, mais plus faciles à appréhender. Pour cela, les participants à ce projet ont montré, grâce à l'étude de nombreux exemples de spécifications, que la plupart des besoins de spécification (environ 80%) sont couverts par un nombre restreint de schémas de propriétés. Les travaux proposés dans [Trentelman et al., 2002] reprennent cette approche pour spécifier des propriétés temporelles pour les applications JavaCard.

Comme le but de nos travaux est de guider et contrôler les reconfigurations de systèmes à composants, nous nous sommes intéressés aux politiques d'adaptation qui influencent à l'exécution l'évolution de tels systèmes (comme le fait Tangram4Fractal [Chauvel et al., 2009] avec Julia [2], l'implémentation de référence de Fractal). Or, ces politiques d'adaptation n'autorisaient pas l'expression de contraintes temporelles, c'est pourquoi un modèle de système à composants permettant la mise en place de politiques d'adaptation utilisant les schémas temporels fournis par qMEDL [3][Gonnord et al., 2009] a été introduit dans [Dormoy et al., 2010].

Ainsi, la logique FTPL était utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permettait pas la prise en compte d'événements externes, contrairement à qMDEL qui était utilisé pour déclencher l'application de politiques d'adaptation. C'est pourquoi nous avons [Kouchnarenko et al., 2014a] étendu la logique FTPL avec des événements externes afin de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité.

Les contraintes sur les éléments architecturaux et les relations entre ces éléments sont spécifiées en utilisant les propriétés de configuration de la définition 31. De plus, la version de la logique FTPL que nous proposons contient des événements externes, des événements (internes) déterminés par les résultats des opérations de reconfiguration, des propriétés temporelles et des propriétés de trace incorporées dans les propriétés temporelles. Nous désignons par $Prop_{FTPL}$ l'ensemble des formules FTPL qui obéissent à la grammaire ci-dessous.

La table 5.1 donne la syntaxe du langage FTPL. Il est composé de différentes couches :

— Les propriétés de configuration cp,

— les propriétés de trace <*trp*>,

— les événements <*event*> qui sont

1. http://bandera.projects.cs.ksu.edu/
2. http://fractal.ow2.org/julia/index.html
3. qMEDL est une logique, basée sur MEDL, permettant d'exprimer des propriétés relatives à des contraintes sur des quantités de ressources.

TABLE 5.1 – Syntaxe de la logique FTPL

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

  - soit les événements internes liés aux reconfigurations (comme pour les événements "ope **normal**", "ope **exceptional**" ou "ope **terminates**" liés à l'opération de reconfiguration ope),
  - soit les événements externes *ext*,
- les listes d'événements <*events*>, et
- les propriétés temporelles <*tpp*>.

Nous décrivons chaque couche en définissant sa sémantique dans la section suivante.

5.2/ SÉMANTIQUE DE LA LOGIQUE FTPL

Pour évaluer une propriété sur un chemin d'évolution σ appartenant à l'ensemble des chemins d'évolution Σ d'un système à composants non-primitif S, nous devons décider de la satisfaction de la propriété sur ce chemin. Nous présentons ci-dessous le domaine de vérité que nous utilisons avant de détailler la sémantique de la logique FTPL pour les différentes couches spécifiées ci-dessus.

5.2.1/ DOMAINE DE VÉRITÉ

Soient S un ensemble et R une relation sur $S \times S$. R est une relation de pré-ordre si et seulement si elle est réflexive et transitive ; c'est une relation d'ordre partiel si et seulement si et seulement si elle est aussi antisymétrique. Pour une relation R d'ordre partiel, la paire (S, R) est appelé un ensemble d'ordre partiel qui est parfois noté S lorsqu'il n'y a pas d'ambiguïté sur la relation d'ordre. Un treillis est un ensemble d'ordre partiel (S, R) tel que pour chaque $x, y \in S$, il existe *a*) une unique plus grande borne inférieure, et *b*) une unique plus petite borne supérieure. Un treillis est fini si et seulement si S est fini. Tous les treillis finis ont un unique plus petit élément bien défini, souvent appelé le minimum, et un unique plus grand élément bien défini, souvent appelé le maximum.

Plus spécifiquement, nous considérons une sémantique à deux valeurs avec $\mathbb{B}_2 = \{\bot, \top\}$ où $\bot$ et $\top$ correspondent respectivement aux valeurs "faux" et "vrai". Nous considérons une relation d'ordre non-stricte de vérité $\sqsubseteq_2$ sur $\mathbb{B}_2$ qui satisfait $\bot \sqsubseteq_2 \top$. Sur $\mathbb{B}_2$, nous définissons l'opération unaire $\neg_2$ comme $\neg_2\bot = \top$ et $\neg_2\top = \bot$. Les deux opérations binaires $\sqcap_2$ et $\sqcup_2$ sont définies comme respectivement le minimum et le maximum interprétés en fonction de $\sqsubseteq_2$. Ainsi, $(\mathbb{B}_2, \sqsubseteq_2)$ est un treillis booléen car pour tout $x, y \in \mathbb{B}_2$, $x \sqcup_2 \neg_2 x = \top$ et $x \sqcap_2 \neg_2 x = \bot$.

Dans la suite de ce chapitre et de façon générale, lorsqu'il n'y a pas d'ambiguïté, on notera respectivement $\sqsubseteq$, $\neg$, $\sqcap$ et $\sqcup$ en lieu et place de $\sqsubseteq_2$, $\neg_2$, $\sqcap_2$ et $\sqcup_2$.

5.2.2/ SÉMANTIQUE

Soit $\sigma \in \Sigma$ un chemin d'évolution et une propriété FTPL φ exprimée par une formule de $Prop_{FTPL}$. La valeur de vérité de φ sur σ est définie par la fonction d'interprétation $[\_ \models \_] : \Sigma \times Prop_{FTPL} \rightarrow \mathbb{B}_2$ détaillée dans les sections suivantes.

5.2.2.1/ LES PROPRIÉTÉS DE CONFIGURATION

Définition 40 : Satisfaction d'une propriété de configuration

Soient $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$ un système à composants reconfigurable, $cp \in CP$ une propriété de configuration comme définie dans la définition 31 et $\sigma(i)$ une configuration de C. La valeur de vérité de cp sur $\sigma(i)$ est définie par la fonction d'interprétation $[\_ \models \_] : \Sigma \times CP \rightarrow \mathbb{B}_2$ comme suit :

$$[\sigma(i) \models cp] = \begin{cases} \top & \text{ssi } l(\sigma(i)) \Rightarrow cp \\ \bot & \text{sinon} \end{cases}$$

5.2.2.2/ LES ÉVÉNEMENTS

Les événements internes permettent la détection des effets d'une reconfiguration (primitive ou non). Ceci nous permet d'exprimer des propriétés en prenant en compte les résultats des différentes reconfigurations du système pouvant survenir. Etant donné une opération de reconfiguration $ope \in \mathcal{R}$, nous considérons les événements 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.

Les événements externes, comme dans [Kim et al., 2002], sont considérés comme des invocations de méthodes effectuées par des capteurs (externes) lorsqu'un changement est détecté dans leurs environnements. Pour tout événement externe ext qui peut être détecté sur un chemin d'évolution donné $\sigma \in \Sigma$, nous définissons *a*) un ensemble de gardes G tel que pour chaque méthode M_{ext} invoquée suite à la détection de ext, g_{ext} est une formule de la logique du premier ordre sur les paramètres spécifiés dans l'invocation de la méthode M_{ext} et $g_{ext} \in G$, et *b*) une fonction d'évaluation $eval_\sigma : G \times \mathbb{N} \rightarrow \mathbb{B}_2$ telle que si avant ou au $i^{\text{ème}}$ état et après le $i-1^{\text{ème}}$ état (ou, si $i = 0$, au premier état) du chemin d'évolution σ, il y a au moins une occurrence de ext telle que $g_{ext} = \top$, alors $eval_\sigma(g_{ext}, i) = \top$, sinon $eval_\sigma(g_{ext}, i) = \bot$.

Définition 41 : Satisfaction d'un événement sur une configuration

Soient σ un chemin d'évolution de $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l\rangle$, $ope \in \mathcal{R}$ une opération de reconfiguration (primitive ou non) et ext un événement externe. Étant donné un événement e décrit par <*event*> dans la syntaxe de la logique FTPL, sa satisfaction sur la $i^{\text{ème}}$ configuration de σ, notée $\sigma(i) \models e$, est définie par :

$$
\begin{array}{lcl}
[\sigma(i) \models ope \textbf{ normal}] & = & \begin{cases} \top & \text{ssi } i > 0 \wedge \sigma(i-1) \neq \sigma(i) \wedge \sigma(i-1) \xrightarrow{ope} \sigma(i) \\ \bot & \text{sinon} \end{cases} \\
[\sigma(i) \models ope \textbf{ exceptional}] & = & \begin{cases} \top & \text{ssi } i > 0 \wedge \sigma(i-1) = \sigma(i) \wedge \sigma(i-1) \xrightarrow{ope} \sigma(i) \\ \bot & \text{sinon} \end{cases} \\
[\sigma(i) \models ope \textbf{ terminates}] & = & [\sigma(i) \models ope \textbf{ normal}] \sqcup [\sigma(i) \models ope \textbf{ exceptional}] \\
[\sigma(i) \models ext] & = & eval_{\sigma}(g_{ext}, i)
\end{array}
$$

Une liste d'événements est satisfaite sur une configuration si au moins un de ses éléments est satisfait.

Définition 42 : Satisfaction d'une liste d'événements sur une configuration

Soient σ un chemin d'évolution de $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l\rangle$ et une liste d'événements $e = e_1, e_2, \ldots, e_n$, décrite par <*events*> dans la syntaxe de la logique FTPL, sa satisfaction sur la $i^{\text{ème}}$ configuration de σ, notée $\sigma(i) \models e$, est définie par :

$$
[\sigma(i) \models e] = \begin{cases} \top & \text{ssi } \exists e_0.(e_0 \in e \wedge [\sigma(i) \models e_0] = \top) \\ \bot & \text{ssi } \forall e_0.(e_0 \in e \Rightarrow [\sigma(i) \models e_0] = \bot) \end{cases}
$$

5.2.2.3/ LES PROPRIÉTÉS DE TRACE

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

Définition 43 : Satisfaction d'une propriété de trace sur un chemin d'évolution

Soient σ un chemin d'évolution de $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l\rangle$ et cp une propriété de configuration. Étant donné une propriété de trace trp décrite par <*trp*> dans la syntaxe de la logique FTPL, sa satisfaction sur le chemin σ, notée $\sigma \models trp$, est définie inductivement par :

$$
\begin{array}{lcl}
[\sigma \models \textbf{always } cp] & = & \begin{cases} \top & \text{si } \forall i.(i \geq 0 \Rightarrow [\sigma(i) \models cp] = \top) \\ \bot & \text{sinon} \end{cases} \\
[\sigma \models \textbf{eventually } cp] & = & \begin{cases} \top & \text{ssi } \exists i.(i \geq 0 \wedge [\sigma(i) \models cp] = \top) \\ \bot & \text{sinon} \end{cases} \\
[\sigma \models trp_1 \wedge trp_2] & = & [\sigma \models trp_1] \sqcap [\sigma \models trp_2] \\
[\sigma \models trp_1 \vee trp_2] & = & [\sigma \models trp_1] \sqcup [\sigma \models trp_2]
\end{array}
$$

Intuitivement,

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

— **eventually** cp est satisfaite sur un chemin d'évolution 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.

5.2.2.4/ LES PROPRIÉTÉS TEMPORELLES

Les propriétés temporelles sont basées sur les listes d'événements et les propriétés de trace. Nous rappelons que pour un chemin d'évolution $\sigma \in \Sigma$, σ_i décrit le suffixe du chemin contenant $\sigma(i), \sigma(i+1), \ldots$ et σ_i^j décrit le segment du chemin contenant $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1), \sigma(j)$.

Définition 44 : Satisfaction d'une propriété temporelle sur un chemin d'évolution

Soient σ un chemin d'évolution de $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$, e une liste d'événements, trp une propriété de trace et tpp une propriété temporelle. Étant donnée une propriété de temporelle tpp_0 décrite par <*tpp*> dans la syntaxe de la logique FTPL, sa satisfaction sur la configuration σ, notée $\sigma \models tpp_0$, est définie inductivement par :

$$[\sigma \models \textbf{after } e\ tpp] = \begin{cases} \top & \text{si } \forall i.(i \geq 0 \wedge [\sigma(i) \models e] = \top \Rightarrow [\sigma_i \models tpp] = \top) \\ \bot & \text{sinon} \end{cases}$$

$$[\sigma \models \textbf{before } e\ trp] = \begin{cases} \top & \text{si } \forall i.(i > 0 \wedge [\sigma(i) \models e] = \top \Rightarrow [\sigma_0^{i-1} \models trp] = \top) \\ \bot & \text{sinon} \end{cases}$$

$$[\sigma \models trp\ \textbf{until } e] = \begin{cases} \top & \text{si } \exists i.(i > 0 \wedge [\sigma(i) \models e] = \top \Rightarrow [\sigma_0^{i-1} \models trp] = \top) \\ \bot & \text{sinon} \end{cases}$$

Intuitivement,

— la propriété **after** $e\ tpp$ est satisfaite sur un chemin d'évolution σ si la satisfaction d'un événement de e sur une configuration de σ implique la satisfaction de la propriété temporelle tpp sur le suffixe de σ débutant à cette configuration,

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

— trp **until** e est satisfaite sur un chemin d'évolution σ s'il y a une configuration de σ, qui satisfait un événement de e et la propriété de trace trp est satisfaite sur le préfixe de σ finissant juste avant l'occurrence de cet événement.

5.3/ SATISFACTION D'UNE PROPRIÉTÉ FTPL SUR UN SYSTÈME À COMPOSANTS

La sémantique précédente définit la satisfaction d'une propriété FTPL sur un chemin d'un système à composants reconfigurable. Pour qu'une telle propriété soit satisfaite, sur un

système à composants reconfigurable entier, tous les chemins doivent satisfaire cette propriété.

Définition 45 : Satisfaction d'une propriété temporelle sur un système à composants

Soient $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$, un système à composants reconfigurable et $\Pi(S)$, l'ensemble des exécutions de S. Soit φ une propriété FTPL. S satisfait φ, noté $S \models \varphi$, si

$$\forall \pi.(\pi \in \Pi(S) \Rightarrow [\pi \models \varphi])$$

Nous pouvons illustrer la notion de satisfaction d'une propriété temporelle sur un système à composants avec le système composite **Location** présenté à la section 3.1. Celui-ci supporte le retrait ou l'ajout de ses sous-composants **GPS** et **Wi-Fi** pour peu qu'à un instant donné au moins un des deux soit présent.

Dans certains cas, néanmoins, il peut être pertinent d'enlever l'un de ces composants. Par exemple, lorsque le niveau d'énergie des batteries du véhicule est bas, le composant **Wi-Fi** peut être retiré, puis rajouté plus tard lorsque les batteries sont rechargées. De plus, lorsque le véhicule entre dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible, il est pertinent de retirer le composant **GPS** qui peut être rajouté après que le véhicule soit sorti de cette zone si le niveau des batteries le permet.

En terme de logique FTPL, ce comportement du composant **GPS** peut être décrit par la propriété $\varphi =$ **after** $removegps$ **terminates** (**before** $addgps$ **terminates eventually** $\neg powerlow$), où $addgps$ et $removegps$ sont des reconfigurations non-primitives permettant respectivement l'ajout ou le retrait du composant **GPS** dans le système composite **Location** et $powerlow$ est une propriété de configuration représentant le fait que le niveau des batteries du véhicule autonome est faible (e.g., $power < 33$ où $power$ représente le pourcentage de charge global des batteries). Ainsi φ exprime en logique FTPL le fait que "après que l'on ait essayé d'enlever le composant **GPS**, le niveau des batteries doit être ou avoir été mesuré comme non faible avant d'essayer de rajouter ce composant".

Exemple 17 : Évaluation de φ sur un chemin donné

En supposant que $e_1 = removegps$ **terminates**, $e_2 = addgps$ **terminates** et $cp = \neg powerlow$, on a $\varphi =$ **after** e_1 (**before** e_2 **eventually** cp). Considérons le chemin σ décrit dans la table 5.2 où les événements e_1 et e_2 ont respectivement lieu aux configurations $\sigma(2)$ et $\sigma(7)$ et où la propriété de configuration est seulement satisfaite aux configurations $\sigma(4)$ et $\sigma(5)$.
D'après la définition 44, sur σ, $[\sigma \models \varphi] = \top$ si $[\sigma_2 \models$ **before** e_2 **eventually** $cp] = \top$, c'est-à-dire, si $[\sigma_2^6 \models$ **eventually** $cp] = \top$, ce qui est le cas car, par exemple, $[\sigma(4) \models cp] = \top$.

TABLE 5.2 – Exemple de chemin d'évolution pour l'évaluation de φ
Soient $e_1 = removegps$ **terminates**, $e_2 = addgps$ **terminates** et $cp = \neg powerlow$.

i	0	1	2	3	4	5	6	7	8	$\cdots$
e_1	$\bot$	$\bot$	$\boxed{\top}$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\cdots$
e_2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\boxed{\top}$	$\bot$	$\cdots$
cp	$\bot$	$\bot$	$\bot$	$\bot$	$\boxed{\top}$	$\boxed{\top}$	$\bot$	$\bot$	$\bot$	$\cdots$

Considérons la propriété ψ qui est similaire à la propriété φ ci-dessus à ceci près qu'on a remplacé la clause **before** par une clause **until** utilisant le même événement et la même propriété de trace (ce qui nous donne **eventually** $\neg powerlow$ **until** $addgps$ **terminates**) ; ainsi, on a ψ = **after** $removegps$ **terminates** (**eventually** $\neg powerlow$ **until** $addgps$ **terminates**). Voyons, dans l'exemple 18 ci-dessous quelles sont les implications de cette différence entre φ et ψ.

Exemple 18 : Évaluation de φ et ψ sur un chemin donné

Toujours en posant e_1 = $removegps$ **terminates**, $e_2 = addgps$ **terminates** et $cp = \neg powerlow$, on a φ = **after** e_1 (**before** e_2 **eventually** cp) d'une part et ψ = **after** e_1 (**eventually** cp **until** e_2) d'autre part. Considérons le chemin σ' décrit dans la table 5.3 où l'événement e_1 a lieu à la configuration $\sigma'(2)$ et l'événement e_2 n'a pas lieu.
D'après la définition 44, sur σ', $[\sigma' \models \varphi] = \top$ si $[\sigma'_2 \models$ **before** e_2 **eventually** $cp] = \top$ ce qui est le cas car si l'événement e n'a pas lieu **before** e trp est toujours évaluée à "vrai" quelque soit la propriété de trace trp.
En revanche, si l'événement e n'a pas lieu trp **until** e est toujours évaluée à "faux" quelque soit la propriété de trace trp. Donc, d'après la définition 44, $[\sigma' \models \psi] = \bot$ et, a fortiori d'après la définition 45, $S \not\models \psi$.

TABLE 5.3 – Exemple de chemin d'évolution pour l'évaluation de φ et ψ
Soient e_1 = $removegps$ **terminates** et $e_2 = addgps$ **terminates**.

i	0	1	2	3	4	5	6	7	8	$\cdots$
e_1	$\bot$	$\bot$	$\boxed{\top}$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\cdots$
e_2	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\cdots$

5.4/ CONCLUSION

Dans ce chapitre nous avons présenté la logique FTPL introduite dans [Dormoy et al., 2012a] que nous avons étendue, comme dans [Kouchnarenko et al., 2014a] avec des événements externes. Ceci permet de faire le lien entre les schémas temporels prenant en compte des événements externes qui ont été introduits dans [Dormoy et al., 2010] et la logique FTPL de [Dormoy et al., 2012a], qui se base sur les travaux de Dwyers sur les schémas de spécification [Dwyer et al., 1999]. Ensuite, nous avons décrit la sémantique de cette logique et avons défini la notion de satisfaction d'une propriété FTPL. Nous avons enfin fini ce chapitre avec deux exemples permettant d'illustrer l'utilisation de propriétés temporelles dans le cadre du système composite **Location** introduit au chapitre 3.

III

VÉRIFICATION ET TEST DES RECONFIGURATIONS DYNAMIQUES

6

ÉVALUATION DE PROPRIÉTÉS TEMPORELLES À L'EXÉCUTION

Dans la plupart des cas, une propriété FTPL ne peut pas être évaluée à "vrai" ou à "faux" pendant l'exécution du système car le modèle sémantique du chapitre 5 est basé sur un système de transitions infini où les exécutions du système peuvent être infinies. C'est pourquoi nous proposons, après avoir introduit le concept général d'évaluation à l'exécution, une logique à quatre valeurs ("faux", "potentiellement faux", "potentiellement vrai" et "vrai") pour évaluer les propriétés FTPL à l'exécution sur des traces d'exécution incomplètes. En outre, pour pouvoir évaluer les propriétés FTPL à l'exécution sans avoir à maintenir un historique des évaluations, nous introduisons une méthode d'évaluation progressive des propriétés FTPL permettant dans la plupart des cas d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente.

Sommaire

6.1/ ÉVALUATION À L'EXÉCUTION DE TRACES INCOMPLÈTES

Nous nous intéressons dans cette section aux concepts liés à la vérification à l'exécution. Nous présentons dans un premier temps le concept général d'évaluation à l'exécution que nous illustrons par les notions d'enforcement et de réflexion à l'exécution avant d'introduire le domaine de vérité utilisé pour l'évaluation de propriétés FTPL à l'exécution. Dans un second temps, nous présentons la sémantique pour l'évaluation de propriétés FTPL à l'exécution pour les propriétés de configurations, les événements, les propriétés de trace et les propriétés de temporelles.

6.1.1/ VÉRIFICATION DE PROPRIÉTÉS À L'EXÉCUTION

Nous nous intéressons d'abord aux concepts liés à la vérification à l'exécution en en présentant tout d'abord le concept général que nous illustrerons par deux utilisations : l'enforcement à l'exécution et la réflexion à l'exécution. Enfin, nous détaillerons le domaine de vérité que nous utiliserons pour l'évaluation de propriétés temporelles à l'exécution.

6.1.1.1/ CONCEPT DE L'ÉVALUATION À L'EXÉCUTION

La vérification à l'exécution [Havelund et al., 2005, Bauer et al., 2010] est une technique permettant de s'assurer de manière efficace de la satisfaction d'exigences par un système lors de son exécution. Ces exigences peuvent représenter le comportement souhaité du programme ou peuvent mettre en évidence les comportements non-désirés. Elles sont généralement spécifiées grâce à un formalisme de haut niveau tel que des assertions ou une formule utilisant, par exemple, la logique temporelle.

Pour prendre en compte ces exigences, la vérification à l'exécution porte sur une exécution du système et non sur tout le système comme pour les méthodes de vérification classiques. La vérification se faisant lors de l'exécution, il est nécessaire d'identifier les événements qui vont nécessiter l'évaluation de la satisfaction des exigences. Ces événements correspondent à des moments critiques de l'exécution du programme comme, par exemple, la modification d'une variable qui peut entraîner une défaillance du logiciel. Pour cela, le système doit être observé lors de son exécution pour détecter ces événements et évaluer ensuite si l'exigence est satisfaite.

En pratique, la vérification à l'exécution se fait grâce à un moniteur ou un contrôleur. Un moniteur est généralement généré à partir d'une propriété représentant une exigence et d'un modèle du système, tandis qu'un contrôleur peut agréger les informations de plusieurs moniteurs afin d'évaluer des propriétés plus complexes. Lors de l'exécution, à chaque fois qu'un événement critique se produit, une représentation abstraite de la trace d'exécution est générée à partir de l'exécution réelle du système et est ensuite fournie à un moniteur ou un contrôleur qui a pour but de décider, pour cette trace d'exécution, de la satisfaction d'une spécification fournie par l'utilisateur représentant une exigence. De nombreuses approches ont été proposées pour vérifier à l'exécution des spécifications fournies par l'utilisateur, celles-ci allant de la vérification d'assertions simples jusqu'à la vérification de propriétés temporelles.

Dans [Havelund et al., 2005], un classement des approches utilisés dans le cadre de la vérification à l'exécution est proposé. Ce classement utilise les catégories suivantes :

— la vérification basée sur des spécifications ;
— la vérification prédictive où, plutôt que de vérifier une propriété P, on vérifie une propriété Q plus forte (i.e., $\forall x, Q(x) \Rightarrow P(x)$) ; et
— l'instrumentation, qui est basée sur la modification du système cible de façon que celui-ci puisse être vérifié via l'appel d'une méthode ou la consultation d'une variable.

La vérification à l'exécution a seulement pour but d'observer le système et de donner un verdict sur la satisfaction ou non d'une exigence. Il n'implique donc aucune modification du système ou de son exécution en cas de non respect de l'exigence. Cependant, le cadre défini par la vérification à l'exécution peut-être utilisé pour mettre en œuvre des approches différentes. Nous citons, par exemple, les travaux de [Ernst et al., 2007, Rogin et al., 2008] qui utilisent la vérification à l'exécution sans spécification de propriété à vérifier. Cette approche a pour but d'observer les exécutions du système pour créer une spécification du système sur laquelle des propriétés pourront être vérifiées ou pour détecter des invariants [Ernst et al., 2007].

Ce cadre de vérification à l'exécution peut aussi permettre de réagir lors du non-respect d'une exigence ou d'anticiper une exécution non désirée. Pour cela, plusieurs approches existent et nous nous intéressons plus particulièrement à deux d'entre elles : l'enforcement de propriétés à l'exécution et la réflexion à l'exécution. Ces deux approches ont pour but d'utiliser l'évaluation des propriétés lors de la vérification à l'exécution pour agir sur l'exécution du système.

6.1.1.2/ ENFORCEMENT ET RÉFLEXION À L'EXÉCUTION

L'approche de l'enforcement ou mise en application à l'exécution [Schneider, 2000, Falcone et al., 2008, Ligatti et al., 2009, Delaval et al., 2010] consiste à utiliser la vérification des propriétés à l'exécution pour essayer d'anticiper l'apparition d'une exécution non désirée qui pourrait rendre le système défaillant. Cette approche est en fait une extension de l'approche de vérification à l'exécution qui permet de répondre au besoin de prévenir et de corriger un comportement non désiré.

L'enforcement à l'exécution utilise un mécanisme de moniteur d'enforcement ou de contrôleur d'enforcement. Ceux-ci sont semblables aux moniteurs et contrôleurs utilisés pour la vérification à l'exécution car ils prennent toujours en entrée une propriété qui représente une exigence à respecter lors de l'exécution. Cependant, un moniteur ou un contrôleur d'enforcement possèdent aussi une mémoire d'enforcement qui permettent de sauvegarder certains événements intéressants observés sur le système. Ils disposent d'un ensemble d'opérations d'enforcement qui peuvent être mises en œuvre en fonction des événements observés et des exigences données par l'utilisateur. Ces opérations d'enforcement sont alors appliquées sur le système dans le but d'essayer d'éviter qu'un comportement non désiré se produise.

La réflexion à l'exécution [Bauer et al., 2008] est une approche utilisant aussi la vérification de propriétés lors de l'exécution du système. Cette méthode a non seulement pour but d'observer le comportement du système mais aussi de donner un diagnostic

de l'exécution. Lorsque le diagnostic est assez détaillé et non-ambigü, celui-ci peut être utilisé pour faire réagir le système en exécutant des commandes, telles que des reconfigurations dynamiques par exemple, pour rétablir le système dans un état bien défini.

Cette approche consiste à observer les différents événements du système que les différents moniteurs et contrôleurs pourront ensuite traiter pour donner un verdict sur l'exécution du système. Si un comportement non-désiré est détecté, une phase de diagnostic est lancée pour en déterminer les causes afin de déclencher des actions d'atténuation (*mitigation*) sur le système pour corriger son comportement en fonction d'exigences pré-définies. Dans le cas où il n'est pas possible de corriger le comportement du système, on peut sauvegarder les erreurs d'exécution ainsi que toutes autres informations pertinentes pour une analyse a-posteriori.

6.1.1.3/ DOMAINE DE VÉRITÉ

Dans la plupart des cas, une propriété FTPL ne peut pas être évaluée à "vrai" ou à "faux" pendant l'exécution du système car le modèle sémantique du chapitre 5 est basé sur un système de transitions infini. Ces restrictions rendent impossible la génération automatique et l'utilisation de moniteurs tels que proposés dans [Falcone et al., 2008].

Dans ce chapitre nous utilisons, comme dans [Bauer et al., 2010] un domaine de vérité à quatre valeurs que nous détaillerons ci-dessous en nous appuyant sur la notion de treillis évoquée en section 5.2.1.

Nous considérons $\mathbb{B}_4 = \{\bot, \bot^p, \top^p \top\}$ comme un ensemble pour lequel $\bot$ et $\top$ correspondent respectivement aux valeurs "faux" et "vrai", tandis que $\bot^p$ et $\top^p$ correspondent respectivement aux valeurs "potentiellement faux" et "potentiellement vrai". Ces deux dernières valeurs caractérisent l'insatisfiabilité ou la satisfiabilité potentielle d'une propriété en plus de la sémantique FTPL déjà donnée au chapitre 5. Les propriétés de cette logique sont alors évaluées sur des segments de chemins comme dans les travaux sur RV-LTL [Bauer et al., 2010]. Les valeurs "potentiellement faux" ou "potentiellement vrai" sont affectées chaque fois qu'un comportement observé ne permet pas de conclure à la violation ou à la satisfaction de la propriété considérée, en fonction de la forme de la propriété considérée.

Considérons une relation d'ordre non-stricte de vérité $\sqsubseteq_4$ sur $\mathbb{B}_4$ qui satisfait $\bot \sqsubseteq_4 \bot^p \sqsubseteq_4 \top^p \sqsubseteq_4 \top$. Sur $\mathbb{B}_4$, nous définissons l'opération unaire $\neg_4$ comme $\neg_4 \bot = \top$, $\neg_4 \top = \bot$, $\neg_4 \bot^p = \top^p$ et $\neg_4 \top^p = \bot^p$. Les deux opérations binaires $\sqcap_4$ et $\sqcup_4$ sont définies comme respectivement le minimum et le maximum interprétés en fonction de $\sqsubseteq_4$. Par exemple, en considérant les deux valeurs de vérité $\top^p$ et $\bot$, $\top^p \sqcup_4 \bot$ sera égal à $\top^p$ puisque l'interprétation des deux valeurs donne $\top^p$ comme le maximum par rapport à la relation d'ordre $\sqsubseteq_4$. A l'inverse, $\top^p \sqcap_4 \bot$ sera égal à $\bot$ puisque l'interprétation des deux valeurs donne $\bot$ comme le minimum par rapport à la relation d'ordre $\sqsubseteq_4$.

Notons que $(\mathbb{B}_4, \sqsubseteq_4)$ n'est pas un treillis booléen comme $\mathbb{B}_2$ car on a, par exemple, l'inégalité $\bot^p \sqcup_4 \neg_4 \bot^p = \bot^p \sqcup_4 \top^p \neq \top$. $(\mathbb{B}_4, \sqsubseteq_4)$ est un treillis de De Morgan fini car chaque élément $x \in \mathbb{B}_4$ a un élément dual $\neg_4 x$ tel que $\neg_4(\neg_4 x) = x$ et $x \sqsubseteq_4 y$ implique $\neg_4 y \sqsubseteq_4 \neg_4 x$.

Dans la suite de ce chapitre et de façon générale, lorsqu'il n'y a pas d'ambiguïté, on notera respectivement $\sqsubseteq$, $\neg$, $\sqcap$ et $\sqcup$ en lieu et place de $\sqsubseteq_4$, $\neg_4$, $\sqcap_4$ et $\sqcup_4$.

6.1.2/ SÉMANTIQUE POUR L'ÉVALUATION DE PROPRIÉTÉS FTPL À L'EXÉCUTION

Définissons à présent la sémantique pour évaluer une propriété FTPL sur un segment d'un chemin d'évolution σ_0^n appartenant à l'ensemble Σ des chemins d'évolution d'un système à composants reconfigurable non-primitif S.

Soit $\sigma_0^n \in \Sigma^f$ un chemin d'évolution fini. La valeur de vérité d'une propriété FTPL sur σ_0^n est définie par la fonction d'interprétation $[\![\_ \models \_]\!] : \Sigma^f \times Prop_{FTPL} \to \mathbb{B}_4$ dépendant de la forme de la propriété FTPL considéré, comme détaillé ci-dessous.

6.1.2.1/ PROPRIÉTÉS DE CONFIGURATION

L'interprétation des propriétés de configuration dépend seulement du fait que les configurations considérées appartiennent bien au chemin σ_0^n nous reprenons donc la sémantique FTPL pour l'évaluation dans $\mathbb{B}_2$.

Définition 46 : Satisfaction d'une propriété de configuration

Soient $\sigma_0^n \in \Sigma^f$ un chemin d'évolution fini et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Soit $cp \in CP$ une propriété de configuration comme définie dans la définition 31. La valeur de vérité de cp sur σ_0^n est définie comme suit :

$$[\![\sigma_0^n(i) \models cp]\!] = \begin{cases} \top & \text{si } [\sigma_0^n(i) \models cp] = \top \\ \bot & \text{sinon} \end{cases}$$

6.1.2.2/ ÉVÉNEMENTS

Il en est de même pour les événements, nous reprenons aussi la sémantique FTPL pour l'évaluation dans $\mathbb{B}_2$. Comme dans la section 5.2.2.2, nous utilisons aussi une fonction d'évaluation $eval$ pour la gestion des événements externes .

Définition 47 : Satisfaction d'un événement sur une configuration

Soient $\sigma_0^n \in \Sigma^f$ un chemin d'évolution fini et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Soient $ope \in \mathcal{R}$ une opération de reconfiguration non-primitive et ext un événement externe. La satisfaction d'un événement e sur $\sigma_0^n(i)$ est définie par :

$$[\![\sigma_0^n(i) \models ope \textbf{ normal}]\!] = \begin{cases} \top & \text{si } i > 0 \wedge \sigma_0^n(i-1) \neq \sigma_0^n(i) \\ & \qquad \wedge\, \sigma_0^n(i-1) \xrightarrow{ope} \sigma_0^n(i) \\ \bot & \text{sinon} \end{cases}$$

$$[\![\sigma_0^n(i) \models ope \textbf{ exceptional}]\!] = \begin{cases} \top & \text{si } i > 0 \wedge \sigma_0^n(i-1) = \sigma_0^n(i) \\ & \qquad \wedge\, \sigma_0^n(i-1) \xrightarrow{ope} \sigma_0^n(i) \\ \bot & \text{sinon} \end{cases}$$

$$[\![\sigma_0^n(i) \models ope \textbf{ terminates}]\!] = [\![\sigma_0^n(i) \models ope \textbf{ normal}]\!] \sqcup [\![\sigma_0^n(i) \models ope \textbf{ exceptional}]\!]$$

$$[\![\sigma_0^n(i) \models ext]\!] = eval_\sigma(g_{ext}, i)$$

Une liste d'événements est satisfaite sur une configuration si au moins un de ses éléments est satisfait sur $\sigma_0^n(i)$.

Définition 48 : Satisfaction d'une liste d'événements sur une configuration

Soient σ_0^n un chemin d'évolution et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. La satisfaction d'une liste d'événements $E = e_1, e_2, \ldots, e_n$ sur σ_0^n est définie par :

$$[\![\sigma_0^n(i) \models E]\!] = \begin{cases} \top & \text{ssi } \exists e.(e \in E \wedge [\![\sigma_0^n(i) \models e]\!] = \top) \\ \bot & \text{ssi } \forall e.(e \in E \Rightarrow [\![\sigma_0^n(i) \models e]\!] = \bot) \end{cases}$$

6.1.2.3/ PROPRIÉTÉS DE TRACE

Intuitivement l'interprétation des propriétés de trace est définie comme suit :

— La propriété de trace **always** cp n'est pas satisfaite sur σ_0^n s'il existe une configuration de σ_0^n qui ne satisfait pas cp. Dans les autres cas la propriété est évaluée à "potentiellement vrai". En effet, si l'exécution se terminait à la configuration $\sigma_0^n(n)$, la propriété serait satisfaite.

— La propriété de trace **eventually** cp est satisfaite sur σ_0^n s'il y a au moins une configuration de σ_0^n qui satisfait cp. Dans les autres cas la propriété est évaluée à "potentiellement faux". En effet, si l'exécution se terminait à la configuration $\sigma_0^n(n)$, la propriété ne serait pas satisfaite.

Nous définissons à présent formellement la satisfaction d'une propriété de trace.

Définition 49 : Satisfaction d'une propriété de trace sur un chemin d'évolution

Soient σ_0^n un chemin d'évolution et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Étant donnée une propriété de trace trp, sa satisfaction sur σ_0^n est définie inductivement par :

$$[\![\sigma_0^n \models \textbf{always } cp]\!] = \begin{cases} \bot & \text{si } \exists i.(0 \leq i \leq n \wedge [\![\sigma_0^n(i) \models cp]\!] = \bot) \\ \top^p & \text{sinon} \end{cases}$$

$$[\![\sigma_0^n \models \textbf{eventually } cp]\!] = \begin{cases} \top & \text{si } \exists i.(0 \leq i \leq n \wedge [\![\sigma_0^n(i) \models cp]\!] = \top) \\ \bot^p & \text{sinon} \end{cases}$$

$$[\![\sigma_0^n \models trp_1 \wedge trp_2]\!] = [\![\sigma_0^n \models trp_1]\!] \sqcap [\![\sigma_0^n \models trp_2]\!]$$

$$[\![\sigma_0^n \models trp_1 \vee trp_2]\!] = [\![\sigma_0^n \models trp_1]\!] \sqcup [\![\sigma_0^n \models trp_2]\!]$$

Exemple 19 : Évaluation d'une propriété de trace dans $\mathbb{B}_4$

La table 6.1 illustre l'évaluation de la propriété de trace **eventually** cp sur un chemin d'évolution σ_0^n pour une propriété de configuration cp donnée. Le paramètre i de la première ligne permet d'adresser les configurations $\sigma_0^n(0)$ à $\sigma_0^n(8)$; ainsi lorsque $i = 0$, à la configuration $\sigma_0^n(0)$, la propriété de configuration n'est pas satisfaite.
En considérant les configurations $\sigma_0^n(0)$ à $\sigma_0^n(8)$ (ligne 2), on remarque que seule la configuration $\sigma_0^n(5)$ satisfait cp. Ceci explique que pour $i \geq 5$, **eventually** cp soit évaluée à "vrai" (ligne 3). En revanche, pour $0 \leq i < 5$, **eventually** cp est évaluée à "potentiellement faux" car si l'exécution devait se finir à la configuration $\sigma_0^i(i)$, la propriété ne serait pas satisfaite.

TABLE 6.1 – Évaluation de la propriété de trace **eventually** cp dans $\mathbb{B}_4$

i	0	1	2	3	4	5	6	7	8	$\cdots$
cp	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\cdots$
eventually cp	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$

6.1.2.4/ PROPRIÉTÉS TEMPORELLES

La valeur de la propriété temporelle **after** E tpp est évaluée à "potentiellement vrai" si aucun des événements de E n'a eu lieu dans aucune des configurations du chemin σ_0^n, ou si l'occurrence d'un événement de E sur une configuration implique que la propriété temporelle tpp est évaluée à "potentiellement vrai" ou "vrai" sur le suffixe du chemin qui commence à cette configuration. La propriété temporelle **after** E tpp est évaluée à "faux" s'il existe une configuration $\sigma_0^n(i)$ de σ_0^n pour laquelle un des événements de E a eu lieu et tpp est évaluée à "faux" sur le suffixe σ_i^n. Dans les autres cas, c'est-à-dire s'il existe une configuration $\sigma_0^n(i)$ de σ_0^n pour laquelle un des événements de E a eu lieu et tpp est évaluée à "potentiellement faux" sur le suffixe σ_i^n et il n'existe pas de configuration $\sigma_0^n(j)$ pour laquelle un des événements de E a eu lieu et tpp est évaluée à "faux" sur le suffixe σ_j^n, la propriété temporelle **after** E tpp est évaluée à "potentiellement faux".

Définition 50 : Satisfaction d'une propriété after E tpp

Soient $\sigma_0^n \in \Sigma^f$ un chemin d'évolution et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Soient E une liste d'événements et tpp une propriété temporelle. La satisfaction de la propriété temporelle **after** E tpp sur σ_0^n est définie inductivement par :

$$[\![\sigma_0^n \models \textbf{after } E\ tpp]\!] = \begin{cases} \top^p & \text{si } \forall i.(0 \leq i \leq n \wedge [\![\sigma_0^n(i) \models E]\!] = \top \\ & \quad \Rightarrow [\![\sigma_i^n \models tpp]\!] \in \{\top^p, \top\}) \vee \forall i.(0 \leq i \leq n \\ & \quad \Rightarrow [\![\sigma_0^n(i) \models E]\!] = \bot) \\ \bot & \text{si } \exists i.(0 \leq i \leq n \wedge [\![\sigma_0^n(i) \models E]\!] = \top \\ & \quad \wedge [\![\sigma_i^n \models tpp]\!] = \bot) \\ \bot^p & \text{sinon} \end{cases}$$

La valeur de la propriété temporelle **before** E trp est évaluée à "potentiellement vrai" si la

propriété de trace trp est évaluée à "vrai" ou "potentiellement vrai" sur chaque préfixe du chemin se finissant à la configuration précédant immédiatement une configuration pour laquelle un événement de E a eu lieu. Elle est aussi évaluée à "potentiellement vrai" si aucun des événements de E n'a eu lieu dans aucune des configurations du chemin σ_0^n. La propriété temporelle **before** E trp est évaluée à "faux" s'il existe une configuration $\sigma_0^n(i)$ de σ_0^n pour laquelle un événement de E a lieu, et trp est évaluée à "faux" ou "potentiellement faux" sur le chemin se terminant à $\sigma_0^n(i)$ ($\sigma_0^n(i)$ exclu).

Définition 51 : Satisfaction d'une propriété before E trp

Soient $\sigma_0^n \in \Sigma^f$ un chemin d'évolution et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Soient E une liste d'événements et trp une propriété de trace. La satisfaction de la propriété temporelle **before** E trp sur σ_0^n est définie inductivement par :

$$[\![\sigma_0^n \models \textbf{before } E\ trp]\!] = \begin{cases} \top^p & \text{si } \forall i.(0 < i \leq n \wedge [\![\sigma_0^n(i) \models E]\!] = \top \\ & \quad \Rightarrow [\![\sigma_0^{i-1} \models trp]\!] \in \{\top^p, \top\}) \vee \forall i.(0 < i \leq n \\ & \quad \Rightarrow [\![\sigma_0^n(i) \models E]\!] = \bot) \vee n = 0 \\ \bot & \text{si } \exists i.(0 < i \leq n \wedge [\![\sigma_0^n(i) \models E]\!] = \top \\ & \quad \wedge [\![\sigma_0^{i-1} \models trp]\!] \in \{\bot, \bot^p\}) \end{cases}$$

La valeur de la propriété temporelle trp **until** E est évaluée à "potentiellement vrai" si la propriété de trace trp est évaluée à "vrai" ou "potentiellement vrai" sur chaque préfixe du chemin se finissant à la configuration précédant immédiatement une configuration pour laquelle un événement de E a eu lieu. La propriété temporelle trp **until** E est évaluée à "faux" si σ_0^n ne satisfait pas la propriété trp ou s'il existe une configuration $\sigma_0^n(i)$ de σ_0^n pour laquelle un événement de E a lieu et que trp est "potentiellement faux" sur le chemin se terminant par $\sigma_0^n(i)$ ($\sigma_0^n(i)$ exclu). La propriété est évaluée à "potentiellement faux" si aucun des événement de E n'a lieu dans aucune des configurations du chemin σ_0^n.

Définition 52 : Satisfaction d'une propriété trp until E

Soient $\sigma_0^n \in \Sigma^f$ un chemin d'évolution et $\sigma_0^n(i) \in C$ une configuration de σ_0^n. Soient E une liste d'événements et trp une propriété de trace. La satisfaction de la propriété temporelle trp **until** E sur σ_0^n est définie inductivement par :

$$[\![\sigma_0^n \models trp\ \textbf{until } E]\!] = \begin{cases} \top^p & \text{si } \forall i.(0 < i \leq n \wedge [\![\sigma_0^n(i) \models E]\!] = \top \\ & \quad \Rightarrow [\![\sigma_0^{i-1} \models trp]\!] \in \{\top^p, \top\}) \\ \bot & \text{si } ([\![\sigma_0^n \models trp]\!] = \bot) \vee \exists i.(0 < i \leq n \\ & \quad \wedge [\![\sigma_0^n(i) \models E]\!] = \top \wedge [\![\sigma_0^{i-1} \models trp]\!] = \bot^p) \\ \bot^p & \text{si } \forall i.(0 < i \leq n \Rightarrow [\![\sigma_0^n(i) \models E]\!] = \bot) \end{cases}$$

Exemple 20 : Évaluation d'une propriété temporelle dans $\mathbb{B}_4$

La table 6.2 illustre l'évaluation de la propriété temporelle **eventually** cp **until** e_7 sur un chemin d'évolution σ_0^n pour une propriété de configuration cp donnée et e_7 un événement qui n'a lieu qu'à la configuration $\sigma_0^n(7)$. Le paramètre i de la première ligne représente les indices des configurations $\sigma_0^n(0)$ à $\sigma_0^n(8)$.
La propriété de configuration cp (ligne 2) et la propriété de trace **eventually** cp (ligne 3) sont évaluées comme dans la table 6.1 de l'exemple 19.
La propriété temporelle **eventually** cp **until** e_7 (ligne 4) est évaluée conformément à la définition 52 :

— tant que l'événement e_7 n'a pas eu lieu ($0 \leq i \leq 6$), la propriété **eventually** cp **until** e_7 est évaluée à "potentiellement faux" ;

— comme $[\![\sigma_0^n(7) \models e_7]\!] = \top$ et $[\![\sigma_0^6 \models \textbf{eventually } cp]\!] = \top$, par définition **eventually** cp **until** e_7 est évaluée à "potentiellement vrai" pour $i \geq 7$.

TABLE 6.2 – Évaluation de la propriété de trace **eventually** cp **until** e_7 dans $\mathbb{B}_4$

i	0	1	2	3	4	5	6	7	8	$\cdots$
cp	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\cdots$
eventually cp	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$
eventually cp **until** e_7	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top^p$	$\top^p$	$\top^p$

6.2/ ÉVALUATION PROGRESSIVE DE PROPRIÉTÉS FTPL

Afin de pouvoir évaluer les propriétés FTPL à l'exécution sans avoir à maintenir un historique des évaluations, nous introduisons, comme dans [Kouchnarenko et al., 2014a], une méthode d'évaluation progressive des propriétés FTPL permettant dans la plupart des cas d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente. Nous présentons dans un premier temps les notations utilisées pour l'évaluation progressive de propriétés FTPL. Ensuite, nous définissons l'évaluation progressive des propriétés de trace, puis nous introduisons la notion de l'évaluation d'une liste d'événements sur un suffixe. Enfin nous nous définissons l'évaluation progressive des propriétés temporelles.

6.2.1/ NOTATIONS

Étant donné un chemin $\sigma \in \Sigma$, et ϕ une propriété FTPL de $Prop_{FTPL}$. Si ϕ est une liste d'événements, une propriété de trace ou une propriété temporelle (le cas d'une propriété de configuration est traité ci-dessous), nous utilisons la notation $\phi_\sigma = [\![\sigma \models \phi]\!]$ pour désigner l'évaluation de ϕ dans $\mathbb{B}_2$ sur le chemin σ. On notera $\phi_\sigma(i)$ l'évaluation de ϕ dans $\mathbb{B}_4$ sur le chemin σ effectuée à la $i^{\text{ème}}$ configuration du chemin σ, c'est-à-dire à la configuration $\sigma(i)$.

De plus, si l'évaluation d'une propriété FTPL est restreinte au suffixe d'un chemin σ_k, pour $k \geq 0$, on notera pour $\phi_{\sigma_k} = [\![\sigma_k \models \phi]\!]$ l'évaluation sur σ_k. De même, $\phi_{\sigma_k}(i)$ représentera

l'évaluation de ϕ dans $\mathbb{B}_4$ sur le suffixe σ_k effectuée à la $i^{\text{ème}}$ configuration du chemin σ, où $i \geq k$.

Dans le cas d'une propriété de configuration cp, on notera $cp(i) = [\![\sigma(i) \models cp]\!]$.

6.2.2/ PROPRIÉTÉS DE TRACES

Définition 53 : Evaluation progressive d'une propriété de trace

Soient cp une propriété de configuration, ϕ et φ deux propriétés de trace telles que $\phi = \textbf{always}\ cp$ et $\varphi = \textbf{eventually}\ cp$. Nous définissons $\phi_{\sigma_k}(i)$ et $\varphi_{\sigma_k}(i)$, les évaluations respectives, dans $\mathbb{B}_4$ sur le suffixe σ_k, de $[\![\sigma_k \models \phi]\!]$ et $[\![\sigma_k \models \varphi]\!]$ à la $i^{\text{ème}}$ configuration de σ_k, ainsi :

$$\phi_{\sigma_k}(i) = \begin{cases} \top^p \sqcap cp_\sigma(k) & \text{si } i = k \\ \phi_{\sigma_k}(i-1) \sqcap cp_\sigma(i) & \text{si } i > k \end{cases}$$

$$\varphi_{\sigma_k}(i) = \begin{cases} \bot^p \sqcup cp_\sigma(k) & \text{si } i = k \\ \varphi_{\sigma_k}(i-1) \sqcup cp_\sigma(i) & \text{si } i > k \end{cases}$$

Exemple 21 : Évaluation progressive d'une propriété de trace

Sur le suffixe commençant à la $k^{\text{ème}}$ configuration d'un chemin σ,

- à la configuration d'indice k, pour une propriété de configuration cp donnée,
  - si $cp_\sigma(k) = \bot$, la propriété de trace $\varphi = \textbf{eventually}\ cp$ serait évaluée à $\bot^p$,
  - sinon ($cp_\sigma(k) = \top$) φ serait évaluée à $\top$;
- à la $i^{\text{ème}}$ configuration du chemin σ, ϕ est évaluée au maximum, interprété selon $\sqsubseteq$, de $\varphi_{\sigma_k}(i-1)$, son évaluation à la configuration précédente, et de $cp_\sigma(i)$.

La table 6.3 illustre l'évaluation de $\phi = \textbf{always}\ \neg cp$ ainsi que celle de φ détaillée ci-dessus sur le suffixe σ_k.

TABLE 6.3 – Évaluation progressive de propriétés de trace avec $\phi = \textbf{always}\ \neg cp$ et $\varphi = \textbf{eventually}\ cp$

i	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$	$k+6$	$k+7$	$k+8$	$\cdots$
$cp_\sigma(i)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\cdots$
$\phi_{\sigma_k}(i)$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\varphi_{\sigma_k}(i)$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$

6.2.3/ LISTES D'ÉVÉNEMENTS

Définition 54 : Évaluation progressive d'une liste d'événements

Soient e une liste d'événements. Nous définissons $e_{\sigma_k}(i)$, l'évaluation dans $\mathbb{B}_4$ sur le suffixe σ_k, de $[\![\sigma_k \models e]\!]$ à la $i^{\text{ème}}$ configuration du suffixe σ_k, ainsi :

$$e_{\sigma_k}(i) = \begin{cases} [\![\sigma_k(k) \models e]\!] & \text{si } i = k \\ [\![\sigma_k(i) \models e]\!] \sqcup (\top^p \sqcap e_{\sigma_k}(i-1)) & \text{si } i > k \end{cases}$$

Intuitivement, l'expression $[\![\sigma_k(i) \models e]\!] \sqcup (\top^p \sqcap e_{\sigma_k}(i-1))$ est évaluée à $\top$ s'il y a une occurrence d'un événement de la liste e à la configuration $\sigma_k(i)$. S'il n'y a pas d'occurrence d'un événement de e à la configuration $\sigma_k(i)$ et qu'il n'y en a pas eu sur le suffixe σ_k, cette expression est évaluée à $\bot$. Si, en revanche il n'y a pas d'occurrence d'un événement de e à la configuration $\sigma_k(i)$ mais qu'au moins une telle occurrence est advenue précédemment sur le suffixe σ_k, alors cette expression est évaluée à $\top^p$.

6.2.4/ PROPRIÉTÉS TEMPORELLES

Définition 55 : Évaluation progressive d'une propriété temporelle

Soient tpp une propriété temporelle, trp une propriété de trace, e une liste d'événements, ϕ, φ et ψ des propriétés temporelles telles que $\phi = \textbf{after } e\ tpp$, $\varphi = \textbf{before } e\ trp$ et $\psi = trp\ \textbf{until } e$. Nous définissons $\phi_{\sigma_k}(i)$, $\varphi_{\sigma_k}(i)$ et $\psi_{\sigma_k}(i)$, l'évaluation dans $\mathbb{B}_4$ sur le suffixe σ_k, de $[\![\sigma_k \models \phi]\!]$, $[\![\sigma_k \models \varphi]\!]$ et $[\![\sigma_k \models \psi]\!]$ à la $i^{\text{ème}}$ configuration du suffixe σ_k, ainsi :

$$\phi_{\sigma_k}(i) = \left(\bigsqcap_{j \in \mathcal{I}_{\sigma_k^i}(e)} tpp_{\sigma_j}(i) \right) \sqcap \top^p$$

où $\mathcal{I}_{\sigma_k^i}(e) = \{j | k \leq j \leq i \wedge [\![\sigma(j) \models e]\!] = \top\}$
représente l'ensemble des indices des configurations
pour lesquelles un événement de la liste e a lieu.

$$\varphi_{\sigma_k}(i) = \begin{cases} \top^p & \text{si } e_{\sigma_k}(i) = \bot \vee i = k \\ \bot & \text{si } e_{\sigma_k}(i) = \top \wedge trp_{\sigma_k}(i-1) \in \{\bot, \bot^p\} \\ \varphi_{\sigma_k}(i-1) & \text{sinon} \end{cases}$$

$$\psi_{\sigma_k}(i) = \begin{cases} \top^p & \text{si } trp_{\sigma_k}(i) \neq \bot \wedge e_{\sigma_k}(i) = \top \\ & \qquad \wedge\ e_{\sigma_k}(i-1) = \bot \wedge trp_{\sigma_k}(i-1) \in \{\top^p, \top\} \\ \bot^p & \text{si } trp_{\sigma_k}(i) \neq \bot \wedge (e_{\sigma_k}(i) = \bot \vee i = k) \\ \bot & \text{si } trp_{\sigma_k}(i) = \bot \vee (e_{\sigma_k}(i) = \top \wedge trp_{\sigma_k}(i-1) \in \{\bot, \bot^p\}) \\ \psi_{\sigma_k}(i-1) & \text{sinon} \end{cases}$$

Par définition, l'évaluation de $\phi = \textbf{after } e\ tpp$ est

1. $\top^p$ tant qu'aucune occurrence d'un événement de e n'a lieu ou si tpp est évaluée à $\top^p$ ou $\top$ sur chacun des suffixes de σ commençant à une occurrence d'un événement de e,
2. $\bot$ si sur au moins un de ces suffixes, tpp est évaluée à $\bot$, ou

3. $\bot^p$, sinon.

Pour $\varphi = \textbf{before}\ e\ trp$, l'évaluation est

1. $\top^p$ tant qu'aucune occurrence d'un événement de e n'a eu lieu (ou au tout début su suffixe. i.e., $i = k$, même si un événement de e a lieu),
2. $\bot$ si pour chaque occurrence de e, trp est évaluée à $\bot$ ou $\bot^p$ sur le segment commençant au début du préfixe considéré et finissant à la configuration précédant l'occurrence de e considérée,
3. sinon, à la $i^{\text{ème}}$ configuration, φ, est évaluée à la même valeur qu'à la configuration précédente (ayant $i-1$ pour indice).

Intuitivement, l'évaluation de la propriété $\psi = trp\ \textbf{until}\ e$ peut être vue comme celle de $\varphi = \textbf{before}\ e\ trp$ avec les deux exceptions suivantes :

1. lorsque trp est évaluée à $\bot$, ψ est évaluée à $\bot$;
2. au debut du suffixe, tant que e n'a pas eu lieu, ψ est évaluée à $\bot^p$.

Exemple 22 : Évaluation progressive d'une propriété de temporelle

La table 6.4 a pour but de montrer le détail de l'évaluation de la propriété temporelle $\phi = \textbf{after}\ start, exit\ (\top^p\ \textbf{until}\ entry)$, qui peut s'écrire $\phi = \textbf{after}\ e\ \varphi$ lorsque $e = start, exit$ et $\varphi = \top^p\ \textbf{until}\ entry$. On voit que ϕ est toujours évaluée à $\bot^p$ sauf entre les configurations (incluses) où l'événement $entry$ a lieu jusqu'à la configuration précédant celle où l'événement $exit$ a lieu. L'événement $entry$ a lieu aux $j^{\text{ème}}$ et $l^{\text{ème}}$ configurations, tandis que l'évaluation de $e = start, exit$ est $\top$ aux configurations d'indices 0 et k. Ainsi $\phi_{\sigma_0}(i) = \varphi_{\sigma_0}(i)$ pour $i < k$ et $\phi_{\sigma_0}(i) = \varphi_{\sigma_0}(i) \sqcap \varphi_{\sigma_k}(i)$ pour $i \geq k$.

TABLE 6.4 – Détail de l'évaluation de "$\textbf{after}\ start, exit\ (\top^p\ \textbf{until}\ entry)$" avec $e = start, exit$, $\varphi = \top^p\ \textbf{until}\ entry$ et $\phi = \textbf{after}\ start, exit\ (\top^p\ \textbf{until}\ entry) = \textbf{after}\ e\ \varphi$

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$k-1$	k	$k+1$	$\cdots$	$l-1$	l	$l+1$	$\cdots$
$[\![\sigma(i) \models start]\!]$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$[\![\sigma(i) \models entry]\!]$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$
$[\![\sigma(i) \models exit]\!]$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$[\![\sigma(i) \models e]\!]$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$\mathcal{I}_{\sigma_0^i}(e)$	$\{0\}$	$\{0\}$	$\cdots$	$\{0\}$	$\{0\}$	$\{0\}$	$\cdots$	$\{0\}$	$\{0,k\}$	$\{0,k\}$	$\cdots$	$\{0,k\}$	$\{0,k\}$	$\{0,k\}$	$\cdots$
$entry_{\sigma_0}(i)$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top$	$\top^p$	$\cdots$
$\varphi_{\sigma_0}(i)$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$
$entry_{\sigma_k}(i)$	$\times$	$\times$	$\cdots$	$\times$	$\times$	$\times$	$\cdots$	$\times$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\top^p$	$\cdots$
$\varphi_{\sigma_k}(i)$	$\times$	$\times$	$\cdots$	$\times$	$\times$	$\times$	$\cdots$	$\times$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$
$\phi_{\sigma_0}(i)$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$

On notera que l'exemple 22 illustre le cas de la propriété FTPL de type **after** pour lequel l'évaluation à une configuration donnée ne se base pas uniquement sur l'évaluation effectuée à la configuration précédente. En effet, d'après la définition 55, une propriété de la forme $\textbf{after}\ e\ tpp$ est évaluée sur le suffixe σ_k en utilisant la conjonction des évaluations de tpp sur les suffixes σ_j $(j \geq k)$ commençant aux configurations auxquelles un événement de la liste e a lieu.

6.3/ CONCLUSION

Dans la plupart des cas, une propriété FTPL ne peut pas être évaluée à “vrai” ou à “faux” pendant l’exécution du système car le modèle sémantique du chapitre 5 est un système de transitions infini. C’est pourquoi nous utilisons une logique à quatre valeurs correspondant aux valeurs “faux” et “vrai”, d’une part et “potentiellement faux” et “potentiellement vrai”, d’autre part. Ces deux dernières valeurs caractérisent l’insatisfiabilité ou la satisfiabilité potentielle d’une propriété en plus de la sémantique FTPL déjà donnée chapitre 5.

Nous avons dans ce chapitre présenté la sémantique FTPL utilisant ces quatre valeurs pour l’évaluation de propriétés FTPL à l’exécution sur des traces d’exécution incomplètes. De plus, afin de pouvoir évaluer les propriétés FTPL à l’exécution sans avoir à maintenir un long historique des évaluations, nous avons introduit une méthode d’évaluation progressive des propriétés FTPL permettant dans la plupart des cas d’évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente. Nous avons également présenté le concept général d’évaluation à l’exécution que nous avons illustré par les notions d’enforcement et de réflexion à l’exécution.

7

ÉVALUATION DÉCENTRALISÉE DE PROPRIÉTÉS TEMPORELLES À L'EXÉCUTION

En s'inspirant de l'évaluation décentralisée de formules LTL présentée dans [Bauer et al., 2012], nous proposons ci-dessous une méthode pour l'évaluation décentralisée de propriétés FTPL. Nous définissons dans un premier temps les conventions utilisées pour l'évaluation décentralisée. Ensuite, nous présentons les notions de progression et d'urgence. Une fonction de progression nous permet de réécrire une formule représentant l'évaluation d'une propriété FTPL à une configuration donnée de façon qu'elle soit toujours pertinente à la configuration suivante. Dans le contexte de l'évaluation décentralisée, nous décomposons les formules en sous-formules auxquelles nous attribuons un niveau d'urgence pour déterminer l'ordre dans lequel les évaluer. Enfin, nous posons formellement le problème de l'évaluation décentralisée et proposons un algorithme pour le résoudre ainsi que des résultats de correction et de terminaison.

7.1/ CONVENTIONS UTILISÉES POUR L'ÉVALUATION DÉCENTRALISÉE

Nous considérons AE, l'ensemble des événements atomiques composé des propositions atomiques de l'ensemble CP (voir definition 31) et des événements basiques de la logique FTPL (qui sont des listes d'événements contenant un seul élément). Dans la suite de ce

Sommaire

chapitre, un événement θ est un élément de $\Theta = 2^{AE}$.

Soit S, un système à composants reconfigurable dont le nombre de composants est n. Supposons que chaque composant C_i de S dispose d'un moniteur local M_i qui lui est attaché ; nous avons donc l'ensemble des moniteurs $\mathcal{M} = \{M_0, \ldots, M_{n-1}\}$. Nous introduisons la fonction de projection $\Pi_i : 2^{AE} \to 2^{AE}$ pour restreindre les événements de Θ à la vue locale du seul moniteur M_i et nous notons $AE_i = \Pi_i(AE)$ en supposant $\forall i, j \leq n.i \neq j \Rightarrow AE_i \cap AE_j = \emptyset$. Intuitivement, cela signifie que chaque proposition atomique de CP ou chaque événement FTPL basique n'est connu que d'un seul moniteur. Pour les relations (comme $Delegate$ ou $Parent$) mettant en œuvre plusieurs composants on considère que seul le moniteur du composant parent est au courant de cette relation. En ce qui concerne la relation $Binding$, seul le moniteur du composant ayant l'interface requise connait l'existence de la connexion de type "$Binding$".

Soit $ev : C \to \Theta$ une fonction permettant d'associer des événements de Θ à des configurations. Étant donné une configuration $\sigma(j)$ d'un chemin σ avec $j \geq 0$, l'événement correspondant est $\theta(j) = ev(\sigma(j))$. Ainsi, le comportement individuel d'un composant C_i peut être défini comme une séquence (finie ou infinie) d'événements $\theta_i = \theta_i(0) \cdot \theta_i(1) \cdots \theta_i(j) \cdots$ telle que $\forall j \geq 0.\theta_i(j) = \Pi_i(ev(\sigma(j)))$. Une telle séquence est aussi appelée trace. Nous appelons Θ^* l'ensemble des traces finies sur Θ et Θ^ω l'ensemble de telles traces infinies. L'ensemble de toutes les traces est $\Theta^\infty = \Theta^* \cup \Theta^\omega$.

Intuitivement, pour une configuration donnée $\sigma(j)$ d'un chemin σ avec $j \geq 0$, $\theta_i(j)$ représente toutes les propriétés de configuration et les événements FTPL basiques que le moniteur M_i peut connaitre à la configuration $\sigma(j)$.

Pour l'évaluation décentralisée de propriétés FTPL, plutôt que $\mathbb{B}_4$, nous considérons, comme dans [Kouchnarenko et al., 2014b], l'ensemble $\mathbb{B}_5 = \{\bot, \bot^p, \top^p, \top, \#\}$, où $\bot$ et $\top$ désignent respectivement les valeurs "faux" et "vrai", $\bot^p$ et $\top^p$ pour "potentiellement faux" et "potentiellement vrai", et # pour la valeur "indéterminé". Avec $\mathbb{B}_5$, nous considérons la relation d'ordre non-stricte $\sqsubseteq_5$ qui satisfait $\bot \sqsubseteq_5 \bot^p \sqsubseteq_5 \top^p \sqsubseteq_5 \top \sqsubseteq_5 \#$. Nous définissons deux opérations symétriques binaires $\sqcap_5$ et $\sqcup_5$ comme, respectivement le minimum et le maximum interprétés vis-à-vis de $\sqsubseteq_5$. Ainsi, $(\mathbb{B}_5, \sqsubseteq_5)$ est un treillis fini, mais ce n'est ni un treillis de De Morgan, ni, a fortiori, un treillis booléen.

Enfin, comme les formules FTPL sont exprimées dans $\mathbb{B}_4$, on a $\forall \varphi \in Prop_{FTPL}.\varphi \sqcap_5 \# = \varphi$. De plus, pour toute formule $\varphi \in Prop_{FTPL}$, on désigne par $\hat{\varphi}$ l'évaluation de φ dans $\mathbb{B}_5$. Dans la suite de ce chapitre et de façon générale, lorsqu'il n'y a pas d'ambiguïté, on notera respectivement $\sqsubseteq$, $\neg$, $\sqcap$ et $\sqcup$ en lieu et place de $\sqsubseteq_5$, $\neg_5$, $\sqcap_5$ et $\sqcup_5$.

7.2/ PROGRESSION ET URGENCE D'EXPRESSIONS FTPL

Dans le contexte de l'évaluation décentralisée, chaque moniteur n'est pas forcément au courant des informations nécessaires à l'évaluation d'une propriété FTPL donnée et peut donc ne pas être capable de l'évaluer. Une telle propriété est représentée par une formule FTPL écrite en des termes faisant référence à la configuration (souvent la configuration courante) pour laquelle on souhaite évaluer cette propriété. Néanmoins, après la transition à la configuration suivante la formule considérée peut ne plus être valide. C'est pourquoi nous définissons une fonction de progression afin de pouvoir réécrire une formule FTPL donnée de façon pertinente pour sa possible évaluation à la configuration

suivante. Ensuite, nous introduisons la notion de forme de progression normalisée afin de pouvoir décomposer les formules en sous-formules auxquelles nous attribuons un niveau d'urgence pour déterminer l'ordre dans lequel les évaluer.

7.2.1/ PROGRESSION D'EXPRESSIONS FTPL

Intuitivement, étant donnés une formule FTPL et un ensemble d'événements atomiques, la fonction de progression donne *a*) soit la valeur (dans $\mathbb{B}_4$) à laquelle la propriété est évaluée, *b*) soit la formule réécrite de façon pertinente pour la configuration suivante.

Commençons par définir la fonction de progression pour les événements atomiques de AE et les événements FTPL.

Définition 56 : Fonction de progression pour les événements

Soit $\varepsilon, \varepsilon_1, \varepsilon_2 \in AE$, $e = e_1, e_2 \dots e_m$, une liste d'événements FTPL de AE et $\theta(i)$ un événement de Θ. La fonction de progression $P : Prop_{FTPL} \times \Theta \to Prop_{FTPL}$ est définie inductivement par :

$$
\begin{array}{llll}
P(\varepsilon, \theta(i)) = \top \text{ si } \varepsilon \in \theta(i), \bot \text{ sinon} & ; & P(\bot, \theta(i)) = \bot \\
P(\varepsilon_1 \vee \varepsilon_2, \theta(i)) = P(\varepsilon_1, \theta(i)) \vee P(\varepsilon_2, \theta(i)) & ; & P(\bot^p, \theta(i)) = \bot^p \\
P(\neg\varepsilon, \theta(i)) = \neg P(\varepsilon, \theta(i)) & ; & P(\top^p, \theta(i)) = \top^p \\
P(e, \theta(i)) = \bigvee_{1 \le j \le m} P(e_j, \theta(i)) & ; & P(\top, \theta(i)) = \top
\end{array}
$$

Avant d'aller plus loin, il faut trouver un moyen de faire référence à l'évaluation d'une propriété FTPL à une configuration précédente. C'est pourquoi, nous introduisons l'opérateur $\overline{\mathbf{X}}$ qui précède une propriété FTPL pour désigner son évaluation à la configuration précédant la configuration courante, c'est-à-dire, $P(\overline{\mathbf{X}}\xi, \theta(i)) = P(\xi, \theta(i-1))$. On écrit $\overline{\mathbf{X}}^m \xi$ pour exprimer $\overbrace{\overline{\mathbf{X}}\overline{\mathbf{X}}\dots\overline{\mathbf{X}}}^{m}\xi$. Ainsi, lorsque $m = 0$, $\overline{\mathbf{X}}^m \xi = \xi$.

Définition 57 : Progression des propriété FTPL de trace

Soit cp une propriété de configuration, ϕ et φ des propriétés de trace telles que $\phi =$ **always** cp, $\varphi =$ **eventually** cp et ψ_1, ψ_2 deux propriétés de trace quelconques. La fonction de progression P pour les propriétés de trace est définie par :

$$
P(\phi_{\sigma_k}, \theta(i)) = \begin{cases} P(cp, \theta(i)) \sqcap \top^p & \text{pour } i = k \\ P(cp, \theta(i)) \sqcap P(\overline{\mathbf{X}}\phi_{\sigma_k}, \theta(i)) & \text{pour } i > k \end{cases}
$$

$$
P(\varphi_{\sigma_k}, \theta(i)) = \begin{cases} P(cp, \theta(i)) \sqcup \bot^p & \text{pour } i = k \\ P(cp, \theta(i)) \sqcup P(\overline{\mathbf{X}}\varphi_{\sigma_k}, \theta(i)) & \text{pour } i > k \end{cases}
$$

$$
P(\psi_1 \wedge \psi_2, \theta(i)) = P(\psi_1, \theta(i)) \sqcap P(\psi_2, \theta(i))
$$

$$
P(\psi_1 \vee \psi_2, \theta(i)) = P(\psi_1, \theta(i)) \sqcup P(\psi_2, \theta(i))
$$

Définition 58 : Progression d'une liste d'événements FTPL

Soit e une liste d'événements FTPL, la fonction de progression P pour les listes d'événements est définie par :

$$P(e_{\sigma_k}, \theta(i)) = \begin{cases} P(e, \theta(i)) & \text{pour } i = k \\ P(e, \theta(i)) \sqcup (\top^p \sqcap P(\overline{\mathbf{X}} e_{\sigma_k}, \theta(i))) & \text{pour } i > k \end{cases}$$

On notera que les définition 57 et 58 sont respectivement basées sur les sémantiques des définitions 53 (et aussi de la définition 49, pour ce qui concerne les deux dernières formules) et 54.

En nous basant sur la définition 55, nous pouvons définir la fonction de progression pour les propriétés temporelles du type **after**, **before** et **until**.

Définition 59 : Progression de la propriété temporelle after

Soit e une liste d'événements FTPL, tpp une propriété de temporelle et α une propriété temporelle de la forme $\alpha = \textbf{after}\ e\ tpp$. La fonction de progression P pour la propriété de type **after** est définie par :

$$P(\alpha_{\sigma_k}, \theta(i)) = F_{\mathcal{A}}(F_{\mathcal{I}}(\mathcal{I}_{\sigma_k}(e), \theta(i)), P(e, \theta(i)), P(tpp_{\sigma_i}, \theta(i))) \text{ pour } i \geq k$$

où $F_{\mathcal{I}}$ et $F_{\mathcal{A}}$, basées sur la sémantique FTPL progressive, sont définies comme suit :

$$F_{\mathcal{I}}(\mathcal{I}_{\sigma_k}(e), \theta(i)) = \begin{cases} F_{\mathcal{I}}(\mathcal{I}_{\sigma_k}(e), \theta(i-1)) \cup \{j | j = i \wedge P(e, \theta(j)) = \top\} & \text{pour } i \geq k \\ \varnothing & \text{sinon} \end{cases}$$

$$F_{\mathcal{A}}(F_{\mathcal{I}}(\mathcal{I}_{\sigma_k}(e), \theta(i)), P(e, \theta(i)), P(tpp_{\sigma_i}, \theta(i))) = \top^p \sqcap \bigsqcap_{j \in F_{\mathcal{I}}(\mathcal{I}_{\sigma_k}(e), \theta(i))} P(tpp_{\sigma_j}, \theta(i)))$$

On notera que $F_{\mathcal{I}}$ permet la progression de l'ensemble $\mathcal{I}_{\sigma_k}(e)$, qui contient les indices de tous les états pour lesquels e a eu lieu. Ainsi, $F_{\mathcal{A}}$ peut utiliser $F_{\mathcal{I}}$ en plus des fonctions de progression de e et tpp pour faire progresser la propriété $\alpha = \textbf{after}\ e\ tpp$.

Nous définissons ci-dessous la fonction de progression pour les propriétés temporelles du type **before** :

Définition 60 : Progression de la propriété temporelle before

Soit e une liste d'événements FTPL, trp une propriété de trace et β une propriété temporelle de la forme $\beta = \textbf{before}\ e\ trp$. La fonction de progression P pour la propriété de type **before** est définie par :

$$P(\beta_{\sigma_k}, \theta(i)) = \begin{cases} \top^p & \text{pour } i = k \\ F_{\mathcal{B}}(P(e_{\sigma_k}, \theta(i)), P(\overline{\mathbf{X}} trp_{\sigma_k}, \theta(i)), P(\overline{\mathbf{X}} \beta_{\sigma_k}, \theta(i))) & \text{pour } i > k \end{cases}$$

où $F_{\mathcal{B}}$, basée sur la sémantique FTPL progressive, est définie comme suit :

$$F_{\mathcal{B}}(\varepsilon, trp, tpp) = \begin{cases} \top^p & \text{si } \varepsilon = \bot \\ \bot & \text{si } \varepsilon = \top \wedge trp \in \{\bot, \bot^p\} \\ tpp & \text{sinon} \end{cases}$$

Rappelons que, d'après la définition 55, l'évaluation d'une propriété temporelle de la

forme $\beta = \textbf{before}\ e\ trp$ est basée sur l'évaluation à l'état courant de la liste d'événements e, mais aussi sur l'évaluation à l'état précédent de la propriété temporelle β et de la propriété de trace trp. C'est pourquoi $F_{\mathcal{B}}$ utilise la fonction de progression de e (considérée à l'état courant) ainsi que celles des propriétés trp et β, elle-même, considérées à l'état précédent au moyen de l'opérateur $\overline{\mathbf{X}}$.

Passons à présent à la définition de la fonction de progression pour les propriétés temporelles du type **until** :

Définition 61 : Progression de la propriété temporelle until

Soit e une liste d'événements FTPL, trp une propriété de trace et γ une propriété temporelle de la forme $\gamma = trp\ \textbf{until}\ e$. La fonction de progression P pour la propriété de type **until** est définie par :

$$P(\gamma_{\sigma_k}, \theta(i)) = \begin{cases} P(trp_{\sigma_k}, \theta(i)) \sqcap \bot^p & \text{pour } i = k \\ F_{\mathcal{U}}(F_{\mathcal{P}}(e_{\sigma_k}, \theta(i)), F_{\mathcal{P}}(trp_{\sigma_k}, \theta(i)), P(\overline{\mathbf{X}}\gamma_{\sigma_k}, \theta(i))) & \text{pour } i > k \end{cases}$$

où $F_{\mathcal{P}}$ et $F_{\mathcal{U}}$, basées sur la sémantique FTPL progressive, sont définies comme suit :

$$F_{\mathcal{P}}(\xi, \theta(i)) = (P(\xi, \theta(i)), P(\xi, \theta(i-1)))$$

$$F_{\mathcal{U}}((\varepsilon, \varepsilon'), (trp, trp'), tpp) = \begin{cases} \top^p & \text{si } trp \neq \bot \wedge \varepsilon = \top \wedge \varepsilon' = \bot \wedge trp' \in \{\top^p, \top\} \\ \bot^p & \text{si } trp \neq \bot \wedge \varepsilon = \bot \\ \bot & \text{si } trp = \bot \vee (\varepsilon = \top \wedge trp' \in \{\bot, \bot^p\}) \\ tpp & \text{sinon} \end{cases}$$

La définition 55 décrit l'évaluation d'une propriété temporelle de la forme $\gamma = trp\ \textbf{until}\ e$ en se basant sur l'évaluation à l'état précédent de la propriété temporelle γ elle-même, mais aussi sur l'évaluation aux états précédent et courant de la liste d'événements e et de la propriété de trace trp. C'est pourquoi nous utilisons $F_{\mathcal{P}}$ pour faire progresser une propriété simultanément depuis son état courant et son état précédent. Ainsi $F_{\mathcal{U}}$ utilise $F_{\mathcal{P}}$ (appliquée à la liste d'événements e et à la propriété de trace trp) et la fonction de progression de γ, elle-même, considérée à l'état précédent au moyen de l'opérateur $\overline{\mathbf{X}}$.

L'exemple ci-dessous illustre l'utilisation de la fonction de progression pour les propriétés temporelles du type **before** :

Exemple 23 : Utilisation de la fonction de progression

Soit $\varphi = \textbf{before}\ e\ trp$ où e est une liste d'événements FTPL et trp une propriété de trace. Pour évaluer φ à la configuration d'indice $i > 0$ sur le suffixe σ_0, en supposant que $P(e_{\sigma_0}, \theta(i)) = e_{\sigma_0}(i) = \top$ et $P(trp_{\sigma_0}, \theta(i-1)) = trp_{\sigma_0}(i-1) = \bot^p$. En utilisant la définition 60, on a :

$$\begin{aligned} P(\varphi_{\sigma_0}, \theta(i)) &= F_{\mathcal{B}}(P(e_{\sigma_0}, \theta(i)), P(\overline{\mathbf{X}}trp_{\sigma_0}, \theta(i)), P(\overline{\mathbf{X}}\varphi_{\sigma_0}, \theta(i))) \\ &= F_{\mathcal{B}}(P(e_{\sigma_0}, \theta(i)), P(trp_{\sigma_0}, \theta(i-1)), P(\varphi_{\sigma_0}, \theta(i-1))) \\ &= F_{\mathcal{B}}(\top, \bot^p, P(\varphi_{\sigma_0}, \theta(i-1))) \\ &= \bot \end{aligned}$$

7.2.2/ FORME DE PROGRESSION NORMALISÉE ET URGENCE

Afin de pouvoir effectuer l'évaluation d'une formule FTPL de façon décentralisée, nous définissons ci-dessous la Forme de Progression Normalisée (NPF pour Normalised Progression Form) pour définir le point jusqu'au quel une formule doit être développée au moyen de la fonction de progression.

Définition 62 : NPF

Soit φ une propriété FTPL et θ un événement. une formule $P(\varphi, \theta)$ est en NPF si l'opérateur $\overline{\mathbf{X}}$ ne précède que des événements atomiques.

Théorème 2 : Existence de formule en NPF

Soit φ une propriété FTPL et θ un événement. Toute formule $P(\varphi, \theta)$ peut être réécrite en une formule équivalente (du point de vue sémantique) en NPF.

Démonstration. La preuve, par induction sur les indices des événements (i.e., sur les traces) est immédiate en utilisant les définitions 56 à 61. □

Exemple 24 : Développement en NPF

Soient $\varphi = \textbf{before}\ e\ trp$, $e = a, b$ et $trp = \textbf{always}\ cp$, où a et b sont des événements FTPL basiques et $cp \in CP$. La formule développée résultante (à la dernière ligne du présent exemple) est en NPF, comme prévu par le théorème 2, car l'opérateur $\overline{\mathbf{X}}$ ne précède que des événements atomiques.

$$
\begin{aligned}
P(\varphi_{\sigma_0}, \theta(0)) &= \top^p \\
P(\varphi_{\sigma_0}, \theta(1)) &= F_{\mathcal{B}}(P(e_{\sigma_0}, \theta(1)), P(\overline{\mathbf{X}} trp_{\sigma_0}, \theta(1)), P(\overline{\mathbf{X}} \varphi_{\sigma_0}, \theta(1))) \\
&= F_{\mathcal{B}}(P(e, \theta(1)) \sqcup (\top^p \sqcap P(\overline{\mathbf{X}} e_{\sigma_0}, \theta(1))), P(trp_{\sigma_0}, \theta(0)), P(\varphi_{\sigma_0}, \theta(0))) \\
&= F_{\mathcal{B}}(P(a, \theta(1)) \sqcup P(b, \theta(1)) \sqcup (\top^p \sqcap P(e_{\sigma_0}, \theta(0))), P(cp, \theta(0)) \sqcap \top^p, \top^p) \\
&= F_{\mathcal{B}}(P(a, \theta(1)) \sqcup P(b, \theta(1)) \sqcup (\top^p \sqcap P(e, \theta(0))), P(cp, \theta(0)) \sqcap \top^p, \top^p) \\
&= F_{\mathcal{B}}(P(a, \theta(1)) \sqcup P(b, \theta(1)) \sqcup (\top^p \sqcap (P(a, \theta(0)) \sqcup P(b, \theta(0)))), P(cp, \theta(0)) \sqcap \top^p, \top^p) \\
&= F_{\mathcal{B}}(P(a, \theta(1)) \sqcup P(b, \theta(1)) \sqcup (\top^p \sqcap (P(\overline{\mathbf{X}} a, \theta(1)) \sqcup P(\overline{\mathbf{X}} b, \theta(1)))), P(\overline{\mathbf{X}} cp, \theta(1)) \sqcap \top^p, \top^p)
\end{aligned}
$$

Nous avons vu, section 7.1, que chaque composant C_i de S dispose d'un moniteur local M_i qui lui est attaché. Comme dans [Bauer et al., 2012], pour l'évaluation de formules LTL, le moniteur M_j du composant C_j accepte en entrée un événement $\theta(i)$ et une formule FTPL, dans notre cas. L'application de la définition 56 peut engendrer un résultat incorrect dans un contexte décentralisé. Par exemple, si $\varepsilon \notin \theta(i)$ est vrai localement pour le moniteur M_j cela peut être dû au fait que $\varepsilon \notin AE_j$. La règle de progression décentralisée doit donc être adaptée en prenant en compte un ensemble local d'événements atomiques. Ainsi, la règle de progression pour les événements atomiques précédés par l'opérateur $\overline{\mathbf{X}}$ est donnée ci-dessous.

$$
P(\overline{\mathbf{X}}^m \varsigma, \theta(i), AE_j) = \begin{cases} \top & \text{si } \varsigma = \varsigma' \text{ pour } \varsigma' \in AE_j \cap \Pi_j(\theta(i-m)), \\ \bot & \text{si } \varsigma = \varsigma' \text{ pour } \varsigma' \in AE_j \setminus \Pi_j(\theta(i-m)), \\ \overline{\mathbf{X}}^{m+1} \varsigma & \text{sinon.} \end{cases}
$$

Nous complétons cette spécification de la fonction de progression avec le symbole spécial $\# \notin AE$ pour qui la progression est definie par $\forall j.P(\#, \theta, AE_j) = \#$.

Finalement, lors de l'évaluation (décentralisée) d'une formule FTPL, la notion d'urgence permet d'exprimer quelles sont les priorités avec lesquelles ses sous-formules doivent être évaluées. Intuitivement, l'urgence d'une formule en NPF est 0 si cette formule ne contient aucun opérateur $\overline{\mathbf{X}}$ ou la valeur du plus grand exposant de cet opérateur. Comme nous considérons une formule NPF, chaque sous-formule ς suivant l'opérateur $\overline{\mathbf{X}}$ est atomique (i.e., $\exists j.\varsigma \in AE_j$) et ne peut être évaluée que par un et un seul moniteur M_j. Une définition plus formelle est donnée ci-dessous.

Définition 63 : Urgence

Soit φ une formule FTPL et $\Upsilon : Prop_{FTPL} \to \mathbb{N}^{\geq 0}$ une fonction définie inductivement qui assigne un degré d'urgence à une formule FTPL comme suit :

$$\begin{array}{lll} \Upsilon(\varphi) = \text{match } \varphi \text{ with } & \varphi_1 \sqcup \varphi_2 | \varphi_1 \sqcap \varphi_2 | F_{\mathcal{A}}(\_, \varphi_1, \varphi_2) & \to max(\Upsilon(\varphi_1), \Upsilon(\varphi_2)) \\ & | \ F_{\mathcal{B}}(\varphi_1, \varphi_2, \varphi_3) & \to max\{\Upsilon(\varphi_i) | 1 \leq i \leq 3\} \\ & | \ F_{\mathcal{U}}((\varphi_1, \varphi_2), (\varphi_3, \varphi_4), \varphi_5) & \to max\{\Upsilon(\varphi_i) | 1 \leq i \leq 5\} \\ & | \ \overline{\mathbf{X}}\varphi & \to 1 + \Upsilon(\varphi) \\ & | \ \_ & \to 0 \end{array}$$

où $F_{\mathcal{A}}$, $F_{\mathcal{B}}$ et $F_{\mathcal{U}}$ décrivent respectivement les propriétés temporelles de type **after**, **before** et **until** comme dans les définitions 59 à 61.

Une formule φ est dite plus urgente qu'une formule ψ si et seulement si $\Upsilon(\varphi) > \Upsilon(\psi)$. Une formule φ telle que $\Upsilon(\varphi) = 0$ est dite non urgente. De plus, on définit l'ensemble des sous-formules urgentes de φ par $sus(\varphi)$ (pour Set of Urgent Sub-formulae), par $sus : Prop_{FTPL} \to 2^{Prop_{FTPL}}$ définie inductivement comme suit :

$$\begin{array}{lll} sus(\varphi) = \text{match } \varphi \text{ with } & \varphi_1 \sqcup \varphi_2 | \varphi_1 \sqcap \varphi_2 | F_{\mathcal{A}}(\_, \varphi_1, \varphi_2) & \to sus(\varphi_1) \cup sus(\varphi_2) \\ & | \ F_{\mathcal{B}}(\varphi_1, \varphi_2, \varphi_3) & \to \cup_{1 \leq i \leq 3} sus(\varphi_i) \\ & | \ F_{\mathcal{U}}((\varphi_1, \varphi_2), (\varphi_3, \varphi_4), \varphi_5) & \to \cup_{1 \leq i \leq 5} sus(\varphi_i) \\ & | \ \neg\varphi & \to sus(\varphi) \\ & | \ \overline{\mathbf{X}}\varphi & \to \{\overline{\mathbf{X}}\varphi\} \\ & | \ \_ & \to \varnothing \end{array}$$

7.3/ PROBLÈME DE L'ÉVALUATION DÉCENTRALISÉE

Soit $\hat{\varphi}_{\sigma_k}(s)$ l'évaluation de φ à la configuration d'indice s du suffixe σ_k. Du fait du contexte décentralisé, l'évaluation de $\varphi_{\sigma_k}(s)$ par un moniteur M_i peut être différée, en utilisant la fonction de progression, à une configuration $\sigma(t)$, avec $t > s$. Dans ce cas, on note ${}_i\varphi^s_{\sigma_k}(t)$ la formule décentralisée de l'évaluation de φ sur le suffixe σ_k commencée à la configuration $\sigma(s)$ réécrite en la fonction de progression pour qu'elle corresponde au contexte de la configuration $\sigma(t)$.

Ceci étant établi, nous considérons le problème de décision suivant.

Évaluation décentralisée de schéma temporel sur un chemin (***TPDEP*** pour Temporal Pattern Decentralised Evaluation on a Path)
Entrée : Une propriété temporelle FTPL φ, un suffixe σ_k avec $k \geq 0$, une configuration $\sigma(s)$ avec $s \geq k$ et un nombre $n = |\mathcal{M}|$ de moniteurs.
Sortie : $i, j < n$ et ${}_i\hat{\varphi}^s_{\sigma_k}(s+j) \in \mathbb{B}_4$, la valeur établie par le moniteur M_i à la configuration $\sigma(s+j)$ correspondant à l'évaluation de φ à la configuration $\sigma(s)$.

Afin de résoudre le problème de l'évaluation décentralisée, nous présentons, dans la suite de cette section, un algorithme pour l'évaluation décentralisée de schémas temporels sur un chemin. Dans un second temps, nous énoncerons des résultats de correction et de terminaison.

7.3.1/ ALGORITHME D'ÉVALUATION DÉCENTRALISÉE

Nous considérons, comme cas de base du problème TPDEP, la situation où seule l'opération d'évolution *run* a lieu après l'entrée du problème. Ainsi, jusqu'à ce qu'un résultat soit retourné en sortie, les communications entre moniteurs sont couvertes par les opérations *run*.

L'idée d'une évaluation décentralisée est la suivante. Comme dans [Bauer et al., 2012], à la configuration $\sigma(t)$, si ${}_i\varphi^s(t)$ ne peut être évaluée dans $\mathbb{B}_4$, le moniteur M_i fait progresser sa formule courante de ${}_i\varphi^s(t)$ vers ${}_i\varphi^s(t+1) = P({}_i\varphi^s(t), \theta(t), AE_i)$ et la transmet à un autre moniteur capable d'évaluer la sous-formule la plus urgente de ${}_i\varphi^s(t+1)$, puis M_i réinitialise sa nouvelle formule avec la valeur "indéterminé" : ${}_i\varphi^s(t+1) = \#$. Lorsque M_i reçoit une ou plusieurs formules d'autres moniteurs, chacune est "ajoutée" à la formule courante en utilisant l'opérateur $\sqcap$.

Contrairement à [Bauer et al., 2012], où le monitoring décentralisé de propriétés LTL détermine une valeur finale dans $\mathbb{B}_2$, notre méthode d'évaluation décentralisée tient compte du fait que la valeur d'une propriété FTPL dans $\mathbb{B}_4$ peut varier à différentes configurations et dépend d'autres facteurs comme le suffixe considéré ou l'occurrence d'événements externes. C'est pourquoi un résultat dans $\mathbb{B}_4$ obtenu par un moniteur est diffusé aux autres moniteurs, permettant ainsi à chaque moniteur de maintenir un historique complet sur un nombre limité de configurations passées. Cet historique peut être utilisé pour la résolution du problème TPDEP aux configurations suivantes.

Pour répondre au problème TPDEP, nous proposons l'algorithme `LDMon` présenté figure 7.1. Il prend en entrée l'indice i du moniteur courant, l'ensemble de ses événements atomiques AE_i, l'indice s de la configuration courante, la formule d'une propriété FTPL φ à évaluer et l'indice k correspondant au début du suffixe sur lequel φ doit être évaluée. Une variable entière t indique l'indice de la configuration courante au fur et à mesure qu'elle évolue. L'algorithme diffuse à tous les moniteurs, aussitôt qu'il est déterminé le résultat de l'évaluation de φ dans $\mathbb{B}_4$. Cette méthode peu sophistiquée de transmission du résultat aux autres moniteurs a été choisie car nous préférons nous concentrer sur la faisabilité de la décentralisation de l'évaluation de propriétés FTPL et nous considérons que la transmission du résultat est en dehors du cadre de nos travaux.

Les trois fonctions suivantes sont utilisées dans l'algorithme `LDMon`.

— `send`(φ), envoie φ ainsi que ses sous-formules évaluées à la configuration courante au moniteur M_j, différent du moniteur courant. Afin de déterminer M_j, nous

```
1  (*LDMon*)
2  Input
3      i        (*Indice du moniteur courant*)
4      AE_i     (*Événements atomiques du moniteur courant*)
5      s        (*Indice de la configuration d'entrée*)
6      φ        (*Propriété FTPL à évaluer*)
7      σ_k      (*Suffixe*)
8  Variables
9      t := s   : integer
10 Begin
11 |   WHILE (_iφ̂^s_σk(t) ∉ 𝔹4) DO
12 |   |   _iφ^s_σk(t+1) := P(_iφ^s_σk(t), θ_i(t), AE_i)
13 |   |   t := t + 1
14 |   |   IF (sus(φ)∩AE_i ≠ ∅ ∧ _iφ^s_σk(t) ≠ #) DO
15 |   |   |   send(_iφ^s_σk(t))
16 |   |   |   _iφ^s_σk(t) = #
17 |   |   FI
18 |   |   receive({_jφ^s_σk(t)}_{j≠i})
19 |   |   _iφ^s_σk(t) := _iφ^s_σk(t) ⊓ ⨅_{j≠i} _jφ^s_σk(t)
20 |   ENDWHILE
21 |   IF (∀j ≠ i._jφ̂^s_σk(t) ∉ 𝔹4) DO
22 |   |   broadcast(_iφ^s_σk(t))
23 |   FI
24 |   RETURN _iφ̂^s_σk(t)
25 End
```

FIGURE 7.1 – Algorithme LDMon

considérons $\psi \in sus(\varphi)$, la plus urgente des sous-formules de φ et choisissons j tel que $\psi \in AE_j$. Dans le cas où la formule la plus urgente n'est pas unique, on choisit arbitrairement l'indice j le plus petit.

— `receive`$(\{\varphi, \ldots\})$, reçoit les formules envoyées et diffusées par les autres moniteurs.

— `broadcast`(φ), diffuse φ à tous les autres moniteurs.

Tant que l'évaluation de φ dans $\mathbb{B}_4$ n'est pas obtenue (ligne 11), l'algorithme LDMon boucle de la façon suivante. On fait progresser la formule courante (ligne 12) et on met à jour l'indice de la configuration (ligne 13). Si au moins une des sous-formules urgentes appartient à l'ensemble AE_i des événements atomiques du moniteur courant ($sus(\varphi) \cap AE_i \neq \emptyset$) et si ${}_i\varphi^s_{\sigma_k}(t) \neq \#$ (ligne 14) la formule est envoyée à un moniteur capable d'évaluer une des formules sous-formules les plus urgentes (ligne 15) et est réinitialisée avec la valeur "indéterminé" (ligne 16). Les formules ainsi que les éventuelles diffusions de résultats d'évaluation envoyées par les autres moniteurs sont reçues (ligne 18) et combinées à la formule locale en utilisant l'opérateur $\sqcap$. Si l'évaluation de la formule résultante n'est pas dans $\mathbb{B}_4$, la boucle continue, sinon on passe à l'instruction suivante.

Avant de passer à l'instruction suivante, revenons un court instant sur la condition de la ligne 14 afin d'expliciter la signification de ${}_i\varphi^s_{\sigma_k}(t) \neq \#$. Si ${}_i\varphi^s_{\sigma_k}(t) = \#$, une formule a été déjà été envoyée par le moniteur courant et il n'est donc pas nécessaire de la renvoyer à nouveau. Si ${}_i\varphi^s_{\sigma_k}(t) \neq \#$, soit aucune formule n'a été envoyée par le moniteur, soit une

formule a été déjà été envoyée mais au moins une autre a été reçue d'un autre moniteur. Dans ce cas cet autre moniteur a envoyé cette formule au moniteur courant car ce dernier a été jugé capable d'évaluer une des sous-formules les plus urgentes.

La sortie de la boucle correspond à l'évaluation de la formule courante dans $\mathbb{B}_4$. Si le résultat de cette évaluation n'a pas déjà été diffusé par un autre moniteur (ligne 21), le résultat de cette l'évaluation locale est diffusé (ligne 22). Dans tous les cas, le résultat de l'évaluation est retourné par l'algorithme `LDMon` (ligne 24).

La figure 7.2 résume le fonctionnement de la partie de l'algorithme `LDMon` entre les lignes 10 (**Begin**) et 25 (**End**).

7.3.2/ RÉSULTATS DE CORRECTION ET DE TERMINAISON

La proposition 4, ci-dessous, garantit que l'algorithme `LDMon` fournit un résultat en un nombre fini de configurations, les opérations de communications étant assimilées à des opérations *run*.

Proposition 4 : Existence du résultat de l'algorithme `LDMon`

Soit $\varphi \in Prop_{FTPL}$ et σ_k un suffixe avec $k \geq 0$. Pour une configuration donnée $\sigma(s)$ avec $s > k$ lorsqu'on a un nombre $n = |\mathcal{M}|$ de moniteurs, l'algorithme `LDMon` donne un résultat tel que $\exists i, j.i, j < n \wedge_i \hat{\varphi}^s_{\sigma_k}(s+j) \in \mathbb{B}_4$.

Démonstration. Soient $M_0, M_1, \ldots, M_{n-1}$ n moniteurs. A la configuration d'indice s, si l'un des moniteurs $M_i \in \mathcal{M}$ est capable d'évaluer sa formule dans $\mathbb{B}_4$, la proposition 4 est valide avec $j = 0$, sinon chaque moniteur M_i (où $0 \leq i < n$) fait progresser sa formule de ${}_i\varphi^s_{\sigma_k}(s)$ à ${}_i\varphi^s_{\sigma_k}(s+1)$ et l'envoie à un autre moniteur capable d'évaluer sa sous-formule la plus urgente.

Supposons que ${}_{i_1}\varphi^s_{\sigma_k}(s+1)$ soit envoyée au moniteur $M_{i_2 \neq i_1}$. A la configuration d'indice s, le moniteur $M_{i_2 \neq i_1}$ reçoit ${}_{i_1}\varphi^s_{\sigma_k}(s+1)$ qu'il combine avec ${}_{i_2}\varphi^s_{\sigma_k}(s+1)$, ainsi qu'avec d'éventuelles autres formules reçues d'autres moniteurs, en utilisant l'opérateur $\sqcap$. Si l'une des ses formules (ou un nombre suffisant de leurs sous-formules) peut être évaluée dans $\mathbb{B}_4$, la proposition 4 est valide avec $j = 1$ et $i = i_2$, sinon chaque moniteur M_i fait progresser sa formule de ${}_i\varphi^s_{\sigma_k}(s+1)$ à ${}_i\varphi^s_{\sigma_k}(s+2)$ et l'envoie à un autre moniteur capable d'évaluer sa sous-formule la plus urgente.

Supposons que ${}_{i_2}\varphi^s_{\sigma_k}(s+2)$ soit envoyée au moniteur $M_{i_3 \neq i_2}$. Dans ce cas i_3 est aussi différent de i_1 car toutes les sous-formules pouvant être évaluée en utilisant l'ensemble AE_{i_1} des événements atomiques du moniteur M_{i_1} ont déjà été évaluées précédemment. Ainsi, nous avons réduit le problème de n à $n-1$ moniteurs. Comme pour un moniteur unique le résultat de l'algorithme `LDMon` est $\hat{\varphi}_{\sigma_k}(s)$ (avec $j = 0$), nous pouvons déduire que pour n moniteurs, il y a au moins un moniteur M_{i_0} tel que ${}_{i_0}\hat{\varphi}^s_{\sigma_k}(s+j) \in \mathbb{B}_4$ avec $j < n$. □

Comme nous l'avons expliqué précédemment, durant la description de l'algorithme `LDMon`, lorsqu'on cherche à évaluer $\varphi_{\sigma_k}(s)$, la formule ${}_i\varphi^s_{\sigma_k}(t)$ à la configuration d'indice t par M_i a un résultat ${}_i\hat{\varphi}^s_{\sigma_k}(t) \in \mathbb{B}_4$ ou à une valeur indéterminée. Ce dernier cas correspond à ${}_i\hat{\varphi}^s_{\sigma_k}(t) = \#$. Ainsi ${}_i\hat{\varphi}^s_{\sigma_k}(t) \in \mathbb{B}_5$.

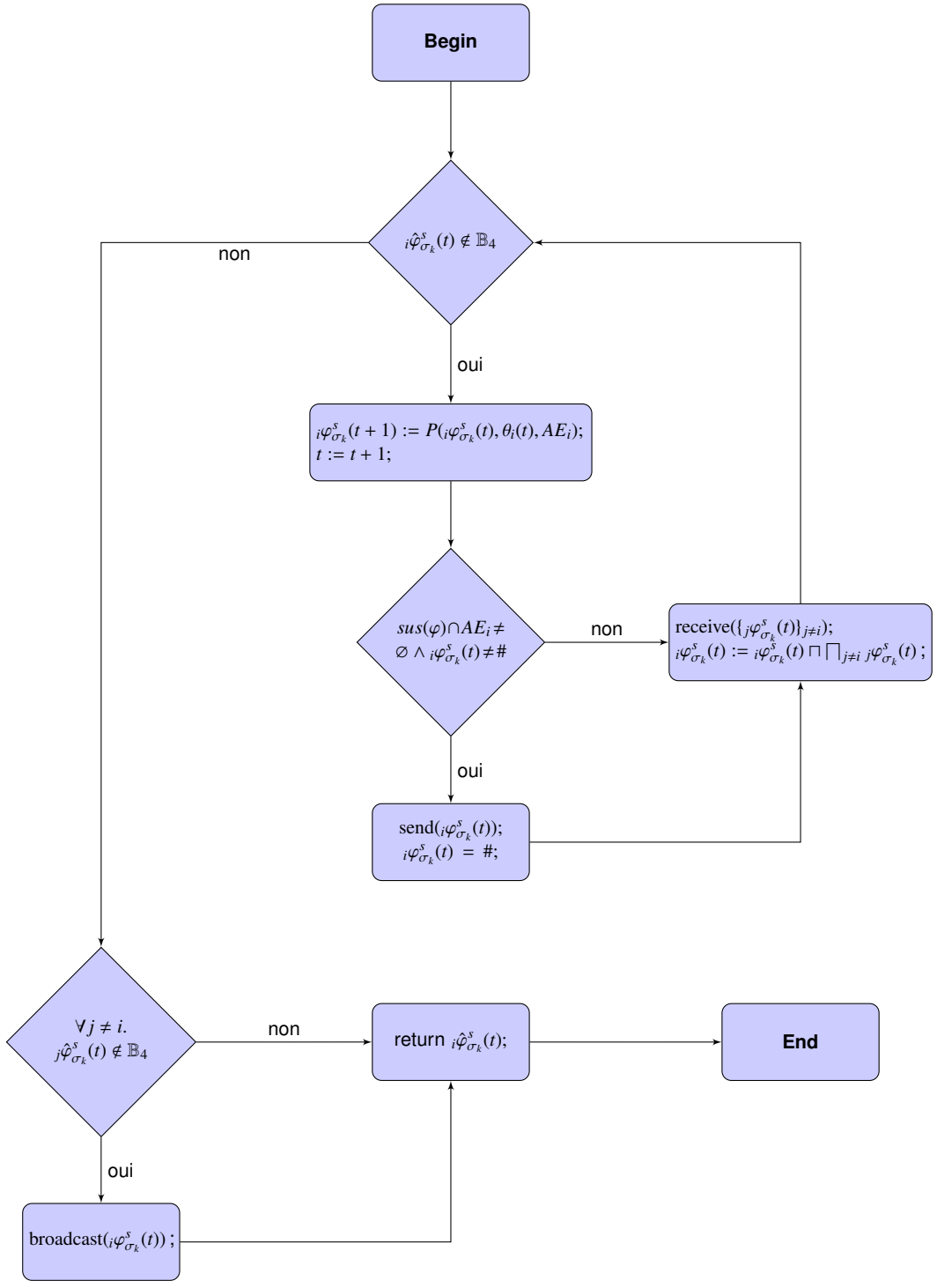


FIGURE 7.2 – Représentation graphique de l'algorithme `LDMon`

Théorème 3 : Correction de la sémantique décentralisée

$$_i\hat{\varphi}^s_{\sigma_k}(t) \neq \# \Leftrightarrow {}_i\hat{\varphi}^s_{\sigma_k}(t) = \hat{\varphi}_{\sigma_k}(s)$$

Démonstration. Nous démontrons ce théorème en prouvant séparément les deux implications qui composent l'équivalence :

(i)⇒ : Si ${}_i\hat{\varphi}^s_{\sigma_k}(t) \neq \#$, un résultat a été obtenu dans $\mathbb{B}_4$, sinon ${}_i\hat{\varphi}^s_{\sigma_k}(t) = \#$. Ainsi, nous avons seulement à vérifier que la fonction de progression des définitions 56 à 61 correspond à la sémantique FTPL dans $\mathbb{B}_4$. Ceci est immédiat en tenant compte du fait que les définition 57 et 58 sont respectivement basées sur les sémantiques des définitions 53 (et aussi de la définition 49, pour ce qui concerne les deux dernières formules) et 54 et que les définitions 59 à 61 sont basées sur la définition 55.

(ii)⇐ : ${}_i\hat{\varphi}^s_{\sigma_k}(t) = \hat{\varphi}_{\sigma_k}(s) \Rightarrow {}_i\hat{\varphi}^s_{\sigma_k}(t) \in \mathbb{B}_4 \Rightarrow {}_i\hat{\varphi}^s_{\sigma_k}(t) \neq \#$.

□

Corolaire 1 : Unicité des résultats de l'algorithme LDMon

Si ${}_i\hat{\varphi}^s_{\sigma_k}(t) \neq \#$ et ${}_j\hat{\varphi}^s_{\sigma_k}(t) \neq \#$ pour $i \neq j$, alors ${}_i\hat{\varphi}^s_{\sigma_k}(t) = {}_j\hat{\varphi}^s_{\sigma_k}(t)$.

Corolaire 2 : Généralisation de l'unicité des résultats de l'algorithme LDMon

Soit ${}_S\varphi^s_{\sigma_k}(t) = \bigsqcap_{i\in S} {}_i\varphi^s_{\sigma_k}(t)$ pour $S \subseteq [1,n]$. Si ${}_S\hat{\varphi}^s_{\sigma_k}(t) \neq \#$ alors, pour tout $j \in S$, ${}_j\hat{\varphi}^s_{\sigma_k}(t) \neq \#$ implique ${}_j\hat{\varphi}^s_{\sigma_k}(t) = {}_S\hat{\varphi}^s_{\sigma_k}(t)$.

Le corolaire 2 permet à un moniteur de simplifier la combinaison de formules avec l'opérateur $\sqcap$. Pour une propriété FTPL donnée, une conjonction de formules reçues d'autres moniteurs avec la formule du moniteur courant peut être remplacée par m'importe laquelle de ces formules pourvue qu'elle soit évaluée à une valeur différente de #.

Exemple 25 : Combinaison de formules

Considérons une nouvelle fois $\varphi = \textbf{before}\ e\ trp$. Soient les composants A,B et C avec leurs moniteurs respectifs M_A, M_B et M_C tels que ${}_i\varphi^s(t) = F_{\mathcal{B}}({}_ie^s(t), {}_itrp^{s-1}(t), {}_i\varphi^{s-1}(t))$ pour $i \in \{A,B,C\}$ (voir la définition 60).
Supposons que $\varphi(s) = F_{\mathcal{B}}(e(s), trp(s-1), \varphi(s-1))$, et que $\varphi(s)$, $e(s)$, $trp(s-1)$ et $\varphi(s-1)$ soient évaluées dans $\mathbb{B}_4$.
D'après le corolaire 2, $e(s) = {}_Ae^s(t) \sqcap {}_Be^s(t) \sqcap {}_Ce^s(t)$ s'il existe au moins un i tel que l'évaluation de ${}_ie^s(t)$ soit dans $\mathbb{B}_4$; dans ce cas, ${}_ie^s(t) = e(s)$.
Toujours en application du corolaire 2, $trp(s-1) = {}_Atrp^{s-1}(t) \sqcap {}_Btrp^{s-1}(t) \sqcap {}_Ctrp^{s-1}(t)$ s'il existe au moins un i tel que l'évaluation de ${}_itrp^{s-1}(t)$ soit dans $\mathbb{B}_4$; dans ce cas, ${}_itrp^{s-1}(t) = trp(s-1)$.
Enfin, en appliquant une dernière fois le corolaire 2, $\varphi(s-1) = {}_A\varphi^{s-1}(t) \sqcap {}_B\varphi^{s-1}(t) \sqcap {}_C\varphi^{s-1}(t)$ s'il existe au moins un i tel que l'évaluation de ${}_i\varphi^{s-1}(t)$ soit dans $\mathbb{B}_4$; dans ce cas, ${}_i\varphi^{s-1}(t) = \varphi(s-1)$.
Par exemple, si ${}_A\varphi^s(t) = F_{\mathcal{B}}(\top, \phi, {}_A\varphi^{s-1}(t))$, ${}_B\varphi^s(t) = F_{\mathcal{B}}(\varepsilon, \top^p, {}_B\varphi^{s-1}(t))$, et ${}_C\varphi^s(t) = F_{\mathcal{B}}(\epsilon, \psi, \top^p)$, avec ϕ, ${}_A\varphi^{s-1}(t)$, ε, ${}_B\varphi^{s-1}(t)$, ϵ et ψ n'étant pas évaluées dans $\mathbb{B}_4$. on a ${}_Ae^s(t) = e(s) = \top$, ${}_Btrp^{s-1}(t) = trp(s-1) = \top^p$ et ${}_C\varphi^{s-1}(t) = \varphi(s-1) = \top^p$. On peut donc en déduire $\varphi(s) = F_{\mathcal{B}}(\top, \top^p, \top^p) = \top^p$. On notera que dans notre exemple chaque terme (étant une sous-formule de φ) de la fonction $F_{\mathcal{B}}$ ne peut être évalué que par un unique moniteur et qu'aucun moniteur ne peut par lui-même parvenir à évaluer plus d'une de ces sous-formules.

Proposition 5 : Correction et unicité du résultat de l'algorithme `LDMon`

Le résultat fourni par l'algorithme `LDMon` répond au problème TPDEP. Pour une configuration donnée $\sigma(s)$, cette réponse est unique.

Démonstration. D'après la proposition 4, l'algorithme `LDMon` fournit un résultat ${}_i\hat{\varphi}^s_{\sigma_k}(s+j)$ pour au moins un moniteur M_i après un nombre fini $j < n$ de configurations. D'après le théorème 3, ce résultat répond au problème TPDEP. De plus, le corolaire 1 établit que pour tout i_0, si ${}_{i_0}\hat{\varphi}^s_{\sigma_k}(s+j)$ est le résultat de l'algorithme `LDMon` pour le moniteur M_{i_0}, alors ${}_{i_0}\hat{\varphi}^s_{\sigma_k}(s+j) = {}_i\hat{\varphi}^s_{\sigma_k}(s+j)$. □

Proposition 6 : Terminaison de l'algorithme `LDMon`

L'algorithme `LDMon` termine toujours, soit à la configuration à laquelle un résultat est fourni, soit à la suivante. De plus, le nombre de configuration nécessaire à la terminaison de l'algorithme `LDMon` est au plus $|\mathcal{M}|$.

Démonstration. Les propositions 4 et 5 établissent que l'algorithme `LDMon` termine et répond au problème TPDEP pour au moins un moniteur M_i après un nombre fini de configurations $j < |\mathcal{M}|$. Un tel moniteur M_i diffuse son résultat aux autres moniteurs avant de finir (ligne 22 de l'algorithme `LDMon` de la figure 7.1). Ceci permet à chaque moniteur pour lequel l'algorithme n'a pas fini à la configuration $s+j$ de recevoir le résultat et de finir l'exécution de l'algorithme `LDMon` à la configuration $s+j+1 \leq s+|\mathcal{M}|$. □

En général les algorithmes décentralisés peuvent avoir des difficultés à trouver un consensus et requièrent souvent des transmissions de données supplémentaires en quantité significative (*communication overhead*). Néanmoins, la proposition 5 garantit l'exactitude et l'unicité d'un résultat, ce qui implique l'obtention d'un consensus. En ce qui concerne les communications supplémentaires induites par la décentralisation de l'évaluation, considérons un système à composants de N composants qui communiquent tous avec un contrôleur central. Dans un contexte centralisé, l'évaluation d'une propriété FTPL φ requiert que N messages (un par composant) soient envoyés au contrôleur central. Avec l'approche décentralisée, en supposant que les événements atomiques nécessaires à l'évaluation de φ soient distribués sur n composants, nous aurions besoin au plus de n^2-1 messages pour évaluer φ. En effet, dans le pire des cas, un moniteur parvient à évaluer φ en $n-1$ configurations au cours de chacune desquelles chaque moniteur envoie (toujours dans le pire des cas) un message ; ainsi durant les $n-1$ configurations un total de n^2-n messages sont envoyés auxquels il faut rajouter les messages envoyés à la configuration suivante par chacun des $n-1$ composants n'ayant pas encore évalué φ, ce qui nous donne n^2-1 messages dans le pire des cas.

Ceci signifie que pour évaluer une formule φ dont les événements atomiques nécessaires à son évaluation sont distribués sur $n = 10$ composants d'un système à composants de $N = 100$ composants, dans le pire des cas, nous aurions besoin, pour une évaluation décentralisée de 99 messages au lieu de 100 messages dans un contexte centralisé, ce qui nous donne un ratio de 99%. En outre, si le nombre total de composants N est nettement supérieur au nombre n de moniteurs nécessaires à l'évaluation décentralisée, ce ratio peut encore diminuer (e.g., 9.9% pour $N = 1000$). En revanche, si une grande proportion des composants du système détient les événements atomiques nécessaires

à l'évaluation de φ, la méthode centralisée produit de meilleurs résultats. Soit q un nombre représentant cette proportion, i.e., $n = qN$; le ratio précédent est $Nq^2 - 1/N$.

Ce résultat est différent de celui de [Bauer et al., 2012] où l'algorithme décentralisé utilise de 1 à 4 fois moins de communications que son équivalent centralisé. Cette différence est due au fait que dans notre cas, dès qu'une propriété est évaluée dans $\mathbb{B}_4$ pour une configuration donnée, une autre évaluation est initiée. Dans le cas de [Bauer et al., 2012], les évaluations se font dans $\mathbb{B}_2$ et lorsqu'un résultat est évalué, celui-ci est final. Pour résumer, l'approche décentralisée est performante dans le cas de systèmes comprenant un grand nombre de composants pour l'évaluation d'une propriété FTPL dont les événements atomiques sont distribués sur une faible proportion des composants.

7.4/ CONCLUSION

Dans ce chapitre, inspiré par l'évaluation décentralisée de formules LTL, nous avons proposé une méthode pour l'évaluation décentralisée de propriétés FTPL utilisant les notions de progression et d'urgence. Une fonction de progression nous permet de réécrire une formule représentant l'évaluation d'une propriété FTPL à une configuration donnée de façon qu'elle soit toujours pertinente à la configuration suivante. Dans le contexte de l'évaluation décentralisée, nous décomposons les formules en sous-formules auxquelles nous attribuons un niveau d'urgence pour déterminer l'ordre dans lequel les évaluer. Nous avons aussi posé formellement le problème de l'évaluation décentralisée et avons proposé un algorithme pour le résoudre ainsi que des résultats de correction et de terminaison.

8

POLITIQUES D'ADAPTATION

Dans [Dormoy et al., 2010], il a été proposé des politiques d'adaptation utilisant des schémas temporels (différents de la logique FTPL) prenant en compte des événements externes. Ainsi, la logique FTPL pouvait être utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permettait pas la définition de politiques d'adaptation pouvant supporter des événements externes. Nous avons donc étendu la logique FTPL avec des événements externes afin de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité par rapport à [Dormoy et al., 2010]. Dans ce chapitre nous allons, comme dans [Kouchnarenko et al., 2014a], intégrer la logique FTPL dans des politiques d'adaptation. Nous allons aussi définir le problème de l'adaptation et proposer une relation de simulation permettant d'établir que ce problème est semi-décidable. Puis, nous fournirons un algorithme pour l'utilisation de politiques d'adaptation pour l'enforcement de propriétés FTPL sur des systèmes à composants reconfigurables. Enfin, nous présenterons l'usage du fuzzing pour le test de politiques d'adaptation.

Sommaire

8.1/ INTÉGRATION DE PROPRIÉTÉS TEMPORELLES DANS DES POLITIQUES D'ADAPTATION

Bien que l'un des principaux avantages des systèmes à composants reconfigurables réside dans leur capacité à faire évoluer leur architecture à l'exécution, les reconfigurations dynamiques ne peuvent pas s'effectuer à n'importe quel moment, mais seulement dans des circonstances appropriées. Afin de pouvoir superviser et influencer dynamiquement les reconfigurations de systèmes à composants, cette section décrit des politiques d'adaptation basées sur des règles d'adaptation indiquant les reconfigurations dont l'opportunité et la pertinence correspondent le mieux à la situation courante.

Le but de nos travaux est de guider et contrôler les reconfigurations de systèmes à composants. Nous nous sommes donc intéressés aux politiques d'adaptation qui influencent à l'exécution l'évolution de tels systèmes (comme le fait Tangram4Fractal [Chauvel et al., 2009] avec Julia [1], l'implémentation de référence de Fractal).

Or, ces politiques d'adaptation n'autorisaient pas l'expression de contraintes temporelles. C'est pourquoi un modèle de système à composants permettant la mise en place de politiques d'adaptation utilisant les schémas temporels fournis par qMEDL [Gonnord et al., 2009] (une logique, basée sur MEDL, permettant d'exprimer des propriétés relatives à des contraintes sur des quantités de ressources) a été introduit dans [Dormoy et al., 2010].

Ainsi, la logique FTPL était utilisée pour exprimer des contraintes architecturales et temporelles sur des systèmes à composants mais ne permettait pas la prise en compte d'événements externes, contrairement à qMDEL qui était utilisé pour déclencher l'application de politiques d'adaptation. C'est pourquoi nous avons [Kouchnarenko et al., 2014a] étendu la logique FTPL avec des événements externes afin de pouvoir utiliser la même logique pour exprimer les politiques d'adaptation et les contraintes architecturales sur les systèmes à composants sans perdre en expressivité.

Rappelons que la logique FTPL est basée sur les travaux de Dwyer [Dwyer et al., 1999] qui partent du constat que certaines propriétés simples à exprimer en langage naturel sont difficiles à décrire avec une logique temporelle. A partir de ces constatations, la notion de schémas de spécification a été proposée dans le cadre du projet Bandera [2]. Dans ces travaux ont retrouve la notion de portée (*scope*) d'une propriété qui permet de ne considérer cette propriété qu'avant et/ou qu'après un événement donné. On retrouve cette notion dans la logique FTPL exprimée par les propriétés temporelles de type **after**, **before** et **until**.

Dans cette section, nous proposons une définition des politiques d'adaptation intégrant les propriétés de la logique FTPL. Dans un second temps, nous introduisons la notion de restriction d'un système à composants reconfigurable par des politiques d'adaptation.

1. http://fractal.ow2.org/julia/index.html
2. http://bandera.projects.cs.ksu.edu/

8.1.1/ DÉFINITION DES POLITIQUES D'ADAPTATION

Afin de prendre en compte des contraintes au niveau des ressources d'un système, des événements de son environnement ou même des propriétés temporelles basées sur des séquences de configurations, nous proposons d'étendre les politiques d'adaptation en y intégrant les propriétés de la logique FTPL. Les politiques d'adaptation sont définies par *a*) des opérations de reconfigurations architecturales qui spécifient les modifications possibles de l'architecture ; et *b*) des règles d'adaptation permettant de faire le lien entre entre les propriétés du système à composant considéré et le besoin d'activer une reconfiguration donnée.

Comme dans [Chauvel et al., 2009, Dormoy et al., 2010], ce besoin est quantifié en utilisant la logique floue [Pedrycz, 1993] ; plus précisément, on associe à un besoin une valeur (qu'on appelle valeur floue) appartenant à un ensemble qu'on appelle un type flou (par exemple $\{\texttt{low}, \texttt{medium}, \texttt{high}\}$).

Définition 64 : Politiques d'adaptation

Soient un système à composants reconfigurable $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$ et $Ftype$ un ensemble de types flous. Étant donné une configuration $\sigma(i) \in C$, un ensemble de politiques d'adaptation fini AP pour $\sigma(i)$ est composé d'éléments $A = \langle R_N, R_R \rangle$, où :

- $R_N \subseteq \mathcal{R}$ est un ensemble fini (non vide) de reconfiguration gardées,
- R_R est un ensemble fini (non vide) de règles d'adaptation, chacune étant un tuple $\langle F, B, G, I \rangle$, où :
  - $F \in Ftype$ est un type flou,
  - $B \subseteq \{\phi_\sigma(i) = value \mid \phi \in Prop_{FTPL} \wedge value \in \mathbb{B}_4\}$ est un ensemble de propriétés de $Prop_{FTPL}$ évaluées dans $\mathbb{B}_4$ à la configuration $\sigma(i)$,
  - $G \subseteq \{cp_\sigma(i) = value \mid cp \in CP \wedge value \in \mathbb{B}_2\}$ est un ensemble de propriétés de configuration de CP évaluées dans $\mathbb{B}_2$ à la configuration $\sigma(i)$,
  - $I \subseteq R_N \times F$ est une relation entre reconfigurations et valeurs floues.

On note respectivement $B_{\sigma(i)}$ et $G_{\sigma(i)}$ les conjonctions des propriétés de B et de G à la configuration $\sigma(i)$.

Rappelons-nous le système composite **Location** présenté à la section 3.1 qui supporte le retrait ou l'ajout de ses sous-composants **GPS** et **Wi-Fi**. Bien sûr, à un instant donné au moins un des deux doit être présent ; par contre, dans certains cas il est pertinent de retirer l'un de ces composants.

Par exemple, lorsque le niveau d'énergie des batteries du véhicule est bas, le composant **Wi-Fi** peut être retiré, puis rajouté plus tard lorsque les batteries sont rechargées. De plus, lorsque le véhicule entre dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible, il est pertinent de retirer le composant **GPS** qui peut être rajouté après que le véhicule soit sorti de cette zone.

La figure 8.1 montre le contenu d'un fichier décrivant la politique d'adaptation `cycabgps` qui est écrite en utilisant une syntaxe inspirée par les politiques d'adaptation de Tangram4Fractal [Chauvel et al., 2009]. Cette politique d'adaptation peut déclencher les re-

configurations `addgps` et `removegps` qui permettent respectivement d'ajouter ou d'enlever le composant **GPS**.

```
 1 policy cycabgps
 2
 3   event entry
 4   event exit
 5   event start
 6
 7   when (after start,exit (P_TRUE4 until entry)) = P_TRUE4
 8     if (gps in Components) = FALSE
 9     then utility of addgps is low
10
11   when (Power < 33) = TRUE4
12     if (gps in Components) = FALSE
13     then utility of addgps is low
14
15   when (Power < 33) = FALSE4
16     if (gps in Components) = FALSE
17     then utility of addgps is high
18
19   when (Power < 33) = FALSE4
20     if (gps in Components and wifi in Components) = TRUE
21     then utility of removegps is low
22
23   when (after start,exit (P_TRUE4 until entry)) = P_TRUE4
24     if (gps in Components and wifi in Components) = TRUE
25     then utility of removegps is high
26
27   when (Power < 33) = TRUE4
28     if (gps in Components and wifi in Components) = TRUE
29     then utility of removegps is high
30
31 end policy
```

FIGURE 8.1 – La politique d'adaptation `cycabgps`

Le fichier de définition de la politique d'adaptation `cycabgps` déclare dans un premier temps trois événements externes (lignes 3 à 5) qui sont utilisés dans les règles d'adaptation. L'événement *start* a lieu seulement au moment où la politique d'adaptation devient effective, tandis que *entry* et *exit* ont respectivement lieu lorsque le véhicule entre dans une "zone Wi-Fi" et lorsqu'il en sort. La suite de la politique d'adaptation consiste en une séquence de règles d'adaptation où le tuple $\langle F, B, G, I \rangle$ de la définition 64 est représenté par `when` B `if` G `then utility of` ope `is` $fvalue$ où $(ope, fvalue) \in I$.

Exemple 26 : Règle d'adaptation d'une politique d'adaptation

Pour la politique d'adaptation `cycabgps` de la figure 8.1, nous avons l'ensemble de reconfigurations architecturales $R_N = \{\texttt{addgps}, \texttt{removegps}\}$ et $Ftype = \{\{\texttt{low}, \texttt{medium}, \texttt{high}\}\}$ qui contient le seul type flou utilisé dans cette politique d'adaptation.
Pour la règle d'adaptation des lignes 23 à 25, nous avons, en utilisant les notations de la définition 64, $F = \{\texttt{low}, \texttt{medium}, \texttt{high}\}$, $B = \{\textbf{after}\ start, exit\ (\top^p\ \textbf{until}\ entry) = \top^p\}$, $G = \{gps \in Components \wedge wifi \in Components = \top\}$ et $I = \{(\texttt{removegps}, \texttt{high})\}$.
Cette règle d'adaptation exprime le fait que lorsque l'expression B est valide (i.e., le véhicule est dans une "zone Wi-Fi" — cf. table 6.4 pour le détail de l'évaluation), si les deux composants **GPS** et **Wi-Fi** sont présents, alors l'utilité d'enlever le composant **GPS**, au moyen de la reconfiguration `removegps` est haute.

8.1.2/ RESTRICTION PAR POLITIQUES D'ADAPTATION

Afin de pouvoir décrire le comportement d'un système à composants reconfigurable lorsque des politiques d'adaptation lui sont appliquées, nous définissons la notion de restriction d'un tel système par des politiques d'adaptation.

Soit un système à composants reconfigurable $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l\rangle$ et AP_S un ensemble fini de politiques d'adaptation pour S. Nous définissons à présent comment les politiques d'adaptation affectent le comportement du système à composants reconfigurable.

Définition 65 : Restriction par politiques d'adaptation

La restriction de $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l\rangle$ par les politiques d'adaptation de AP_S, définie par $S \triangleleft AP_S = \langle C \triangleleft AP_S, C^0 \triangleleft AP_S, \mathcal{R}_{run}, \rightarrow, l\rangle$, où $C \triangleleft AP_S$ est le plus petit ensemble tel que si $c \in C$ et $A \in AP_S$ alors $c \triangleleft A \in C \triangleleft AP_S$, $\mathcal{R}_{run} \cap (\cup_{A \in AP_S} R_N) \neq \emptyset$, $l : C \triangleleft AP_S \rightarrow CP$ est une fonction totale d'interprétation et pour toute opération de reconfiguration $ope \in \mathcal{R}_{run}$, la relation de transition $\rightarrow \in C \triangleleft AP_S \times \mathcal{R}_{run} \times C \triangleleft AP_S$ est le plus petit sous-ensemble de triplets $(c \triangleleft A, ope, c' \triangleleft A)$ satisfaisant les règles suivantes, où wp_{ope} représente la plus faible pré-condition (*weakest precondition*) devant être satisfaite pour appliquer ope :

$$\text{[ACT1]}\ \frac{c \xrightarrow{ope} c'}{c \triangleleft A \xrightarrow{ope} c' \triangleleft A}\ (ope \in \textstyle\bigcup_{A \in AP_S} R_N) \wedge (l(c) \Rightarrow wp_{ope}) \wedge B_c \wedge G_c$$

$$\text{[ACT2]}\ \frac{c \xrightarrow{ope} c'}{c \triangleleft A \xrightarrow{ope} c' \triangleleft A}\ (ope \notin \textstyle\bigcup_{A \in AP_S} R_N) \wedge (l(c) \Rightarrow wp_{ope})$$

Cette définition signifie que pour $S \triangleleft AP_S = \langle C \triangleleft AP_S, C^0 \triangleleft AP_S, \mathcal{R}_{run}, \rightarrow, l\rangle$, toutes les configurations de $C \triangleleft AP_S$ sont atteignables depuis des configurations initiales soit par des opérations de reconfiguration obéissant aux politiques d'adaptation (règle *[ACT1]*), soit par des opérations de reconfiguration qui ne sont pas considérées dans les politiques d'adaptation (règle *[ACT2]*).

8.2/ APPLICATION DES POLITIQUES D'ADAPTATION À L'EXÉCUTION

Dans cette section, nous proposons une relation de simulation sur les modèles de systèmes à composants reconfigurables et nous définissons le problème de l'adaptation. En utilisant cette relation, nous établissons que le problème de l'adaptation est au moins semi-décidable. Ensuite, nous donnons un algorithme pour l'enforcement des politiques d'adaptation des systèmes à composants reconfigurables. Enfin, nous établissons la terminaison de cet algorithme et énonçons un résultat de correction que nous démontrons.

8.2.1/ PROBLÈME DE L'ADAPTATION

Étant donné un système à composants reconfigurable $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l\rangle$ et un ensemble de politiques d'adaptation AP_S. On souhaite vérifier que malgré l'occurrence de reconfigurations induites par l'application de politiques d'adaptation, le comportement de S est toujours conforme à celui attendu de la part de S sans l'application de politiques d'adaptation. Plus formellement, pour deux systèmes à composants reconfigurables S et $S \triangleleft AP_S$, le problème consiste à décider si le comportement de S lorsqu'il est guidé et contrôlé par les politiques d'adaptations AP_S est aussi un comportement de S. Pour aborder ce problème, nous nous proposons d'utiliser la notion de *ready simulation* [Bloom et al., 1995].

Définition 66 : Ready Simulation

Soient $S_1 = \langle C_1, C_1^0, \mathcal{R}_{run}, \rightarrow_1, l_1\rangle$ et $S_2 = \langle C_2, C_2^0, \mathcal{R}_{run}, \rightarrow_2, l_2\rangle$ deux systèmes à composants reconfigurables sur $\mathcal{R}_{run}$. Une relation binaire $\simeq \subseteq C_1 \times C_2$ est une *ready simulation* si et seulement si, pour toute opération de reconfiguration ope de $\mathcal{R}_{run}$, $(c_1, c_2) \in \simeq$ implique

i) Lorsque $(c_1, ope, c_1') \in \rightarrow_1$, alors il existe $c_2' \in C_2$ tel que $(c_2, ope, c_2') \in \rightarrow_2$ et $(c_1', c_2') \in \simeq$.

ii) Lorsque $c_1 \xrightarrow{ope}\!\!\!\!\!/\;\;$, alors $c_2 \xrightarrow{ope}\!\!\!\!\!/\;\;$.

Étant donné $S_1 = \langle C_1, C_1^0, \mathcal{R}_{run}, \rightarrow_1, l_1\rangle$ et $S_2 = \langle C_2, C_2^0, \mathcal{R}_{run}, \rightarrow_2, l_2\rangle$ deux systèmes à composants reconfigurables, on dit que S_1 et S_2 sont ready-similaires et on écrit $S_1 \simeq S_2$, si $\forall c_1^0 \in C_1^0, \exists c_2^0 \in C_2^0.(c_1^0, c_2^0) \in \simeq$.

Comme dans [Bloom et al., 1995], on définit l'"ensemble ready" (*ready-set*). Ainsi, étant donné $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l\rangle$, cet ensemble, pour $c \in C$, est : $readies(c) = \{ope \mid ope \in \mathcal{R}_{run} \wedge c \xrightarrow{ope}\}$.

Il est intéressant de noter, comme conséquence immédiate de la définition 66, que $(c_1, c_2) \in \simeq$ implique $readies(c_1) = readies(c_2)$. Ainsi, il est suffisant de prouver l'inégalité des "ensembles ready" pour monter que la *ready simulation* n'est pas satisfaite entre les deux configurations.

Afin d'établir si $S \triangleleft AP_S \simeq S$ ou non, ce qui pourra nous permettre de fournir un résultat de correction pour la restriction par un ensemble de politiques d'adaptation, nous considérons le problème de décision suivant.

Problème de l'adaptation
Entrée : Un système à composants reconfigurable $S = \langle C, C^0, \mathcal{R}_{run}, \rightarrow, l\rangle$, $c \in C$ et l'en-

semble $AP \subseteq AP_S$ des politiques d'adaptation pour c.
Sortie : vrai si $\forall A \in AP.c \triangleleft A \simeq c$ et **faux** sinon.

Pour le système à composants reconfigurable S restreint par ses politiques d'adaptation, nous définissions l'"ensemble ready" pour $c \in C$ conformément à la définition 65. Ainsi, on a : $readies(c \triangleleft A) = \{ope \mid (ope \in \mathcal{R}_{run} \setminus \cup_{A \in AP} R_N) \wedge (l(c) \Rightarrow wp_{ope}) \wedge c \xrightarrow{ope}\} \cup \{ope \mid (ope \in \bigcup_{A \in AP} R_N) \wedge (l(c) \Rightarrow wp_{ope}) \wedge B_c \wedge G_c \wedge c \xrightarrow{ope}\}$. On voit donc, comme ci-dessus, que $c \triangleleft A \simeq c$ implique que $readies(c \triangleleft A) = readies(A)$. Lorsque $S \triangleleft AP_S$ qui autorise les reconfigurations en fonction des politiques d'adaptation de AP_S est un système de transitions fini sans chemins infinis, le problème de l'adaptation peut être décidé. Mais, en général, si $S \triangleleft AP_S$ est un système de transitions infini (comme l'est aussi S), la conclusion **vrai** du problème de l'adaptation ne peut pas être atteinte à cause des espaces d'états (*state spaces*) infinis et des chemins infinis. Ainsi, le problème de l'adaptation ne peut être décidé dans le cas général. Néanmoins, quand les "ensembles ready" sont différents, on peut atteindre la conclusion "faux", même si $S \triangleleft AP_S$ est infini, d'où la proposition 7 suivante.

Proposition 7 : Semi-décidabilité du problème de l'adaptation

Le problème de l'adaptation est au moins semi-décidable.

8.2.2/ L'ALGORITHME `AdaptEnfor`

Les politiques d'adaptation peuvent être utilisées pour spécifier des mécanismes de réflexion et d'enforcement (voir section 6.1.1.2). La notion de réflexion consiste à ce que des comportements non souhaités déclenchent des reconfigurations correctives via une ou plusieurs politiques d'adaptation. La notion d'enforcement, illustrée par l'algorithme `AdaptEnfor` de la figure 8.2, signifie qu'aucune reconfiguration qui mènerait le système à se comporter d'une façon non souhaitée n'est autorisée.

Cet algorithme accepte en entrée les deux objets suivants :

1. Un système à composants reconfigurable générique $gcbs$. Il s'agit d'un objet qui encapsule un système à composants reconfigurable. Un tel objet nous permet de gérer un système à composants reconfigurable indépendamment du modèle de système à composants (e.g., Fractal ou FraSCAti) utilisée.
2. Un vecteur v contenant des reconfigurations candidates ordonnées par priorité en fonction de leur utilité. Cette utilité est calculée en prenant en compte les valeurs floues (e.g., des valeurs de l'ensemble {`low`, `medium`, `high`} vu précédemment) associées aux différentes reconfigurations mentionnées dans les politiques d'adaptation.

Chacune des variables $currentConf$, $targetConf$ et $endConf$ représente une configuration tandis que la variable r désigne une reconfiguration. L'idée de base de cet algorithme consiste à stocker la configuration courante du système à composants dans $currentConf$ et de stocker dans $targetConf$ le résultat de l'application à $currentConf$ de la reconfiguration r. Si la configuration stockée dans $targetConf$ satisfait les propriétés que l'on souhaite préserver, elle est appliquée au système à composants. Afin de vérifier la bonne application de cette reconfiguration, on stocke la configuration résultante du système à composants dans $endConf$ que l'on compare à $targetConf$. Si le résultat de cette comparaison est acceptable (au sens de la relation de comparaison utilisée), on considère

```
1  (*AdaptEnfor*)
2  Input
3      gcbs    (*Système à composants reconfigurable générique*)
4      v       (*Vecteur de reconfigurations candidates ordonnées par priorité*)
5  Variables
6      currentConf,
7      targetConf,
8      endConf : configuration
9      r : reconfiguration
10 Begin
11 |   currentConf := retrieveConf(gcbs)
12 |   WHILE (size(v) > 0) DO
13 |   |   r := getNextElement(v)
14 |   |   remove(v, r)
15 |   |   targetConf := applyReconf(currentConf, r)
16 |   |   IF (preserveEnforProps(targetConf)) DO
17 |   |   |   applyToSystem(targetConf, gcbs)
18 |   |   |   endConf := retrieveConf(gcbs)
19 |   |   |   IF (endConf ≡ targetConf) DO
20 |   |   |   |   sendEvent(r, normal)
21 |   |   |   |   break
22 |   |   |   ELSE
23 |   |   |   |   applyToSystem(currentConf, gcbs)
24 |   |   |   |   endConf := retrieveConf(gcbs)
25 |   |   |   |   IF (endConf ≡ currentConf) DO
26 |   |   |   |   |   sendEvent(r, exceptionnal)
27 |   |   |   |   |   break
28 |   |   |   |   ELSE
29 |   |   |   |   |   systemExit
30 |   |   |   |   FI
31 |   |   |   FI
32 |   |   FI
33 |   ENDWHILE
34 End
```

FIGURE 8.2 – Algorithme `AdaptEnfor`

que la reconfiguration s'est bien passée, sinon on applique au système à composants la configuration initialement stockée dans $currentConf$ afin de revenir à l'état initial.

Notons que dans l'algorithme `AdaptEnfor`, la relation ≡ utilisée pour comparer des configurations peut être implémentée de différentes façons en utilisant diverses relations de (pré-)congruences. Ainsi, on peut utiliser l'égalité stricte des ensembles $Elem$ et Rel de la définition 27 ou une égalité "partielle" de ces ensembles en ignorant la valeur de certaines ou de toutes les variables. Ceci peut s'avérer utile dans le cas de variables changeant très fréquemment. Il est aussi possible d'utiliser le raffinement structurel [Dormoy et al., 2012b], ou d'autres relations compatibles avec la relation de reconfiguration.

Notre algorithme contient cinq fonctions :

1. retrieveConf(s) qui retourne la configuration du système à composants reconfigurable générique encapsulé par s,
2. size(v) qui retourne le nombre d'éléments du vecteur v,
3. getNextElement(v) qui retourne le prochain élément du vecteur v,
4. applyReconf(c, r) qui retourne la configuration résultant de l'application de la reconfiguration r à la configuration c, et enfin
5. preserveEnforProps(c) qui retourne $\top$ si toutes les propriétés que l'on souhaite préserver par le mécanisme d'enforcement sont satisfaites à la configuration c, sinon $\bot$ est retourné.

Il y a aussi cinq procédures dans cet algorithme :

1. remove(v, e) permet de retirer l'élément e du vecteur v,
2. applyToSystem(c, s) lance la reconfiguration du système à composants reconfigurable générique encapsulé par s pour qu'il se trouve dans la configuration cible c en utilisant un mécanisme similaire à celui décrit dans [Boyer et al., 2013],
3. sendEvent(r, arg) envoie l'événement "r **normal**" ou "r **exceptional**" où r est une reconfiguration et arg est soit "`normal`", soit "`exceptional`",
4. break qui permet de sortir de la boucle "while" courante, et
5. systemExit qui termine l'exécution courante du programme.

Le fonctionnement de l'algorithme `AdaptEnfor` est le suivant.

(1) On initialise la variable $currentConf$ avec la configuration courante du système à composants reconfigurable encapsulé par $gcbs$ (ligne 11),

(2) tant que le vecteur v contenant des reconfigurations candidates ordonnées par priorité n'est pas vide, on itère (ligne 12) de la façon suivante,

- **(A)** on initialise la variable r avec la première configuration du vecteur v, c'est-à-dire avec la reconfiguration candidate la plus prioritaire (ligne 13),
- **(B)** on enlève cette reconfiguration du vecteur v afin de ne plus avoir à la traiter par la suite (ligne 14),
- **(C)** on applique la reconfiguration r à la configuration courante $currentConf$ et on stocke la configuration résultante dans la variable $targetConf$ (ligne 15),
- **(D)** si les propriétés que l'on souhaite préserver via l'enforcement sont satisfaites par la configuration cible $targetConf$ (ligne 16), on continue
  - **(I)** on applique la configuration cible $targetConf$ au système à composants reconfigurable encapsulé par $gcbs$ (ligne 17),
  - **(II)** on initialise la variable $endConf$ avec la nouvelle configuration du système à composants reconfigurable encapsulé par $gcbs$ (ligne 18),
  - **(III)** si la configuration $endConf$ du système à composants reconfigurable encapsulé par $gcbs$ et la configuration cible $targetConf$ sont liées par la relation $\equiv$ (ligne 19), on continue,
    - **(a)** on envoie l'événement "r **normal**" pour signifier que la reconfiguration r s'est finie normalement (ligne 20),
    - **(b)** on force la sortie de la boucle "while" (ligne 21),

(IV) dans le cas où configuration $endConf$ du système à composants reconfigurable encapsulé par $gcbs$ et la configuration cible $targetConf$ ne sont pas équivalentes (ligne 19), on procède comme suit,

(a) on applique la configuration initiale $currentConf$ au système à composants reconfigurable encapsulé par $gcbs$ (ligne 23),

(b) on initialise la variable $endConf$ avec la nouvelle configuration du système à composants reconfigurable encapsulé par $gcbs$ (ligne 24),

(c) si la configuration $endConf$ du système à composants reconfigurable encapsulé par $gcbs$ et la configuration initiale $currentConf$ sont liées par la relation $\equiv$ (ligne 25), on continue,

(i) on envoie l'événement "r **exceptional**" pour signifier que la reconfiguration r s'est finie anormalement (ligne 26),

(ii) on force la sortie de la boucle "while" (ligne 27),

(d) dans le cas où la configuration $endConf$ du système à composants reconfigurable encapsulé par $gcbs$ et la configuration initiale $currentConf$ ne sont pas liées par la relation $\equiv$ (ligne 25), cela signifie que l'on n'a pas pu appliquer la configuration cible $targetConf$, ni pu revenir à la configuration initiale $currentConf$; comme on ne maîtrise plus le système à composants reconfigurable encapsulé par $gcbs$, on arrête l'exécution du programme (ligne 29) [3].

(E) dans le cas où toutes les propriétés que l'on souhaite préserver via l'enforcement ne sont satisfaites par la configuration cible $targetConf$ (ligne 16), on procède à l'itération suivante,

(3) lorsque l'on sort de la boucle "while" (ligne 33) une reconfiguration du vecteur v a été choisie et appliquée (avec succès ou pas), sinon le vecteur v est vide, soit car il était déjà vide au début de l'exécution de l'algorithme, soit aucune des reconfigurations qu'il contenait initialement ne permet de satisfaire les propriétés que l'on souhaite préserver via l'enforcement.

La figure 8.3 résume le fonctionnement de la partie de l'algorithme `AdaptEnfor` entre les lignes 10 (**Begin**) et 34 (**End**).

8.2.3/ CORRECTION DE L'ADAPTATION

La façon dont nous appliquons l'enforcement via les politiques d'adaptation respecte les principes de correction ("soundness") et de transparence [Ligatti et al., 2005]. Étant donné un ensemble de propriétés que l'on souhaite préserver via enforcement à l'exécution, le mécanisme que nous utilisons est *a*) correct car il prévient (en n'entrant pas dans le corps de la clause "IF" de la ligne 16) l'occurrence de reconfigurations qui mèneraient le système, à la prochaine configuration, dans un état qui ne satisferait pas les propriétés que l'on souhaite préserver via enforcement, et *b*) transparent car il autorise (en entrant dans le corps de la clause "IF" de la ligne 16) l'occurrence de reconfigurations (s'il y en a) qui permettent au système de satisfaire ces propriétés.

3. Dans le cas d'un système critique, l'exécution ne serait pas stoppée ainsi ; nous nous efforcerions de pouvoir maintenir le fonctionnement du système à composants en utilisant, par exemple, des mécanismes de reprise sur erreur qui sont en dehors du contexte des travaux présentés dans ce document.

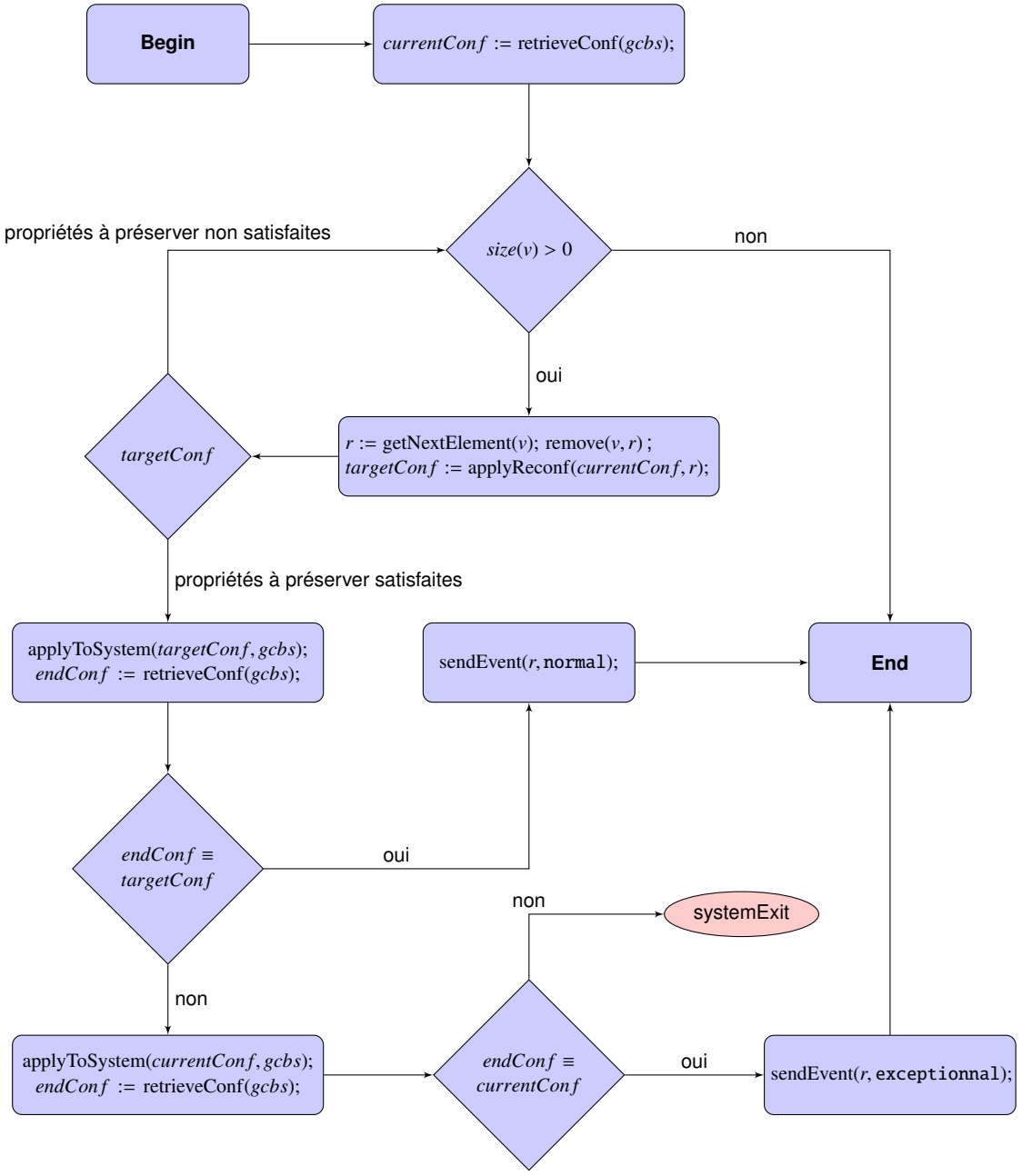


FIGURE 8.3 – Représentation graphique de l'algorithme `AdaptEnfor`

Pour être complet, évoquons le cas où des changements surviennent dans le système à composants reconfigurable pendant que ce système est reconfiguré par une exécution de l'algorithme `AdaptEnfor`. Dans ce cas, il y a deux possibilités : la nouvelle configuration du système est liée par la relation $\equiv$ de l'algorithme `AdaptEnfor` à la configuration cible $targetConf$ ou bien non. Dans l'affirmative, l'algorithme continue comme prévu, sinon un retour à la configuration initiale $currentConf$ est effectué.

Comme la boucle "while" commençant à la ligne 12 de la figure 8.2 finit lorsque la taille du vecteur v est nulle et que la taille de v diminue (ligne 14) à chaque itération, l'algorithme `AdaptEnfor` finit toujours, d'où la proposition 8 suivante.

Proposition 8 : Terminaison de l'algorithme `AdaptEnfor`

L'algorithme `AdaptEnfor` termine toujours.

Quand l'algorithme `AdaptEnfor` se termine sans opération de reconfiguration sélectionnée pour être appliquée à la configuration courante, i.e., lorsque la taille du vecteur v est égale à 0 dans la boucle "while", cela signifie que l'ensemble $\{ope \mid ope \in \bigcup_{A \in AP} R_N \wedge (l(c) \Rightarrow wp_{ope}) \wedge B_c \wedge G_c \wedge c \xrightarrow{ope}\}(\subseteq readies(c \triangleleft A))$ est vide. Dans ce cas, comme chaque politique d'adaptation pour c spécifie au moins une opération de reconfiguration, les "ensembles ready" pour c et $c \triangleleft A$ sont différents. De cette façon l'algorithme `AdaptEnfor` permet de répondre au problème de l'adaptation par **faux**. D'où le théorème 4 :

Théorème 4 : Correction de l'adaptation

Si une configuration c n'est pas atteignable dans S, alors pour tout ensemble AP_S de politiques d'adaptation pour S, c n'est pas atteignable dans $S \triangleleft AP_S$.

Démonstration. La correction est immédiate du fait que si, pour une politique d'adaptation A de AP_S, on omet les ensembles B et G (voir définition 64) restreignant le comportement de $S \triangleleft AP_S$, on obtient un comportement de S. □

La réflexion à l'exécution peut être appliquée d'une façon similaire au mécanisme utilisé ci-dessus pour l'application de l'enforcement à l'exécution. La principale différence réside dans le fait que l'enforcement prévient l'occurrence de reconfigurations spécifiques afin d'éviter des comportements non souhaités avant qu'ils aient lieu, tandis que la réflexion consiste à détecter de tels comportements pour y remédier par des actions correctives via des reconfigurations déclenchées par des politiques d'adaptation. De telles actions peuvent aller jusqu'à l'arrêt total du système dans le cas de la détection de violations qui pourraient le justifier.

Il est intéressant de noter que lorsque la violation d'une propriété FTPL est détectée dans le contexte de la réflexion à l'exécution, la trace d'exécution constitue en elle-même un contre-exemple de la propriété concernée. Par exemple, considérons la propriété FTPL φ = **after** $removewifi$ **terminates** (**before** $addwifi$ **terminates eventually** $\neg powerlow$). Lorsqu'elle est évaluée à "potentiellement vrai" ($\top^p$), cette propriété permet de valider que le composant **Wi-Fi** du système composite **Location** présenté à la section 3.1, après avoir été retiré (via l'opération de reconfiguration $removewifi$) n'est pas rajouté (via $addwifi$) avant qu'un niveau d'énergie considéré comme n'étant pas faible ne soit mesuré.

Considérons un parcours du véhicule autonome au cours duquel le niveau des batterie baisse jusqu'à être considéré comme étant faible. Il peut être pertinent, pour économiser l'énergie restante, de retirer le composant **Wi-Fi** du système composite **Location** en ne laissant que le composant **GPS**. Si, en continuant son parcours, le véhicule entre dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible, il est vital de rajouter le composant **Wi-Fi** au plus vite. Ceci aurait pour conséquence que la propriété φ soit évaluée à "faux" ($\bot$) et la trace ayant mené à cette évaluation constituerait un contre-exemple de φ.

8.3/ USAGE DU FUZZING POUR LE TEST DE POLITIQUES D'ADAPTATION

Le test de la robustesse de politiques d'adaptation peut s'avérer compliqué et nécessite souvent de nombreux cas de tests. C'est la raison pour laquelle nous utilisons le *fuzzing* pour la génération de ces tests. Dans cette section, nous présentons brièvement le fuzzing avant d'expliquer comment nous l'utilisons pour le test de politiques d'adaptation.

8.3.1/ PRÉSENTATION DU FUZZING

Le fuzzing [Takanen et al., 2008] (ou *fuzz testing*) est une technique de test logiciel utilisée pour découvrir des erreurs de codage et des failles de sécurité dans des programmes, des systèmes d'exploitation ou sur les réseaux de communication. Cette technique consiste a envoyer une quantité massive de données, appelée fuzz, à un système dans le but de le faire planter ou, tout du moins, le faire diverger de son comportement normal. Les données utilisées dans les *fuzz* sont souvent générées aléatoirement.

Le fuzzing comportemental (*behavioural fuzzing*) permet d'injecter des séquences invalides de données valides qui peuvent être générées par le biais d'un modèle, comme dans [Schneider et al., 2012]. Il est aussi possible de générer de telles séquences en modifiant des traces d'exécutions passées pour les injecter dans le système testé. Comme ces tests ne sont pas effectués durant, mais après l'exécution, on parle de fuzzing hors-ligne (*offline fuzzing*) par opposition au fuzzing en-ligne (*online fuzzing*) effectué lors de l'exécution, comme dans [Schneider et al., 2013].

Nous intéressons à la génération de *fuzz* que nous pourrons utiliser pour tester les politiques d'adaptation avec des techniques de fuzzing comportemental hors-ligne.

8.3.2/ MISE EN PLACE ET UTILISATION

En modifiant des traces d'exécution ou par le biais d'un modèle, on peut générer des séquences invalides de données valides permettant la mise en place du fuzzing comportemental.

Nous choisissons de combiner ces deux approches en utilisant notre modèle pour analyser les traces d'exécution afin de pouvoir les considérer comme des séquences de configurations. Bien sûr, ces séquences peuvent être enrichies, si besoin, avec les événements (internes et/ou externes) ayant été extraits de la trace initiale.

En effectuant des opérations de mélange aléatoire, duplication et/ou de suppression, nous pouvons modifier ces séquences pour créer des *fuzz*, que nous appelons fuzzy logs. Chacun de ces fuzzy logs peut alors être utilisé dans une nouvelle exécution pour simuler les entrées (*inputs*) d'un système à composants.

La figure 8.4 illustre ce processus en présentant, de gauche à droite les actions suivantes :

— l'exécution d'un système à composants sur lequel sont appliquées des politiques d'adaptation,
— la collecte de la trace d'exécution,

— la génération des fuzzy logs, et

— l'exécution de ces fuzzy logs.

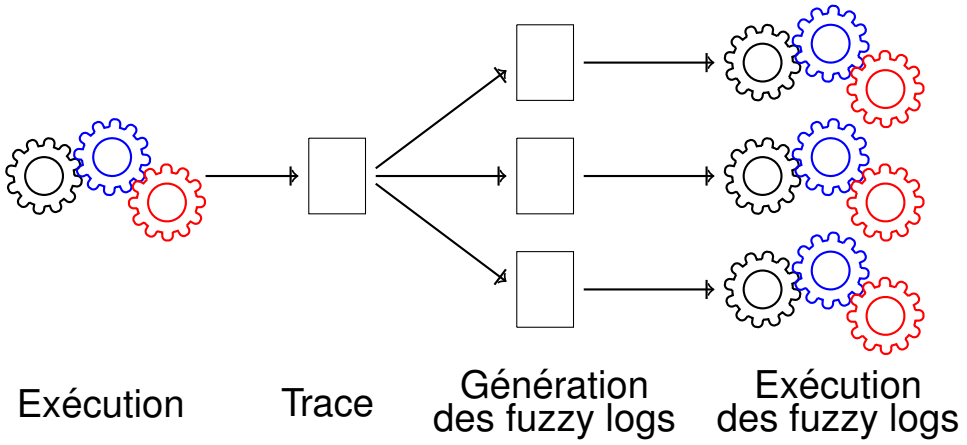


FIGURE 8.4 – Principe de la génération des fuzzy logs

Pour tester une politique d'adaptation spécifique, il suffit de donner un nom unique à chaque reconfiguration déclenchée par la politique d'adaptation que l'on teste. Ainsi, les cas de test ayant une influence sur cette politique peuvent être facilement détectés pour examen ultérieur. Il est aussi possible d'ajouter une politique de réflexion qui stoppe le système (ou prend tout autre action souhaitable) pour chaque succès ou échec d'une reconfiguration déclenchée par la politique d'adaptation testée.

Dans [Weber, 2016], en partant d'une trace d'exécution du système composite **Location** présenté à la section 3.1, nous avons généré et exécuté 1000 fuzzy logs pour tester la politique d'adaptation `cycabgps` de la figure 8.1. Le but de ces tests était de s'assurer que cette politique d'adaptation ne déclenche pas la reconfiguration $addgps$ (qui ajoute le composant **GPS**) lorsque le véhicule se trouve dans une "zone Wi-Fi", où il n'y pas de signal GPS.

Sur les 1000 test effectués, 203 impliquaient la reconfiguration $addgps$; leur analyse a permit l'élaboration de la figure 8.5 où l'axe horizontal représente les indices du chemin d'exécution, l'axe vertical correspond au nombre d'occurrences de la reconfiguration $addgps$ et les lignes verticales symbolisent l'entrée et la sortie de la "zone Wi-Fi".

Les points bleus correspondent au succès de l'application de la reconfiguration $addgps$ et les points rouges à son échec. Ainsi, par exemple, le point bleu situé à l'état 10 signifie que la reconfiguration $addgps$ a été appliquée avec succès à cet état lors de 6 exécutions (sur les 1000 exécutions de fuzzy logs effectuées).

Au delà du succès ou de l'échec de l'application de la reconfiguration $addgps$, ces tests montrent qu'aucune application de cette reconfiguration n'a été tentée pendant que le véhicule était présumé être dans une "zone Wi-Fi". Ceci correspond au comportement attendu de la politique d'adaptation `cycabgps`.

L'utilisation du fuzzing facilite le test et le développement de politiques d'adaptation en permettant au testeur de se concentrer sur les régions des séquences de configurations qui présentent un intérêt.

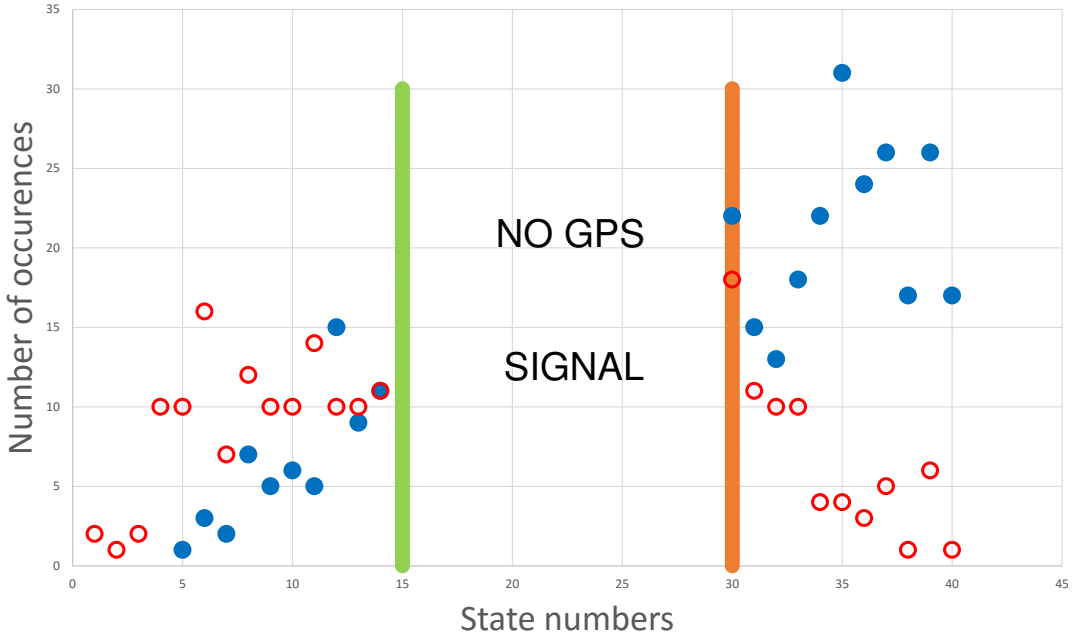


FIGURE 8.5 – Occurrences de la reconfiguration $addgps$

8.4/ CONCLUSION

Dans ce chapitre nous avons, comme dans [Kouchnarenko et al., 2014a], défini des politiques d'adaptation intégrant la logique FTPL. Nous avons aussi proposé une relation de simulation sur les modèles de systèmes à composants reconfigurables et nous avons défini le problème de l'adaptation. En utilisant cette relation, nous avons pu établir que le problème de l'adaptation est semi-décidable et nous avons fourni un algorithme pour l'enforcement des politiques d'adaptation des systèmes à composants reconfigurables. Enfin, nous avons présenté l'usage du fuzzing pour le test de politiques d'adaptation.

IV

VALIDATION EXPÉRIMENTALE

9

STRUCTURE LOGICIELLE POUR RECONFIGURATIONS DYNAMIQUES

Ce chapitre présente l'implémentation que nous avons développée pour les reconfigurations dynamiques de systèmes à composants guidées par des politiques d'adaptation basées sur des schémas temporels. Dans un premier temps, nous décrivons notre implémentation, avec les entités qui la composent et leur fonctionnement, puis nous énonçons un résultat de conformité architecturale. Ensuite, nous détaillons une partie des informations contenues dans les logs générés par notre implémentation et comment nous les utilisons pour en extraire les éléments permettant de produire une interface graphique. Enfin nous présentons l'intégration de la fonctionnalité de fuzzing permettant de tester des politiques d'adaptation.

9.1/ IMPLÉMENTATION

Cette section décrit l'implémentation en Java développée pour les reconfigurations dynamiques de systèmes à composants guidées par des politiques d'adaptation basées sur des schémas temporels. Nous avons déployé un prototype contenu dans un package Java appelé cbsdr (pour *Component-Based System Dynamic Reconfiguration*). Ceci nous permet d'appliquer des politiques d'adaptation à des systèmes à composants développés en utilisant les modèles de systèmes à composants Fractal et FraSCAti qui sont encapsulés et contrôlés par notre implémentation. En plus d'un système à compo-

Sommaire

sants devant être guidé et contrôlé, notre application utilise les informations suivantes fournies via des fichiers de configuration :

— des noms de reconfigurations accompagnés de leurs définitions,
— des politiques d'adaptation (comme, par exemple, la politique d'adaptation `cycabgps` de la figure 8.1), et
— des propriétés temporelles à préserver, que l'on divise en deux catégories :
  — les propriétés temporelles ne devant pas être violées du fait de l'application de reconfigurations déclenchées par des politiques d'adaptation, et
  — les propriétés temporelles dont la violation déclenche une action corrective via le mécanisme de réflexion à l'exécution.

Nous allons d'abord présenter le fonctionnement global de notre implémentation en décrivant chacune des entités qui la composent, puis nous expliciterons les interactions entre ces entités avant de détailler le fonctionnement de chacun des contrôleurs de *cbsdr*. Enfin, en nous appuyant sur les résultats du chapitre 4, nous établirons la conformité architecturale de notre implémentation.

9.1.1/ DESCRIPTION DE NOTRE IMPLÉMENTATION

Comme nous l'avons représenté sur la figure 9.1, l'implémentation de *cbsdr* utilise trois contrôleurs : *a*) le contrôleur d'événements (**EC** pour *Event Controller*) qui reçoit les événements et les stocke jusqu'à ce qu'ils soient demandés via la requête d'une autre entité, *b*) le contrôleur de réflexion (**RC** pour *Reflection Controller*) qui envoie un événement au contrôleur d'événements (**EC**) quand une propriété temporelle devant être préservée via le mécanisme de réflexion à l'exécution est violée, et *c*) le contrôleur de politiques d'adaptation (**APC** pour *Adaptation Policy Controller*) qui gère les reconfigurations ainsi que les mécanismes d'adaptation et d'enforcement à l'exécution en utilisant l'algorithme `AdaptEnfor` représenté sous la figure 8.2.

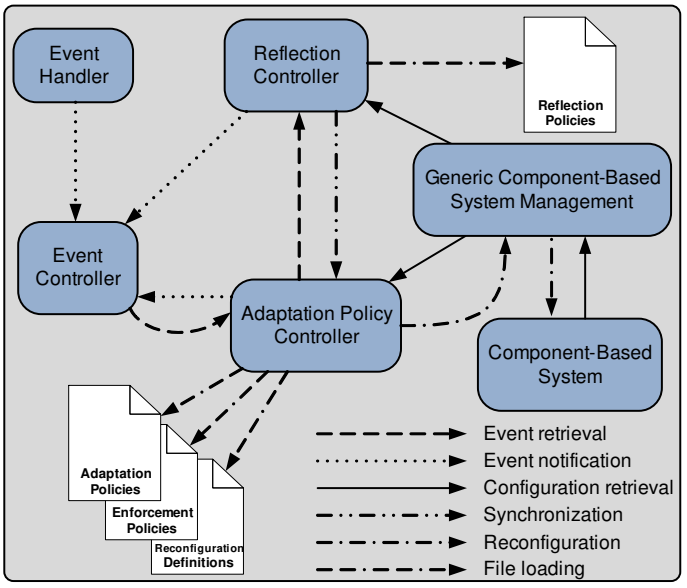


FIGURE 9.1 – Architecture de *cbsdr*

Le contrôleur de réflexion (**RC**) et le contrôleur de politiques d'adaptation (**APC**) sont tous deux basés sur le modèle de la boucle de contrôle (*control loop* [Dobson et al., 2006, Cheng et al., 2009, de Lemos et al., 2013]). La section 9.1.3 ci-dessous fournit davantage d'informations sur le fonctionnement de ces contrôleurs.

Le contrôleur d'événements (**EC**) fonctionne de paire avec un gestionnaire d'événements (**EH** pour *Event Handler*) qui est utilisé pour recevoir des événements provenant d'une source externe afin de les transmettre au contrôleur d'événements (**EC**) .

Notons que l'entité représentant le système à composants (**CBS** pour *Component−Based System*), est un véritable système à composants (et non pas un modèle) utilisant le modèle de systèmes à composants Fractal ou FraSCAti. Les interactions avec le système à composants (**CBS**) que nous souhaitons guider et contrôler par le biais de politiques d'adaptation se font exclusivement via le gestionnaire de systèmes à composants générique (**GCBSM** pour *Generic Component−Based System Management*) qui est un ensemble de classes développées de telle façon qu'elles puissent être invoquées indépendamment de l'implémentation du modèle de systèmes à composants (Fractal ou FraSCAti) utilisé par le système à composants (**CBS**).

Pour s'assurer que notre implémentation soit indépendante d'un modèle de systèmes à composants particulier, seules quelques classes du gestionnaire de systèmes à composants générique (**GCBSM**) utilisent des instructions relatives aux API spécifiques aux modèles supportés (Fractal et FraSCAti). Ainsi les contrôleurs de la figure 9.1 sont de simples objets Java qui ne sont basés ni sur Fractal, ni sur FraSCAti.

9.1.2/ INTERACTIONS ENTRE LES ENTITÉS DE NOTRE IMPLÉMENTATION

On voit sur la figure 9.1 six différents types d'interactions entre les entités de *cbsdr*. Il s'agit *a*) de la lecture d'événements (**er** pour *Event retrieval*), *b*) de la notification d'événements (**en** pour *Event notification*), *c*) de la lecture d'une configuration (**cr** pour *Configuration retrieval*), *d*) de la synchronisation entre entités (**s** pour *Synchronization*), *e*) de l'application d'une opération de reconfiguration (**r** pour *Reconfiguration*), et *f*) du chargement d'un fichier (**fl** pour *File loading*).

Lecture d'événements (er). Cette interaction est symbolisée par une flèche discontinue en tirets (- - ->). On voit, sur la figure 9.1, que le contrôleur de politiques d'adaptation (**APC**) lit les événements depuis le contrôleur d'événements (**EC**). On parle aussi de retrait d'événements car lorsque des événements sont récupérés par un contrôleur depuis le contrôleur d'événements (**EC**), ceux-ci ne seront plus disponibles lors de la lecture suivante demandée par le même contrôleur. On voit aussi que des événements sont retirés par le contrôleur de réflexion (**RC**) depuis le contrôleur de politiques d'adaptation (**APC**), ce retrait particulier est lié à la synchronisation des deux contrôleurs et sera explicité dans le paragraphe ci-dessous consacré à la synchronisation entre entités (**s**).

Notification d'événements (en). Cette interaction est symbolisée par une flèche discontinue en points (· · · >) On voit, sur la figure 9.1 que le contrôleur d'événements (**EC**) est notifié de l'occurrence d'événements par *a*) le gestionnaire d'événements (**EH**) qui transmet les événements qu'il a reçu en provenance de sources externes, *b*) le contrôleur de réflexion (**RC**) qui génère un événement lorsqu'une propriété temporelle dont la violation doit déclencher une action corrective via le mécanisme de réflexion à l'exécution n'est plus préservée, *c*) le contrôleur de politiques d'adaptation (**APC**) qui envoie l'événement "*ope* **normal**" ou "*ope* **exceptional**" selon le résultat de l'opération reconfiguration *ope*. Une fois les événements reçus par le contrôleur d'événements (**EC**) ceux-ci sont stockés jusqu'à leur retrait comme décrit dans le paragraphe précédent.

Lecture d'une configuration (cr). Cette interaction est symbolisée par une flèche continue (——>). Lorsque le contrôleur de réflexion (**RC**) ou le contrôleur de politiques d'adaptation (**APC**) ont besoin de connaitre la configuration courante du système à composants (**CBS**) ils envoient une requête de lecture au gestionnaire de système à composants générique (**GCBSM**). Ce dernier va alors récupérer la configuration courante du système à composants (**CBS**) en utilisant des instructions spécifiques au modèle (Fractal ou FraSCAti) utilisé par le système à composants (**CBS**). Enfin, une fois la configuration courante du système à composants (**CBS**) récupérée, le gestionnaire de système à composants générique (**GCBSM**) peut la transmettre, en utilisant notre sémantique opérationnelle, au contrôleur l'ayant requise.

Synchronisation (s). Cette interaction est symbolisée par une flèche discontinue alternant un tiret et deux points (– · · – · · – >). On voit, sur la figure 9.1 que le contrôleur de réflexion (**RC**) est synchronisé avec le contrôleur de politiques d'adaptation (**APC**). Cela signifie que ces contrôleurs utilisent la même fréquence pour leurs boucles de contrôle. Ainsi, comme leurs boucles de contrôle sont synchronisées, le contrôleur de politiques d'adaptation (**APC**) transmet, lors de leur synchronisation, au contrôleur de réflexion (**RC**) les événements qu'il a récupérés auprès du contrôleur d'événements (**EC**). Ceci explique pourquoi le contrôleur de réflexion (**RC**) ne lit pas d'événements directement depuis le contrôleur d'événements (**EC**). Plus d'informations sur la synchronisation des contrôleurs sont données à la section 9.1.3.

Application d'une opération de reconfiguration (r). Cette interaction est symbolisée par une flèche discontinue alternant tirets et points (– · – · – >) Lorsque le contrôleur de politiques d'adaptation (**APC**) initie une reconfiguration, il contacte le gestionnaire de système à composants générique (**GCBSM**). Celui-ci va alors appliquer la reconfiguration au système à composants (**CBS**) en utilisant des instructions spécifiques au modèle (Fractal ou FraSCAti) utilisé par le système à composants (**CBS**).

Chargement de fichier (fl). Cette interaction est symbolisée par une flèche discontinue alternant deux tirets et un point (– – · – – · ––>). Le contrôleur de réflexion (**RC**) a besoin d'au moins une liste de propriétés FTPL dont il doit vérifier la préservation à l'exécution. Chacune de ces listes constitue une politique de réflexion contenue dans un fichier différent. Elles sont représentées dans la figure 9.1 sous le nom de *Reflection Policies*. De la même façon que le contrôleur de réflexion (**RC**) utilise des politiques de réflexion, le contrôleur de politiques d'adaptation (**APC**) utilise des politiques d'enforcement (*Enforcement Policies* sur la figure 9.1) qui ont une forme similaire. En outre ce contrôleur utilise aussi des politiques d'adaptation (comme, par exemple, la politique d'adaptation `cycabgps` de la figure 8.1) et les définitions (en termes de reconfigurations primitives) des reconfigurations (non-primitives) utilisées dans les politiques d'adaptation.

9.1.3/ FONCTIONNEMENT DES CONTRÔLEURS DE NOTRE IMPLÉMENTATION

Au delà de la synchronisation (**s**) entre le contrôleur de réflexion (**RC**) et le contrôleur de politiques d'adaptation (**APC**) la façon dont les événements sont gérés permet à notre implémentation de fonctionner suivant l'hypothèse de synchronisation parfaite (*perfect*

synchrony hypothesis [Jantsch, 2004]). C'est-à-dire que les événements sont envoyés et reçus à des instances de temps discrètes qui sont représentées par des identifiants $t \in \mathbb{N}^{\geq 0}$.

Dans la pratique, le contrôleur d'événements (**EC**) est la première des entités de notre implémentation à être instanciée. Il est invoqué avec un paramètre correspondant à un objet gestionnaire d'événements (**EH**). La seule contrainte du gestionnaire d'événements est d'exposer des méthodes permettant au contrôleur d'événements (**EC**) de retirer les événements externes (e.g., par des capteurs) qui lui sont transmis.

Il est tout à fait possible de faire en sorte que le gestionnaire d'événements puisse gérer des événements provenant de plusieurs sources externes. La seule contrainte pour mettre en place un gestionnaire d'événements particulier est d'implémenter l'interface Java $cbsdr.reconf.EventHandler$. Dans la pratique le gestionnaire d'événements utilisé est le $TouchFileEventHandler$ qui, dès qu'un fichier est créé (e.g., via la commande UNIX $touch$, d'où le nom de ce gestionnaire d'événements) dans un répertoire prédéfini, va détecter un événement portant le nom du fichier. Après détection par le gestionnaire d'événements $TouchFileEventHandler$ le fichier en question est effacé. Ainsi toute application ayant la possibilité d'écrire dans le répertoire "surveillé" par le gestionnaire d'événements $TouchFileEventHandler$ peut être considérée comme une source d'événements externe.

Le contrôleur d'événements (**EC**) agit donc comme un collecteur d'événements en centralisant les événements qu'il reçoit du gestionnaire d'événements (**EH**), du contrôleur de réflexion (**RC**) et du contrôleur de politiques d'adaptation (**APC**). Lorsqu'un contrôleur contacte le contrôleur d'événements (**EC**), ce dernier lui fournit une liste d'événements ; par la suite, ces événements ne seront plus disponibles lors de la lecture suivante demandée par le même contrôleur.

Le contrôleur de politiques d'adaptation (**APC**) constitue le cœur de notre implémentation. Lors de son instanciation les éléments suivants doivent être fournis.

— Un ou plusieurs fichiers contenant les politiques d'adaptation devant être appliquées,

— un ou plusieurs fichiers contenant les définitions (en termes de reconfigurations primitives) des reconfigurations non-primitives utilisées dans les politiques d'adaptation,

— un ou plusieurs fichiers contenant les politiques d'enforcement qui sont les propriétés FTPL ne devant pas être violées du fait de l'application d'une reconfiguration déclenchée par les politiques d'adaptation, et

— un objet "système à composants générique" correspondant à un système à composants (**CBS**) géré par le gestionnaire de système à composants générique (**GCBSM**).

Comme précisé dans le modèle de boucle de contrôle [Dobson et al., 2006, Cheng et al., 2009, de Lemos et al., 2013]), le contrôleur de politiques d'adaptation (**APC**) collecte, analyse, décide et agit comme décrit ci-dessous.

Collecte : Les événements internes et externes sont récupérés depuis contrôleur d'événements (**EC**) et la configuration courante du système à composants (**CBS**) est obtenue via le gestionnaire de système à composants générique (**GCBSM**).

Analyse : Les politiques d'adaptation sont évaluées et les reconfigurations suggérées (s'il y en a) sont ordonnées par priorité.

Décision : Ces reconfigurations sont appliquées au modèle du système à composants (ligne 15 de l'algorithme `AdaptEnfor` présenté à la figure 8.2) en commençant par celles ayant la plus grande priorité.

Action : La première de ces reconfigurations amenant le système à ne violer aucune des propriétés contenues dans les politiques d'enforcement est appliquée (ligne 16 et 17 de l'algorithme `AdaptEnfor`) au système à composants (**CBS**) via le gestionnaire de systèmes à composants générique (**GCBSM**) en appliquant un processus similaire à celui décrit dans [Boyer et al., 2013]. Si l'opération de reconfiguration ope finit normalement, l'événement ope **normal** (ligne 20) est envoyé au contrôleur d'événements (**EC**). Si l'opération de reconfiguration ope finit avec une erreur, la configuration initiale (obtenue lors de la phase de collection) est ré-appliquée et l'événement ope **exceptional**" (ligne 26) est envoyé au contrôleur d'événements (**EC**).

On notera que pour éviter d'appliquer des reconfigurations ayant une faible utilité, on décide de ne considérer que les reconfigurations ayant une priorité supérieure à une certaine valeur prédéterminée. En effet, l'utilité d'une reconfiguration donnée est mesurée par une valeur floue [Pedrycz, 1993] d'un type flou (voir définition 64). Comme des opérations de reconfigurations provenant de différentes politiques d'adaptation peuvent utiliser des types flous différents, nous convertissons l'utilité qui est une valeur floue en une priorité qui correspond à un nombre réel (type primitif $double$ en Java) compris entre 0 et 1. Ainsi, nous pouvons comparer les priorités de reconfigurations provenant de différentes politiques d'adaptation. De plus, en ne considérant que des reconfigurations ayant une priorité supérieure à une certaine valeur (0.66 par défaut) nous prévenons le choix de reconfigurations d'utilité faible par nos politiques d'adaptation.

Le contrôleur de réflexion (**RC**) fonctionne de façon similaire. Lors de son instanciation les éléments suivants doivent être fournis.

- Un ou plusieurs fichiers contenant les politiques de réflexion qui sont les propriétés FTPL dont le non respect déclenche une action corrective,
- un objet contrôleur :
  - si le contrôleur de réflexion (**RC**) doit être synchronisé avec le contrôleur de politiques d'adaptation (**APC**), cet objet correspond (comme dans le cas de la figure 9.1) au contrôleur de politiques d'adaptation (**APC**),
  - sinon, un objet correspondant au contrôleur d'événements (**EC**), et
- un objet "système à composants générique" correspondant à un système à composants (**CBS**) géré par le gestionnaire de système à composants générique (**GCBSM**).

Comme dans notre cas (figure 9.1) le contrôleur de réflexion (**RC**) est synchronisé avec le contrôleur de politiques d'adaptation (**APC**), les boucles de contrôle des deux contrôleurs s'effectuent simultanément et le contrôleur de réflexion (**RC**) récupère les événements directement depuis le contrôleur de politiques d'adaptation (**APC**). Ceci permet une analyse simplifiée des logs applicatifs du fait que les indices utilisés pour numéroter les états correspondant aux configurations de notre modèle de système à composants sont les mêmes pour les deux contrôleurs. De plus, pour chacun de ces états, les deux contrôleurs utilisent le même ensemble d'événements.

Il est bien entendu possible de configurer le contrôleur de réflexion (**RC**) sans synchronisation ; dans ce cas il faudrait spécifier en paramètre un contrôleur d'événements (**EC**).

De plus les boucles de contrôle des deux contrôleurs (**APC** et **RC**) seraient totalement indépendantes et pourraient même utiliser une fréquence différente.

Lorsque la violation d'une propriété contenue dans une des politiques de réflexion est détectée par le contrôleur de réflexion (**RC**), celui-ci envoie un événement spécifié dans la politique de réflexion au contrôleur d'événements (**EC**). Lorsque cet événement parvient au contrôleur de politiques d'adaptation (**APC**), une action corrective spécifiée comme une opération de reconfiguration dans une politique d'adaptation peut être effectuée.

9.1.4/ CONFORMITÉ ARCHITECTURALE

Nous avons établi précédemment que le modèle de reconfiguration du chapitre 3 est une approximation correcte d'un modèle interprété, défini à la section 4.4, correspondant à une implémentation. Ceci peut s'exprimer en utilisant la notion de conformité (*conformance*). En considérant ioco [Tretmans, 1996], la relation de conformité la plus utilisée couramment, une implémentation $S_{\mathcal{I}}$ est conforme à sa spécification S, si *a*) d'après une trace de S, on peut prévoir le résultat d'une trace de $S_{\mathcal{I}}$ dans S, et si *b*) l'implémentation est autorisée à atteindre un état où aucun résultat n'est produit seulement si c'est aussi le cas de la spécification.

L'utilisation de diverses relations de simulation permet d'exprimer la conformité relative à l'inclusion de traces ainsi que des relations de conformité plus fortes au niveau des systèmes de transitions. Ainsi, en utilisant les mêmes arguments que ceux développés dans la démonstration du théorème 1 et la conformité sur l'inclusion de traces modulo τ, nous obtenons le résultat de conformité suivant avec $S_{\mathcal{I}_{cbsdr}}$ représentant l'implémentation $cbsdr$.

Proposition 9 : Conformité de l'implémentation $cbsdr$.

$S_{\mathcal{I}_{cbsdr}}$ est conforme à S.

Démonstration. Soit $S = \langle C, C^0, \mathcal{R}_{run}, \mapsto, l \rangle$ un système à composants reconfigurable et $\mathcal{R}_{int}$ l'ensemble des opérations internes induites par l'implémentation $cbsdr$ et représentées par τ. Considérons le modèle de reconfigurations interprété de la définition 38 où l'interprétation de la sémantique opérationnelle des systèmes à composants est définie par le système de transitions doublement étiqueté $S_{\mathcal{I}_{cbsdr}} = \langle C_{\mathcal{I}}, C^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ où $C_{\mathcal{I}}$ est un ensemble de configurations avec les états de leur mémoire, $C^0_{\mathcal{I}}$ est l'ensemble des configurations initiales, $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}\}$, $\rightarrow_{\mathcal{I}} \subseteq C_{\mathcal{I}} \times \mathcal{R}_{run_{\mathcal{I}}} \times C_{\mathcal{I}}$ est la fonction de reconfiguration interprétée et $l_{\mathcal{I}} : C_{\mathcal{I}} \rightarrow CP$ est une fonction totale d'interprétation.

Comme dans la section 4.4.1, nous considérons un ensemble (infini en général) $GM = \{u, \ldots\}$ d'états de mémoire partagés et un ensemble (infini en général) $LM = \{v, \ldots\}$ d'états de mémoire étant chacun local à un composant donné. Nous utilisons aussi le fait que pour toute reconfiguration primitive ope de S, la reconfiguration interprétée correspondante $\overline{ope}$ de $S_{\mathcal{I}_{cbsdr}}$ dispose de pré-conditions plus fortes ou équivalentes de façon que toute construction de reconfiguration gardée se comporte de façon déterminisme.

Considérons $ope \in \mathcal{R}_{run} \cup \mathcal{R}_{int}$. Comme la couverture par τ des opérations de $\mathcal{R}_{int}$ sont introduites pour évaluer les gardes de configurations gardées, elle ne forment pas de cycle

infini de τ-transitions ce qui implique qu'il y a toujours une fin au cycle de τ-transitions pour laisser la place soit à la transition $\overline{ope}$, soit à la transition suivante correspondant à $ope' \in \mathcal{R}_{run} \cup \mathcal{R}_{int}$.

Par construction, toute opération de reconfiguration primitive du modèle interprété a des pré-conditions plus fortes ou équivalentes que son homologue du modèle non interprété. Ainsi, en utilisant les hypothèses sur les pré-conditions comme dans [Dijkstra, 1975], nous pouvons établir que les reconfigurations gardées composées de déclarations primitives (voir page 74) comme $G \rightarrow \overline{s}$, avec $\overline{s} \in \mathcal{R}_{run_I} \backslash \mathcal{R}_{int}$ ont des pré-conditions plus fortes ou équivalentes que leurs déclarations primitives homologues $s \in \mathcal{R}_{run}$

En conséquence, en considérant une configuration (initiale pour la première opération de reconfiguration considéré) $c_1 \in C_I$, il y a une configuration correspondante $c_2 \in C$ telle que si une reconfiguration gardée $G \rightarrow \overline{s}$ est appliquée à c_1, il existe une garde G', telle que $G \Rightarrow G'$ et $G' \rightarrow s$ puisse être appliquée à c_2. De plus, les configurations cibles c'_1 et c'_2 respectivement obtenues en appliquant $G \rightarrow \overline{s}$ et $G' \rightarrow s$ à c_1 et c_2 sont conformes entre-elles (au sens de la conformité sur l'inclusion de traces modulo τ) ; car soit il existe une opération ope pouvant être appliqué (dans ce cas on reprend le même raisonnement), soit une telle opération n'existe pas (cas traité ci-dessous).

Si aucune opération ope ne peut être appliquée à $c_1 \in C_I$ (après d'éventuelles exécutions de τ), il en est de même pour $c_2 \in C$. □

9.2/ FICHIERS DE LOG ET INTERFACE GRAPHIQUE

Le résultat d'une exécution de notre implémentation peut être déduit en examinant les fichiers de log produits. Nous utilisons aussi ces fichiers de log afin d'obtenir les informations nécessaires à la production d'une interface graphique. Dans cette section, nous décrivons le contenu du fichier de log du contrôleur de politiques d'adaptation (**APC**) et nous présentons l'interface graphique basée sur les informations contenues dans ce fichier.

9.2.1/ FICHIER DE LOG DU CONTRÔLEUR DE POLITIQUES D'ADAPTATION (**APC**)

Chaque contrôleur de notre implémentation produit des fichiers de log. Dans cette section, nous nous intéressons plus particulièrement au fichier de log produit par le contrôleur de politiques d'adaptation (**APC**) car c'est celui qui contient le plus d'informations utiles. Ce fichier contient, au début, l'état du chargement des fichiers de configuration, puis des informations sur tous les états dans l'ordre d'apparition. Pour la suite nous désignerons ce fichier sous le nom de log **APC**.

La figure 9.2 montre une section du fichier log **APC** contenant le détail du chargement du fichier de définitions des reconfigurations utilisées par les politiques d'adaptation du système composite **Location** présenté à la section 3.1. On voit que cette section est encapsulée par les mots clés "Begin" et "End". Au début de cette section (seconde ligne), on précise quelles classes Java correspondent aux implémentations respectives des composants **Wi-Fi** et **GPS**, puis chaque ligne suivante contient un nom de reconfiguration non-primitive et sa définition en termes de reconfigurations primitives. On notera que le format des fichiers de logs initialement de la forme "`1475997083138 - Sun Oct 09`

09:11:23 CEST 2016 - Begin loading of ...” a été modifié par soucis de lisibilité.

```
Begin loading of Reconfiguration file /home/jfwm/cycab/resources/AdaptationPolicies/cycab.reconf
Context loaded:{wifi=cycab.WifiImpl, gps=cycab.GpsImpl}
Reconfiguration loaded:removegps[location:stop, merger:unbind:getGpsPosition, location:remove:gps, location:start]
Reconfiguration loaded:addgps[location:add:gps, merger:bind:getGpsPosition:gps:gpsPosition, gps:start]
Reconfiguration loaded:removewifi[location:stop, merger:unbind:getWifiPosition, location:remove:wifi, location:start]
Reconfiguration loaded:addwifi[location:add:wifi, merger:bind:getWifiPosition:wifi:wifiPosition, wifi:start]
Reconfiguration loaded:chargeBattery[controller:updateParameter:int:Power:100]
Reconfiguration loaded:stopCycab[location:stop, controller:stop, controller:updateParameter:int:Power:0]
End of loading of Reconfiguration file /home/jfwm/cycab/resources/AdaptationPolicies/cycab.reconf - Duration: 19 ms.
```

FIGURE 9.2 – Fichier de log **APC** : chargement d'un fichier de reconfigurations

La figure 9.3 montre la partie du fichier log **APC** contenant le chargement du fichier `gps.eap` qui contient la politique d'adaptation `cycabgps` de la figure 8.1. Comme précédemment, cette section est encapsulée par les mots clés “Begin” et “End”. On voit que les événements externes sont d'abord identifiés, puis chaque règle de reconfiguration est chargée avec un nom auto-généré (e.g., “cycabgps_0” ou “cycabgps_1”).

```
Begin loading of Policy file /home/jfwm/cycab/resources/AdaptationPolicies/gps.eap
Successful Loading of event: entry
Successful Loading of event: exit
Successful Loading of event: start
Successful Loading of: Reconfiguration rule cycabgps_0:
when after start,exit ((always (Power<200)) until entry) = P_TRUE4
if gps in Components = FALSE
then addgps -> low
Successful Loading of: Reconfiguration rule cycabgps_1:
when Power<33 = TRUE4
if gps in Components = FALSE
then addgps -> low
Successful Loading of: Reconfiguration rule cycabgps_2:
when Power<33 = FALSE4
if gps in Components = FALSE
then addgps -> high
Successful Loading of: Reconfiguration rule cycabgps_3:
when Power<33 = FALSE4
if gps in Components and wifi in Components = TRUE
then removegps -> low
Successful Loading of: Reconfiguration rule cycabgps_4:
when after start,exit ((always (Power<200)) until entry) = P_TRUE4
if gps in Components and wifi in Components = TRUE
then removegps -> high
Successful Loading of: Reconfiguration rule cycabgps_5:
when Power<33 = TRUE4
if gps in Components and wifi in Components = TRUE
then removegps -> high
End of loading of Policy file /home/jfwm/cycab/resources/AdaptationPolicies/gps.eap - Duration: 16 ms.
```

FIGURE 9.3 – Fichier de log **APC** : chargement de la politique d'adaptation `cycabgps`

Une fois que tous les fichiers nécessaires au fonctionnement du contrôleur de politiques d'adaptation (**APC**) ont été chargés, celui-ci peut commencer sa boucle de contrôle. A chaque itération, la configuration du système à composants (**CBS**) géré par le gestionnaire de système à composants générique (**GCBSM**) est lue et est écrite dans le log **APC**, comme illustré figure 9.4.

En plus de ces informations, le détail de l'évaluation des diverses propriétés FTPL utilisées par le contrôleur de politiques d'adaptation (**APC**) peut aussi être affiché. Le niveau de détail peut être ajusté en paramétrant un niveau de débogage (*debug level*) lors de l'invocation du contrôleur.

9.2.2/ INTERFACE GRAPHIQUE

La figure 9.5, ci-dessous, montre l'interface graphique de *cbsdr* qui affiche, dans la partie supérieure un état du système composite **Location** présenté à la section 3.1. La partie de

```
Components     = {controller, gps, location, merger, wifi}
IProvided      = {gpsPosition, mergePosition, position, securePosition, wifiPosition}
IRequired      = {getGpsPosition, getMergePosition, getWifiPosition}
Parameters     = {GpsPowerUsage, Power, Trust, WifiPowerUsage}
ITypes         = {MergePosition, Position, SecurePosition}
PTypes         = {int}
InterfaceType  = {getGpsPosition->Position, getMergePosition->MergePosition, getWifiPosition->Position, gpsPosition->
      Position, mergePosition->MergePosition, position->SecurePosition, securePosition->SecurePosition, wifiPosition->
      Position}
Provider       = {gpsPosition->gps, mergePosition->merger, position->controller, securePosition->location, wifiPosition
    ->wifi}
Requirer       = {getGpsPosition->merger, getMergePosition->controller, getWifiPosition->merger}
Contingency    = {getGpsPosition->optional, getMergePosition->mandatory, getWifiPosition->optional}
ParameterType  = {GpsPowerUsage->int, Power->int, Trust->int, WifiPowerUsage->int}
Definer        = {GpsPowerUsage->merger, Power->controller, Trust->merger, WifiPowerUsage->merger}
Parent         = {(controller, location), (gps, location), (merger, location), (wifi, location)}
Binding        = {getGpsPosition->gpsPosition, getMergePosition->mergePosition, getWifiPosition->wifiPosition}
Delegate       = {position->securePosition}
State          = {controller->started, gps->started, location->started, merger->started, wifi->started}
Value          = {GpsPowerUsage->3, Power->100, Trust->0, WifiPowerUsage->2}
```

FIGURE 9.4 – Fichier de log **APC** : affichage d'un état du système composite **Location**

gauche permet de choisir la configuration à afficher parmi les différent états correspondant à une exécution du système, tandis que la partie inférieure peut-être utilisée pour afficher diverses informations comme l'évolution de paramètres du modèle, la console Java ou encore le résultat des reconfigurations effectuées.

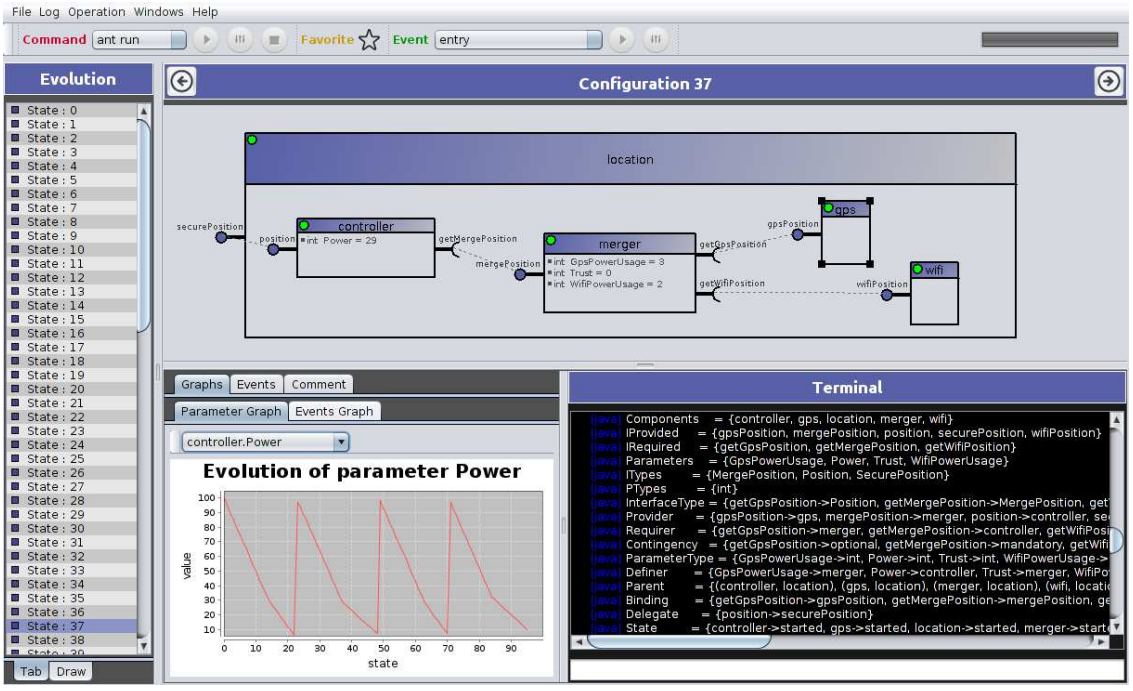


FIGURE 9.5 – Modèle du système composite **Location** affiché dans notre interface

Cette interface graphique utilise les logs produits par le contrôleur de politiques d'adaptation (**APC**) pour générer les éléments à afficher. Ainsi la lecture dans le log **APC** (voir figure 9.4) de la configuration du système à composants (**CBS**) géré par le gestionnaire de système à composants générique (**GCBSM**) permet de générer l'affichage de la partie supérieure de notre interface, mais permet aussi d'avoir des informations sur l'évolution de la valeur des paramètres du système que l'on étudie.

Nous pouvons donc utiliser notre interface pour analyser une exécution après qu'elle se soit finie, mais il est aussi possible d'afficher les informations concernant une exécution

en temps quasi-réel en lisant les logs au fur et à mesure qu'ils sont produits.

Dans le cas où l'on utilise cette interface pour afficher les états d'une exécution en cours, on peut générer un événement externe en le sélectionnant dans une liste déroulante créée à partir des événements identifiés dans le log **APC** (voir figure 9.3). Il suffit alors d'utiliser un gestionnaire d'événements (**EH**) supporté par l'interface, comme par exemple le $TouchFileEventHandler$ pour pouvoir créer des événements externes d'un simple clic.

9.3/ IMPLÉMENTATION DU FUZZING

Notre implémentation utilise des contrôleurs basés sur des boucles de contrôle qui collectent les évolutions de la configuration du système à composants (**CBS**) géré par le gestionnaire de système à composants générique (**GCBSM**). La séquence des reconfigurations collectées durant une exécution constitue une trace qui peut être modifiée soit manuellement, pour la génération de tests très spécifiques, soit automatiquement pour la génération en masse de tests utilisant le mélange aléatoire, la duplication et/ou la suppression de configurations. De tels tests (nommés fuzzy logs) sont obtenus par des transformations pouvant être automatisées en utilisant un sous-package de $cbsdr$ nommé $cbsdr.fuzzy$ qui implémente ce que nous appelons le Fuzzy Engine.

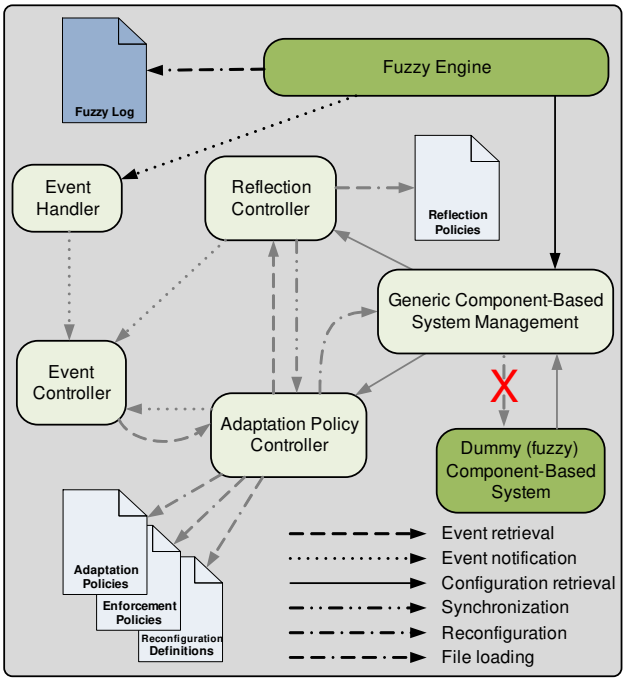


FIGURE 9.6 – Architecture du Fuzzy Engine

Le Fuzzy Engine est intégré dans le développement de $cbsdr$ comme le décrit la figure 9.6. Les couleurs claires représentent les entités faisant partie du développement précédent tandis que les couleurs foncées sont utilisées pour désigner les ajouts réalisés pour l'intégration du Fuzzy Engine.

Le gestionnaire de système à composants générique (**GCBSM**) est principalement composé de classes abstraites génériques qui héritent d'autres classes qui ont été spécialement développées pour la gestion de modèles de système à composants particuliers comme Fractal ou FraSCAti. Nous avons tout simplement ajouté le support d'un nouveau modèle de système à composants que nous avons appelé dummy et qui ne contient aucune information. Ainsi, à chaque fois que le contrôleur de politiques d'adaptation (**APC**) ou le contrôleur de réflexion (**RC**) envoient une requête de lecture au gestionnaire de système à composants générique (**GCBSM**), ce dernier voyant que le système à composants (**CBS**) correspond au modèle "dummy" va requérir la configuration correspondante auprès du Fuzzy Engine.

Bien sur, le Fuzzy Engine doit au préalable avoir été initialisé avec un fuzzy log correspond au test que l'on souhaite effectuer. Afin de pouvoir générer des fuzzy logs à partir

de la trace d'une exécution, nous écrivons dans un fichier, lors d'une exécution donnée, toutes les information échangées entre notre implémentation et le système à composants (**CBS**) auquel nous appliquons des politiques d'adaptation. Comme ces interactions se font toutes via le gestionnaire de système à composants générique (**GCBSM**), c'est cette entité qui va les écrire dans un fichier que nous appellerons la trace longue.

Chaque information écrite dans le fichier de trace longue est *a*) soit une entrée (notée `[ INPUT ]` dans le fichier), c'est-à-dire une requête de la configuration courante du système à composants (**CBS**) ou une directive de reconfiguration, *b*) soit une sortie (notée `[OUTPUT ]`), comme le détail de la configuration courante du système à composants (**CBS**) ou la confirmation qu'une directive de reconfiguration a été appliquée, *c*) soit une information (notée `[ INFO ]`) indiquant, par exemple le début ou la fin d'un état correspondant à une itération de la boucle de contrôle du contrôleur de politiques d'adaptation (**APC**) ou du contrôleur de réflexion (**RC**).

Les entrées et sorties sont écrites dans le fichier de trace longue par le gestionnaire de système à composants générique (**GCBSM**), tandis que les informations sont écrites par le contrôleur de politiques d'adaptation (**APC**) et le contrôleur de réflexion (**RC**). Notons au passage que pour chaque entrée et chaque sortie, la trace longue contient la localisation du code Java qui a initié cette entrée. Cette information est déterminée au sein du gestionnaire de système à composants générique (**GCBSM**) par une analyse de la pile d'exécution ("stacktrace").

Par exemple, si une instruction à la ligne 82 du code du contrôleur de politiques d'adaptation (**APC**) requiert la configuration courante du système à composants (**CBS**), la ligne suivante sera ajoutée à la trace longue : "`[ INPUT ] - 1453989807768 - configuration requested [AdaptationPolicyController:82]`". On voit que cette ligne contient une entrée, effectuée au temps estampillé par le nombre 1453989807768, qui consiste en une requête de configuration initiée depuis une instruction correspondant à la ligne 82 du fichier Java définissant la classe $AdaptationPolicyController$.

Le fait de spécifier dans la trace longue la localisation des instructions à l'origine des entrées et sorties permet de savoir quelle entrée (e.g., une requête de la configuration courante) correspondant à quelle sortie (e.g., le détail d'une configuration). Ceci est particulièrement important du fait que les entrées et sorties sont effectuées de manière asynchrone. En effet, le contrôleur de politiques d'adaptation (**APC**) peut faire une requête de la configuration un peu avant le contrôleur de réflexion (**RC**) mais les réponses du système à composants (**CBS**) peuvent être retournées par le gestionnaire de système à composants générique (**GCBSM**) dans l'ordre inverse.

Comme le fichier contenant la trace longue peut être difficile à manipuler nous effectuons d'abord une transformation de cette trace afin d'en simplifier la gestion. Nous partons du constat que lors d'une exécution l'architecture d'un système à composants donné change beaucoup moins souvent que la valeur de ses paramètres. C'est pourquoi nous stockons dans un fichier les définitions des différentes typologies de configuration sous formes de classes d'équivalences, en considérant que deux configurations sont équivalentes si tous les ensembles de $Elem$ et Rel de la définition 27 sont identiques excepté l'ensemble concernant la relation $Value$ représentant les valeurs des paramètres.

Ainsi, la configuration de la figure 9.4 permet de définir la classe `cycabComplete` affichée figure 9.7. Le fichier contenant les définitions est généré automatiquement avec des noms de classes de reconfigurations auto-générés (e.g., `location_1`, `location_2`, etc.) qui peuvent être changés pour être plus expressifs. Par exemple, la classe de confi-

guration de la figure 9.7 a été renommée `cycabComplete` pour refléter le fait que tous les sous-composants sont présents. Lors des générations successives du fichier de définition des classes de configuration, les classes déjà existantes y sont conservées et les nouvelles y sont ajoutées avec des noms auto-générés.

```
CONFIGURATION CLASS cycabComplete BEGIN
Components     = {controller, gps, location, merger, wifi}
IProvided      = {gpsPosition, mergePosition, position, securePosition, wifiPosition}
IRequired      = {getGpsPosition, getMergePosition, getWifiPosition}
Parameters     = {GpsPowerUsage, Power, Trust, WifiPowerUsage}
ITypes         = {MergePosition, Position, SecurePosition}
PTypes         = {int}
InterfaceType  = {getGpsPosition->Position, getMergePosition->MergePosition, getWifiPosition->Position, gpsPosition->
      Position, mergePosition->MergePosition, position->SecurePosition, securePosition->SecurePosition, wifiPosition->
      Position}
Provider       = {gpsPosition->gps, mergePosition->merger, position->location, securePosition->controller, wifiPosition
      ->wifi}
Requirer       = {getGpsPosition->merger, getMergePosition->controller, getWifiPosition->merger}
Contingency    = {getGpsPosition->optional, getMergePosition->mandatory, getWifiPosition->optional}
ParameterType  = {GpsPowerUsage->int, Power->int, Trust->int, WifiPowerUsage->int}
Definer        = {GpsPowerUsage->merger, Power->controller, Trust->merger, WifiPowerUsage->merger}
Parent         = {(controller, location), (gps, location), (merger, location), (wifi, location)}
Binding        = {getGpsPosition->gpsPosition, getMergePosition->mergePosition, getWifiPosition->wifiPosition}
Delegate       = {securePosition->position}
State          = {controller->started, gps->started, location->started, merger->started, wifi->started}
CONFIGURATION CLASS cycabComplete END
```

FIGURE 9.7 – Définition de la classe de configurations `cycabComplete`

Le fait d'utiliser des classes de configuration nous permet de générer des traces courtes à partir de traces longues. Une trace courte consiste en un fichier dans lequel les paramètres qui vont être utilisés dans la trace sont spécifié dans l'ordre dans lequel on y fera référence dans la suite du fichier pour leur associer des valeurs. La trace courte contient aussi une liste de classes de configuration ou le chemin d'un ou plusieurs fichiers contenant de telles listes. Ainsi, avec ces informations, la configuration correspondant à la figure 9.4 peut être décrite par `cycabComplete(3,100,0,2,#0)`. Nous explicitons cette notations ci-dessous.

```
PARAMETER LIST BEGIN
GpsPowerUsage
Power
Trust
WifiPowerUsage
PARAMETER LIST END
```

FIGURE 9.8 – Paramètres utilisés dans une trace courte

La figure 9.8 montre une liste de paramètres telle qu'elle apparaît dans l'en-tête d'un fichier de trace courte. Ainsi en écrivant `cycabComplete(3,100,0,2,#0)` dans ce même fichier, on fait référence à une configuration ayant tous ses ensembles $Elem$ et Rel (voir définition 27) définis comme dans la figure 9.7 a l'exception de l'ensemble correspondant à la relation fonctionnelle $Value : Parameters \rightarrow Values$. Celui-ci peut être inféré de manière non ambigüe en considérant tous les paramètres de l'expression `cycabComplete(3,100,0,2,#0)` (sauf le dernier) dans l'ordre de la liste de la figure 9.8. Dans ce cas, on obtient "`Value = {GpsPowerUsage->3, Power->100, Trust->0, WifiPowerUsage->2}`". Le dernier paramètre "`#0`" représente l'indice correspondant à la configuration considérée sur le chemin d'évolution σ où l'indice de configuration correspond à l'indice de la configuration courante lors de la demande effectuée depuis une itération de la boucle de contrôle d'un contrôleur donné.

Maintenant nous pouvons décrire la totalité d'un fichier contenant une trace courte. Au début, ce fichier contient les en-têtes définissant les classes de configuration et les paramètres utilisés. Ensuite, suivent, pour chaque localisation la liste des configurations demandée depuis cette localisation dans un format similaire à `cycabComplete(3,100,0,2,#0)`. Finalement la liste des événements externes ainsi que l'indice des configurations auxquelles ces événements ont été détectés est ajoutée en fin

de fichier.

Il est facile, sur ces fichiers de trace courte où chaque configuration de la trace tient sur une seule ligne de générer des tests en masse en utilisant le mélange aléatoire, la duplication et/ou la suppression de configurations. Une fois ces tests générés, on peut reconvertir la trace courte en une trace longue qui pourra être rejouée par le Fuzzy Engine.

On peut automatiquement détecter (ou mettre de coté pour une examination future) des tests ayant une influence sur une politique d'adaptation que l'on souhaite tester en donnant un nom unique aux reconfigurations déclenchées par cette politique. Il est aussi possible d'ajouter une politique de réflexion additionnelle qui stoppe le système (ou effectue tout autre action jugée pertinente) pour chaque succès ou échec d'une reconfiguration déclenchée par la politique d'adaptation que l'on souhaite tester.

Précisons également que nous pouvons utiliser l'interface graphique de la section 9.2.2 pour lancer une exécution se basant sur un fuzzy log afin de simuler, en utilisant un modèle "dummy", l'exécution d'un système à composants (**CBS**) qui initialement utilisait le modèle Fractal ou FraSCAti.

9.4/ CONCLUSION

Dans ce chapitre nous avons décrit notre implémentation, $cbsdr$, et avons présenté un résultat de conformité architecturale en nous basant sur le modèle interprété de la section 4.4. Nous avons aussi présenté les différentes entités qui composent $cbsdr$, et détaillé leur fonctionnement. Ensuite, nous avons montré une partie des informations contenues dans les logs générés par notre implémentation et la façon dont elles sont utilisées pour produire les sorties pour notre interface graphique. Enfin, nous avons décrit l'intégration de la fonctionnalité de fuzzing permettant de tester des politiques d'adaptation.

10

EXPÉRIMENTATIONS

Dans le chapitre précédent, nous avons décrit notre implémentation et avons détaillé les différentes entités qui la composent ainsi que leur fonctionnement. Dans ce chapitre, nous présentons l'application des politiques d'adaptation que nous avons décrites au chapitre 8 au composant **Location** introduit à la section 3.1. L'exécution de notre implémentation permet de contrôler et guider les reconfigurations du composant **Location** en utilisant les mécanismes d'enforcement et de réflexion. Deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti sont présentés. Le premier modélise des machines virtuelles au sein d'un environnement *cloud* et le second s'inspire du serveur HTTP *Comanche Http Server* utilisé dans [Chauvel et al., 2009]. De plus, notre modèle a été implémenté avec l'outil de transformation de graphe GROOVE [Ghamarian et al., 2012]. Une expérimentation de cette implémentation utilisant comme cas d'étude une machine virtuelle de l'environnement *cloud* est présentée. Enfin, nous décrivons la façon dont nous simulons l'évaluation décentralisée sous GROOVE. Nous montrons comment nous représentons les propriétés FTPL sous GROOVE et présentons l'implémentation sous GROOVE de l'algorithme `LDMon` sur le modèle du système à composants **Location**.

Sommaire

10.1/ APPLICATION DE POLITIQUES D'ADAPTATION AU COMPOSANT **LOCATION**

Après avoir présenté, au chapitre 9, notre implémentation et les entités qui la composent, nous l'utilisons dans cette section pour appliquer les politiques d'adaptation que nous avons décrites au chapitre 8 au composant **Location** introduit à la section 3.1. Nous faisons dans un premier temps une brève description des politiques d'adaptation et des propriétés utilisées pour l'enforcement et la réflexion à l'exécution et nous présentons les résultats attendus. Ensuite, nous décrivons l'exécution de notre implémentation permettant de contrôler et guider les reconfigurations du composant **Location**. Enfin, nous examinons les résultats obtenus.

10.1.1/ DESCRIPTION DU COMPORTEMENT ATTENDU

Grâce aux politiques d'adaptation, les reconfigurations dynamiques du composant composite **Location** introduit à la section 3.1 peuvent être contrôlées à l'exécution. Nous utilisons la politique d'adaptation `cycabgps` du chapitre 8 (figure 8.1, qui est reproduite figure 10.1) qui influence l'addition ou le retrait du composant **GPS** ainsi que son alter-ego `cycabwifi` qui influence l'addition ou le retrait du composant **Wi-Fi**. Lors de l'application de ces politiques d'adaptation, nous appliquons l'enforcement à l'exécution de plusieurs propriétés FTPL. Cela signifie (voir section 6.1.1.2) que nous nous autorisons à appliquer une opération de reconfiguration donnée qu'après nous être assuré qu'elle ne mènerait pas à une configuration du système qui violerait l'une de ces propriétés.

Par exemple, pour la propriété FTPL suivante :

$$\varphi = \textbf{after}\ addwifi\ \textbf{terminates}\ (\textbf{before}\ removewifi\ \textbf{terminates}\ (\textbf{eventually}\ (Power < 33)))$$

nous spécifions dans une politique d'enforcement que φ doit toujours être évaluée à "potentiellement vrai", i.e., $\top^p$. Cela signifie que le composant **Wi-Fi**, après avoir été ajouté, ne devrait pas être retiré avant que le niveau de charge des batteries du véhicule (représenté par la variable $Power$) n'ait été mesuré à une valeur inférieure à 33%.

On notera que la signification d'une propriété FTPL comme φ peut être facilement comprise par un spécificateur. C'est un avantage pour l'utilisateur en charge de la conception des politiques d'adaptation. En revanche, la mise en place de reconfigurations (gardées) complexes avant leur inclusion dans des politiques d'adaptation peut s'avérer compliqué. C'est pourquoi il peut être utile d'utiliser des outils graphiques tels que Fractal Explorer [1] ou FraSCAti Explorer [2] en fonction du modèle de système à composants utilisé. Ces outils permettent l'exécution pas-à-pas de reconfigurations primitives lors de la phase de conception.

Rappelons que, intuitivement, l'enforcement permet d'éviter que quelque chose de non souhaitable (une violation de propriétés FTPL dans notre cas) ne se produise, tandis que la réflexion permet de déclencher une action correctrice après que quelque chose de non souhaitable ait été détecté. Considérons les propriétés FTPL ci-dessous. Celles désignées par la lettre ϖ sont utilisées pour l'application de l'enforcement et de la

1. http://fractal.ow2.org/tutorials/explorer/index.html
2. http://frascati.ow2.org/doc/1.1.1/ch06s02.html

```
1                policy cycabgps
2
3                event entry
4                event exit
5                event start
6
7                when (after start,exit (P_TRUE4 until entry))
                     = P_TRUE4
8                if (gps in Components) = FALSE
9                then utility of addgps is low
10
11               when (Power < 33) = TRUE4
12               if (gps in Components) = FALSE
13               then utility of addgps is low
14
15               when (Power < 33) = FALSE4
16               if (gps in Components) = FALSE
17               then utility of addgps is high
18
19               when (Power < 33) = FALSE4
20               if (gps in Components and wifi in Components)
                     = TRUE
21               then utility of removegps is low
22
23               when (after start,exit (P_TRUE4 until entry))
                     = P_TRUE4
24               if (gps in Components and wifi in Components)
                     = TRUE
25               then utility of removegps is high
26
27               when (Power < 33) = TRUE4
28               if (gps in Components and wifi in Components)
                     = TRUE
29               then utility of removegps is high
30
31               end policy
```

FIGURE 10.1 – La politique d'adaptation `cycabgps` du chapitre 8

réflexion à l'exécution, tandis que celle commençant par ϱ ne sont utilisées que pour la réflexion.

$$\varpi_{c_0} = power < 0$$
$$\varpi_{c_1} = \textbf{after } addwifi \textbf{ terminates } (\textbf{before } removewifi \textbf{ terminates } (\textbf{eventually } (power < 33)))$$
$$\varpi_{c_2} = \textbf{after } addgps \textbf{ terminates } (\textbf{before } removegps \textbf{ terminates } (\textbf{eventually } (power < 33)))$$
$$\varpi_{c_3} = \textbf{after } removegps \textbf{ terminates } (\textbf{before } addgps \textbf{ terminates } (\textbf{eventually } (power > 33)))$$
$$\varrho_{c_0} = power > 10$$
$$\varrho_{c_1} = \textbf{after } entry \ (\textbf{eventually } (wifi \in Components))$$

Des valeurs de $\mathbb{B}_4$ sont associées (dans les définitions des politiques d'enforcement et de réflexion) à ces propriétés afin de spécifier les résultats que nous attendons lors de

l'exécution. Lorsque durant l'exécution une de ces propriétés est évaluée à la même valeur que celle à laquelle elle est associée, on dira que cette propriété est satisfaite, sinon on dira qu'elle est violée. Les valeurs associées à chacune de ces propriétés sont les suivantes :

$$\varpi_{c_0} \rightarrow \bot \ ; \ \varpi_{c_1} \rightarrow \top^p \ ; \ \varpi_{c_2} \rightarrow \top^p \ ; \ \varpi_{c_3} \rightarrow \top^p \ ; \ \varrho_{c_0} \rightarrow \top \ ; \ \varrho_{c_1} \rightarrow \top^p$$

$\varpi_{c_0} \rightarrow \bot$ Ceci permet de s'assurer qu'aucune reconfiguration amenant à la violation de cette propriété (niveau d'énergie négatif) ne soit déclenchée via les politiques d'adaptation. S'il s'avère néanmoins que cette propriété soit violée, comme le fait d'avoir un niveau de charge des batteries négatif est incohérent, il est spécifié que le véhicule doit être stoppé via le mécanisme de réflexion.

$\varpi_{c_1} \rightarrow \top^p; \varpi_{c_2} \rightarrow \top^p; \varpi_{c_3} \rightarrow \top^p$ Ces propriétés, lorsqu'elles sont évaluées à une valeur de "potentiellement vrai" expriment qu'après avoir ajouté un composant (**GPS** ou **Wi-FI**), le niveau d'énergie doit avoir été mesuré en-deçà de 33% avant qu'il ne soit retiré (cas $\varpi_{c_1} \rightarrow \top^p; \varpi_{c_2} \rightarrow \top^p$) ou bien qu'après que le composant **GPS** ait été enlevé, le niveau d'énergie doit avoir été mesuré au-delà de 33% avant qu'il ne soit rajouté (cas $\varpi_{c_3} \rightarrow \top^p$). Les politiques d'adaptation permettent d'éviter que des reconfigurations menant le système dans une configuration violant ces propriétés ne soient déclenchées. Dans le cadre de la réflexion, si ces propriétés sont violées, le système est stoppé. On notera qu'il est néanmoins autorisé par nos politiques d'adaptation de rajouter le composant **Wi-Fi** après qu'il été enlevé sans aucune condition (sur le niveau d'énergie). Ceci permet de rajouter au plus vite le composant **Wi-Fi** lorsque le véhicule se trouve dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible.

$\varrho_{c_0} \rightarrow \top$ Cette propriété, utilisée seulement dans le cadre de la réflexion, stipule que le niveau d'énergie du véhicule est supérieur à 10%. Dans le cas contraire, une reconfiguration permettant de recharger les batteries du véhicule est déclenchée.

$\varrho_{c_1} \rightarrow \top^p$ Cette propriété, utilisée seulement dans le cadre de la réflexion, stipule qu'après que le véhicule dans une "zone Wi-Fi", le composant **Wi-Fi** est présent. Si ce n'est pas le cas ce composant est ajouté.

Le comportement attendu doit prendre en compte les politiques d'adaptation `cycabgps` et `cycabwifi` qui favorisent le retrait d'un système de positionnement (à savoir le composant **GPS** ou **Wi-Fi**) lorsque le niveau d'énergie est bas et favorisent son addition lorsque ce niveau est moyen ou haut. De plus, lorsque le véhicule est dans une "zone Wi-Fi", où il n'y a pas de signal GPS, la politique d'adaptation `cycabgps` favorise le retrait du composant **GPS**, tandis que la politique d'adaptation `cycabwifi` favorise l'ajout du composant **Wi-FI**.

Bien sûr, les reconfigurations déterminées par l'application des politiques d'adaptation ne doivent être appliquées que si elles ne mènent pas à une configuration qui viole une des propriétés ϖ_{c_0}, ϖ_{c_1}, ϖ_{c_2} ou ϖ_{c_3}. En outre la violation d'une des propriétés ϖ_{c_0}, ϖ_{c_1}, ϖ_{c_2}, ϖ_{c_3}, ϱ_{c_0} ou ϱ_{c_1} à n'importe quel moment de l'exécution doit déclencher une action corrective via le mécanisme de réflexion. Maintenant que nous avons décrit le comportement attendu, voyons comment se déroule l'exécution.

10.1.2/ EXÉCUTION

La figure 10.2 montre le résultat obtenu en exécutant notre implémentation sur le composant **Location** en appliquant les politiques d'adaptation `cycabgps` et `cycabwifi`. La partie haute de cette figure illustre l'évolution (en pourcentage) du niveau de charge des batteries du véhicule, tandis que les parties du milieu et du bas montrent respectivement la présence (valeur 1) ou l'absence (valeur 0) des composants **GPS** et **Wi-Fi**. Sur les trois graphes, les indices des différentes configurations du chemin d'évolution σ sont indiqués en abscisse.

Lorsque le véhicule entre dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible, l'événement *entry* est envoyé au contrôleur d'événements (**EC**) via le gestionnaire d'événements (**EH**) en charge de la collecte des événements externes. C'est le cas à la configuration d'indice 66 où l'entrée dans une "zone Wi-Fi" est symbolisée par un segment vertical vert ; c'est aussi le cas à la configuration d'indice 134. De façon similaire, la sortie d'une "zone Wi-Fi" correspond à l'occurrence de l'événement *exit* symbolisé par un segment vertical orange comme c'est le cas aux configurations ayant pour indice 78 et 147.

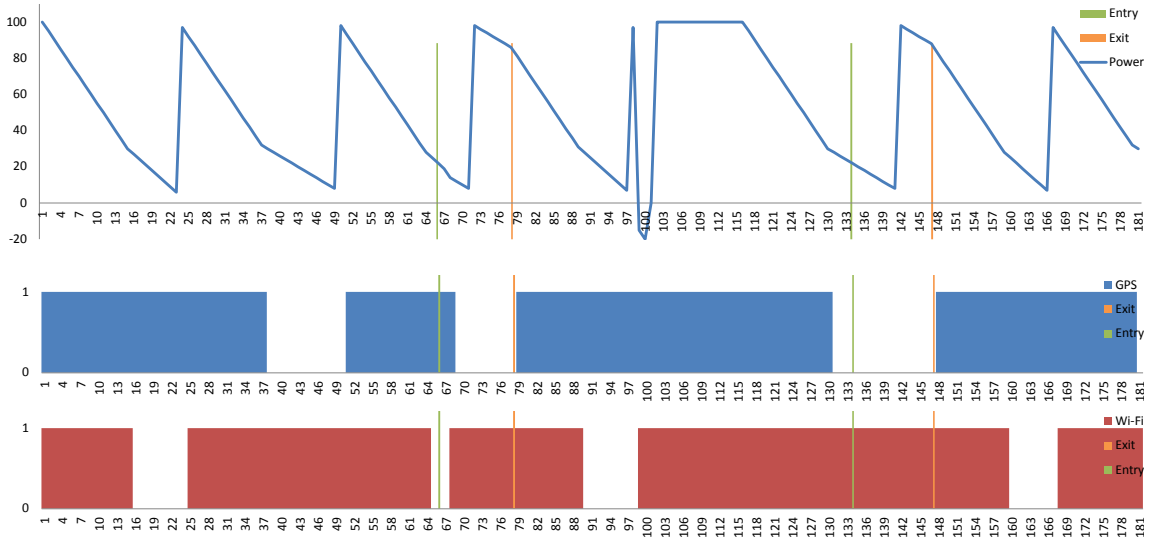


FIGURE 10.2 – Application de politiques d'adaptation sur le composant **Location**

Lorsque le pourcentage de charge des batteries descend au-dessous de 10, une politique de réflexion (utilisant la propriété ϱ_{c_0}) génère un événement qui déclenche une reconfiguration (nommée *chargeBattery*) qui met à jour le niveau d'énergie au maximum de capacité des batteries.

Juste avant la configuration d'indice 100, via l'API du modèle de système à composants utilisé pour l'exécution du composant **Location**, nous avons artificiellement changé le pourcentage de charge des batteries à une valeur négative. Ceci déclenche, via une politique de réflexion (basée sur les propriétés ϖ_{c_0} et ϱ_{c_0}), les reconfigurations *stopCycab* et *chargeBattery*. La reconfiguration *chargeBattery*, évoquée précédemment, met à jour le niveau d'énergie au maximum de capacité des batteries, tandis que la reconfiguration *stopCycab* arrête le composant **Location**.

L'arrêt total du composant **Location** est une mesure de sûreté effectuée en cas de circonstances exceptionnelles, potentiellement dangereuses ou incohérentes ; c'est le cas d'une valeur négative du pourcentage de charge des batteries.

Le composant **Location** étant à l'arrêt complet, le niveau de charge des batteries n'évolue pas jusqu'à la configuration d'indice 116 où le composite est redémarré manuellement via l'API du modèle de système à composants utilisé pour l'exécution.

La politique d'adaptation `cycabgps` favorise le retrait du composant **GPS** lorsque le niveau d'énergie est bas et favorise son addition lorsque ce niveau est moyen ou haut. La politique d'adaptation `cycabwifi` a une incidence similaire sur le composant **Wi-Fi**. De plus, lorsque le véhicule est dans une "zone Wi-Fi", où il n'ya pas de signal GPS, la politique d'adaptation `cycabgps` favorise le retrait du composant **GPS**, tandis que la politique d'adaptation `cycabwifi` favorise l'ajout du composant **Wi-FI**.

A la configuration d'indice 66, le véhicule entre dans une "zone Wi-Fi" en ayant seulement le composant **GPS** présent. A ce moment, le contrôleur de réflexion (**RC**) détecte que la propriété FTPL ϱ_{c_1} = **after** *entry* (**eventually** $wifi \in Components$) n'est pas évaluée à "potentiellement vrai", i.e., $\top^p$, comme spécifié dans la politique de réflexion et génère l'événement $reflexionNoWifiInWifiArea$. A la reconfiguration suivante, suite au retrait de l'événement $reflexionNoWifiInWifiArea$ par le contrôleur de politiques d'adaptation (**APC**), celui ci déclenche la reconfiguration $addwifi$ qui ajoute et démarre le composant **Wi-FI**. Ensuite, à la prochaine configuration, l'application de la politique d'adaptation `cycabgps` (voir figure 8.1 reproduite figure 10.1) cause le retrait du composant **GPS** du fait que le véhicule se trouve dans une "zone Wi-Fi" avec un niveau d'énergie faible.

A la configuration d'indice 134, le véhicule entre dans une "zone Wi-Fi" en ayant seulement le composant **Wi-Fi** présent. Lorsque le niveau d'énergie passe de faible à élevé (configuration d'indice 142) le composant **GPS** n'est pas ajouté malgré le fait le niveau d'énergie soit élevé. Ceci est dû à l'application de la politique d'adaptation `cycabgps` qui tempère l'utilité de l'ajout du composant **GPS** lorsque le véhicule se trouve dans une "zone Wi-Fi". On notera qu'aussitôt que le véhicule sort d'une "zone Wi-Fi", comme c'est le cas aux configurations d'indices 79 et 148, comme le niveau d'énergie est élevé, le composant **GPS** est ajouté et démarré.

En dehors des "zones Wi-Fi", le composant **GPS** est retiré aux configurations d'indices 37 et 130 en conséquence de l'application de la politique d'adaptation `cycabgps` car le niveau d'énergie est bas. C'est aussi le cas du composant **Wi-Fi** aux configurations d'indices 15, 64, 89 et 159 du fait de l'application de la politique d'adaptation `cycawifi`. Toujours en dehors des "zones Wi-Fi", lorsqu'un seul des composants **GPS** ou **Wi-Fi** est présent, l'autre est ajouté lorsque le niveau d'énergie relevé est moyen ou élevé, comme c'est le cas aux configurations d'indices 24, 50, 79, 99, 148 et 167.

10.1.3/ RÉSULTATS OBTENUS

Dans [Dormoy et al., 2010], un cas d'étude similaire utilisait des politiques d'adaptation inspirées par [Chauvel et al., 2009] qui étaient déclenchées par des expressions de la logique qMEDL [Gonnord et al., 2009] basée, entre autre, sur des événements externes. L'extension de la logique FTPL, introduite dans [Dormoy et al., 2012a], avec des événements externes rend possible l'expression de propriétés temporelles qui utilisaient précédemment qMEDL.

Au cours de l'exécution décrite ci-dessus, l'application des politiques d'adaptation `cycabgps` et `cycabwifi` permet bien le retrait du composant **GPS** ou du composant **Wi-Fi** lorsque le niveau de charge des batteries descend en-dessous de 33%. De plus, lorsque le niveau de charge des batteries remonte au-dessus de 33%, si seul un système de positionnement (à savoir le composant **GPS** ou **Wi-Fi**) est présent, l'autre est ajouté [3].

Nous avons aussi pu constater qu'aucune des propriétés (ϖ_{c_0}, ϖ_{c_1}, ϖ_{c_2} et ϖ_{c_3}) utilisées dans le cadre de l'enforcement n'est violée par une configuration obtenue via une reconfiguration déclenchée par les politiques d'adaptation `cycabgps` et `cycabwifi`. Ainsi, ni ϖ_{c_1}, ni ϖ_{c_2} n'ont été violées durant l'exécution, ce qui signifie qu'après qu'un composant (**GPS** ou **Wi-FI**) ait été ajouté, le niveau d'énergie avait été mesuré en-deçà de 33% avant qu'il n'ait été retiré. De façon similaire, ϖ_{c_3} n'a pas été violée durant l'exécution, car après que le système de positionnement **GPS** ait été retiré, le niveau d'énergie avait été mesuré au-delà de 33% avant qu'il n'ait été ajouté.

Notons que ϖ_{c_0} a été violée durant l'exécution lorsque le niveau d'énergie a été artificiellement changé à une valeur négative (configuration d'indice 100). Cette violation a été correctement détectée et a déclenché une action corrective approprié, à savoir l'arrêt du véhicule.

La propriété ϱ_{c_0}, quant à elle, a été violée à plusieurs reprises au cours de l'exécution. À chaque fois que le niveau d'énergie a été mesuré en-deçà de 10%, la violation a été détectée et a déclenché le rechargement des batteries comme action corrective.

Enfin, la propriété ϱ_{c_1} a été violée durant l'exécution à la configuration d'indice 66 lorsque le véhicule est entré dans une "zone Wi-Fi" en ayant seulement le composant **GPS** présent. Cette violation a été détectée et a engendré, en tant qu'action corrective, l'ajout du composant **Wi-Fi**.

Ceci nous permet de conclure que le comportement observé à l'exécution est conforme à celui attendu que nous avions décrit dans la section 10.1.1. De plus, ces résultats expérimentaux montrent qu'en utilisant la même logique (FTPL) pour *a*) définir des politiques d'adaptation et *b*) exprimer des contraintes architecturales, un spécificateur peut aborder et contrôler le comportement de systèmes à composants plus simplement que dans le cadre de [Dormoy et al., 2010].

10.2/ AUTRES CAS D'ÉTUDES

Nous présentons dans cette section deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti. Avec notre implémentation, *cbsdr*, nous avons pu appliquer des politiques d'adaptation à chacun de ces cas d'étude. Le premier modélise des machines virtuelles au sein d'un environnement *cloud* ; une de ces machines virtuelles sera d'ailleurs utilisée dans l'expérimentation de la section suivante. Le second cas d'étude s'inspire du serveur HTTP *Comanche Http Server* utilisé dans [Chauvel et al., 2009].

3. Sauf lorsque le véhicule se trouve dans une "zone Wi-Fi" où l'utilité de l'ajout du composant **GPS** est tempérée par l'application de la politique d'adaptation `cycabgps`.

10.2.1/ MACHINES VIRTUELLES ET ENVIRONNEMENT CLOUD

Dans cette section, nous considérons une application web typique à trois niveaux qui utilise un serveur web frontal, un serveur d'application intermédiaire (*middle-ware*) et un service d'arrière plan (*back-end*) comme une base de données ou un magasin de données (*data store*).

La figure 10.3 montre le composant **virtualMachine** qui représente une machine virtuelle (ou VM pour *virtual machine*) qui héberge les trois services de notre application. Les sous-composants **httpServer**, **appServer** et **dataServer** fournissent respectivement le service web frontal (HTTP), le service d'application intermédiaire (APP) et le service d'arrière plan qui fournit les données (DATA).

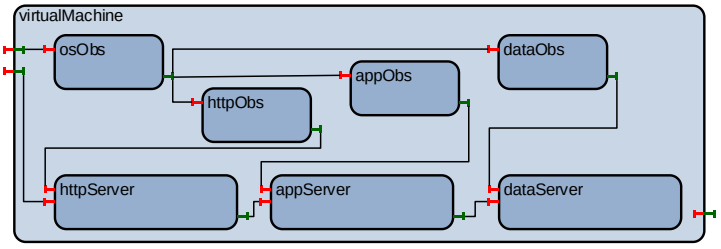


FIGURE 10.3 – Composant représentant une machine virtuelle contenant une application à trois niveaux

De plus, la VM de la figure 10.3 contient aussi quatre observateurs (*observers*) qui sont des sous-composants utilisés pour la surveillance du bon fonctionnement des services hébergés. Le sous-composant **osObs** est utilisé pour la surveillance du système d'exploitation de la VM. Il est aussi lié aux sous-composants **httpObs**, **appObs** et **dataObs** utilisés pour observer les services respectifs des sous-composants **httpServer**, **appServer** et **dataServer**.

Le composant composite **virtualMachine** dispose aussi de deux interfaces fournies : L'une pour fournir des services qui est liée au sous-composant **httpServer** et l'autre pour observer le bon fonctionnement de la VM qui est liée au sous-composant **osObs**.

La figure 10.4 illustre un exemple d'environnement *cloud* qui contient une VM **vmDev** qui regroupe les trois niveaux de notre application web sans aucun observateur ; une telle machine est dite non managée (*unmanaged*). Les trois autres contiennent des observateurs et sont dites managées (*managed*). Chacune de ces VM contient un niveau de l'application ainsi que les sous-composants responsables de l'observation du système d'exploitation et du service hébergé par la VM.

Le composant composite **cloudEnv** qui représente l'environnement *cloud* lui-même dispose de trois interfaces fournies. Deux sont utilisées pour fournir le service selon qu'il soit en version managée ou non. La troisième est liée à un sous-composant **monitorObs** lié aux interfaces fournies dédiées à l'observation du bon fonctionnement des VM.

Un fournisseur d'environnement *cloud* doit pouvoir fournir à la demande des VM, à l'unité ou par lot, configurées avec les bons services et observateurs. Dans ce contexte, nous considérons le provisionnement d'une VM unique. Une VM, en fonction des services à fournir et du fait qu'elle soit managée ou non, doit être configurée avec les sous-composants adéquats. Durant le cycle de vie de la VM des changements de configuration peuvent intervenir ; nous les considérons comme des reconfigurations du système à composants **virtualMachine** représenté figure 10.3.

Nous effectuons ces changements de configuration via des politiques d'adaptation basées sur des événements externes. Ainsi, l'occurrence de l'événement $sethttp$

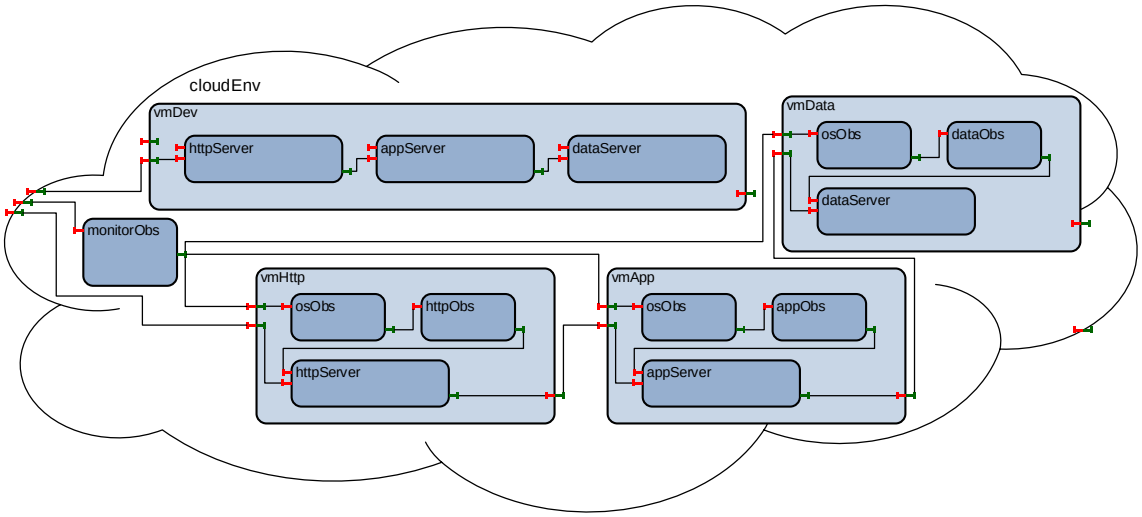


FIGURE 10.4 – Exemple d'environnement *cloud*

déclenche l'ajout du composant **httpServer** et celle de l'événement $unsethttp$ son retrait. Des événements comme $sethttp$ et $unsethttp$ qui déclenchent une action et son contraire sont dits opposés. De même, l'ajout des composants **appServer** ou **dataServer** est déclenché par l'occurrence respective des événements $setapp$ ou $setdata$, tandis que leur retrait respectif correspond aux événements $unsetapp$ ou $unsetdata$. De plus, les événements $manage$ et $unmanage$ permettent respectivement l'ajout et le retrait des composants correspondant aux observateurs permettant la surveillance du bon fonctionnement des services. Enfin, l'événement $init$ permet d'initialiser les interfaces fournies du composite **virtualMachine** en les liant (via la relation $delegate$) à des sous-composants factices ne fournissant qu'un service simpliste afin de permettre l'intégration à un environnement *cloud* comme celui représenté figure 10.4.

Par exemple, étant donné une VM managée qui ne fournit que le service HTTP ; cette VM contient le composant **httpServer** et puisqu'elle est managée, elle contient aussi les composants **osObs** et **httpObs**, comme c'est le cas de la VM **vmHttp**, en bas à gauche de la figure 10.4. Ainsi, la détection de l'événement $setdata$ déclenche via des politiques d'adaptation l'ajout des composants **dataServer** et **dataObs**. Bien sûr, si la VM initiale n'était pas managée, l'événement $setdata$ ne déclencherait que l'ajout du composant **dataServer**. Néanmoins, dans ce dernier cas, la détection des événements $setdata$ et $manage$ donnerait une VM managée fournissant les services HTTP et DATA (i.e., une VM contenant les composants **httpServer**, **dataServer**, **osObs**, **httpObs** et **dataObs**).

L'ordre dans lequel apparaissent les événements permettant les changements de configuration des VM n'est pas impactant. En effet, nous utilisons des expressions FTPL de la forme "**after** $unsetdata$ $(\top^p$ **until** $setdata)$" qui, pour des événements opposés ($setdata$ et $unsetdata$ dans notre cas) est évaluée à "potentiellement vrai" après l'occurrence d'un événement jusqu'à l'occurrence de l'événement opposé.

10.2.2/ SERVEUR HTTP

La figure 10.5 montre le composant composite **httpServer**, un modèle d'un serveur HTTP similaire au serveur *Comanche Http Server* utilisé dans [Chauvel et al., 2009]. Ce com-

posite contient, entre autre, le sous-composant **requestReceiver** qui reçoit les requêtes envoyées au serveur HTTP et les transmet au composant **requestHandler**. Ce dernier peut récupérer la réponse à la requête envoyée soit auprès du composant **requestDispatcher**, soit auprès du composant **cacheHandler** qui maintient un cache contenant les résultats de requêtes passées. Le composant **requestDispatcher**, quand à lui, récupère le contenu du fichier spécifié dans la requête auprès des composants **fileServer1** et **fileServer2** qui correspondent tous deux à des serveurs de fichiers.

En fonction de la charge et de la similarité entre les requêtes il peut être utile d'ajouter ou d'enlever le composant **cacheHandler** qui gère un cache. De plus, par défaut, le composant **fileServer2** n'est pas présent, mais il peut être judicieux de le rajouter en cas de forte activité du composant **fileServer1**.

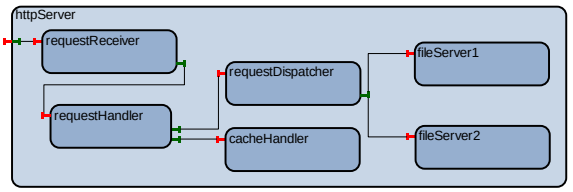


FIGURE 10.5 – Modèle de serveur HTTP

La figure 10.6 montre le résultat d'une expérimentation utilisant le composant composite **httpServer** au cours de laquelle, comme dans [Chauvel et al., 2009], des requêtes HTTP sont simulées. Sur la figure, la courbe en pointillé bleue représente la charge (*load*) en nombre de requêtes par unité de temps, la courbe continue en rouge correspond à l'écart-type entre les différentes requêtes (*request deviation*). La ligne faite de tirets verts représente la présence du composant **cacheHandler** et la courbe faite de tirets bleus ciel représente la taille du cache qui peut prendre trois valeurs que l'on qualifiera de basse, moyenne ou haute [4]. Enfin, la ligne faite de tirets et de points violets représente la présence du composant **fileServer2**.

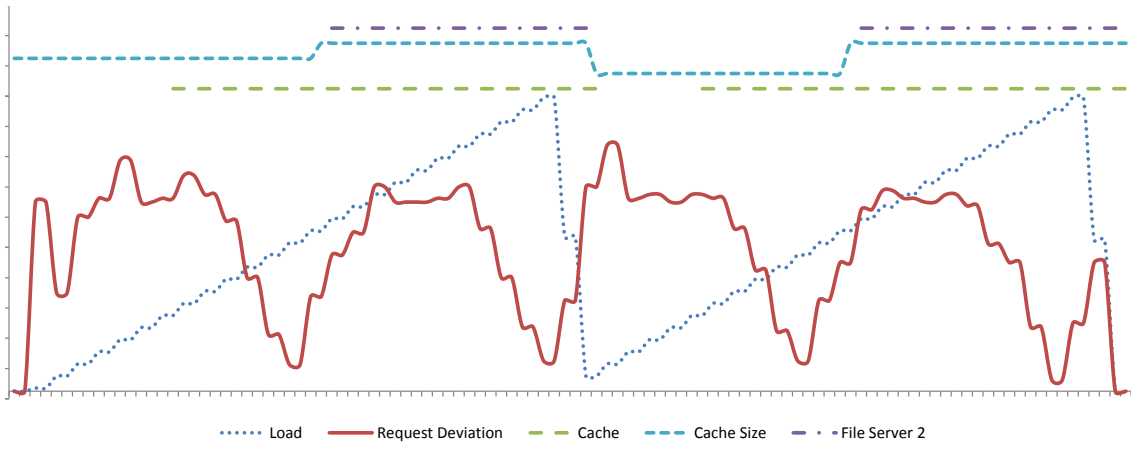


FIGURE 10.6 – Expérimentation avec le composant composite **httpServer**

Il est intéressant de noter que les temps de réponses mesurés lorsque le composant composite **httpServer** est guidé et contrôlé par notre application, $cbsdr$, sont très proches de ceux mesurés pour un composant composite comme celui de la figure 10.5 ayant un cache (avec une valeur de taille haute) et deux composants correspondant à des serveurs de fichiers. Cela signifie que l'usage de $cbsdr$ a permis le retrait de composants non nécessaires, l'ajout de composants appropriés et le bon dimensionnement du cache

4. Les positions basse, intermédiaire et haute de cette courbe représentent respectivement les valeurs basse, moyenne et haute.

sans utiliser plus de ressources que le composant composite configuré avec tous ses sous-composants présents.

10.3/ UTILISATION DE GRAMMAIRES DE GRAPHES

Cette section décrit comment notre modèle a été implémenté avec l'outil de transformation de graphe GROOVE [Ghamarian et al., 2012]. Nous présentons notre implémentation sous GROOVE dans un premier temps puis nous décrivons une petite expérimentation de cette implémentation en utilisant comme cas d'étude le composant composite **virtualMachine** de la figure 10.3.

10.3.1/ IMPLÉMENTATION SOUS GROOVE

GROOVE utilise des graphes simples pour la modélisation de structures de systèmes orientés objets à la conception (*design-time*), à la compilation (*compile-time*) et à l'exécution (*runtime*). Les graphes sont composés de points ou nœuds (*nodes*) et de liens (*edges*) pouvant être étiquetés (*labelled*). Les transformations de graphes présentent une base pour la transformation de modèles ou pour la sémantique opérationnelle de systèmes. Notre implémentation utilise le mode *typé* de GROOVE afin de garantir que tous les graphes soient correctement typés. Ce mode permet la définition de types génériques et de règles de transformation de graphes auxquelles peuvent être intégrées des priorités de sorte qu'une règle n'est appliquée que si aucune autre de plus grande priorité ne correspond au graphe courant.

La figure 10.7 montre le typage défini pour les composants. Un type $component$, qui est abstrait (ceci est indiqué par les pointillés qui l'entourent), peut avoir un attribut $started$ ou $stopped$ correspondant respectivement à un état démarré ou arrêté. Les sous-types de $component$ qui héritent (comme l'indiquent les flèches aux têtes évidées) de ses caractéristiques sont $composite$ et $primitive$ qui sont respectivement utilisés pour représenter des composants composites et primitifs.

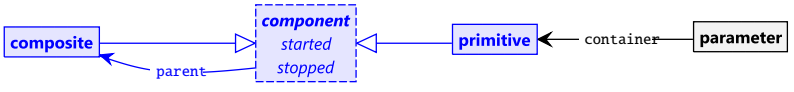


FIGURE 10.7 – Les types de composant définis dans GROOVE

Ainsi on exprime bien qu'un composant composite n'a pas de paramètres et peut être parent d'autres composants (comme l'illustre la flèche étiquetée "*parent*"). De façon similaire, la figure 10.7 montre aussi qu'un composant primitif peut avoir des paramètres (comme l'illustre la flèche étiquetée "*container*") mais ne peut être parent.

Les transformations de graphes sont spécifiées par des règles qui sont composées

— de modèles (*patterns*) qui doivent être présents ou absents pour que la règle considéré s'applique,

— d'éléments (points et liens) à ajouter ou enlever, et

— de paires de points devant être fusionnés.

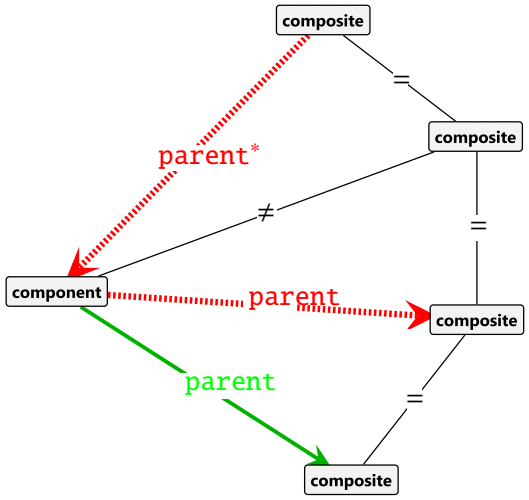
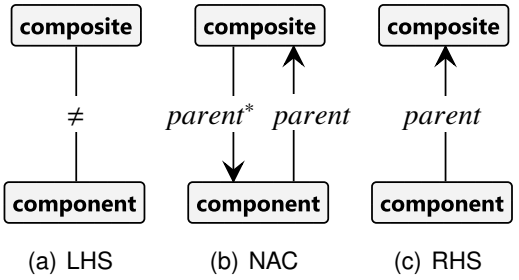


FIGURE 10.8 – Opération primitive *add* représentée dans GROOVE

Dans Groove, une nomenclature basée sur les couleurs et les formes permet de différencier aisément ces règles. Par exemple, notre implémentation sous GROOVE modélise l'opération primitive *add* en utilisant la règle représentée dans la figure 10.8 On y voit *a*) un composant composite et un composant quelconque (primitif ou composite) où les liens étiquetés par "=" assurent que le point "composite" est le même point de type *composite*, tandis que le lien étiqueté "≠" assure que le point "component" est de type *component* et est différent de celui étiqueté "composite" ; *b*) les flèche rouges (discontinue en gras) étiquetées "*parent**" et "*parent*" assurent qu'il n'y a pas de relation *parent* ou *parent** [5] entre les points étiquetés "composite" et "component". La flèche verte (en gras) étiquetée "*parent*" indique que si les conditions ci-dessus sont satisfaites un lien symbolisant la relation "*parent*" entre les points "component" et "composite" doit être créé.

Bien sûr de telles règles de transformation peuvent aussi être exprimées en utilisant des sous-graphes : *a*) un sous-graphe LHS (*Left Hand Side*) représente les pré-conditions de la règle, *b*) un sous-graphe NAC (*Negative Application Condition*) spécifie ce qui prévient l'application de la règle, et *c*) un sous-graphe RHS (*right hand side*) représente les post-conditions. Les sous-graphes LHS, NAC et RHS qui expriment la règle décrite dans GROOVE sur la figure 10.8 sont représentés figure 10.9.



(a) LHS (b) NAC (c) RHS

FIGURE 10.9 – Équivalent de la règle GROOVE de la figure 10.8 en utilisant des graphes LHS, NAC et RHS

Notre implémentation accepte, en entrée, un graphe représentant un système à composants modélisé au moyen du modèle présenté au chapitre 3. Un tel graphe décrit une configuration au sens de la définition 27 où les éléments et les relations sont respectivement représentées par des nœuds et des liens.

La figure 10.10 montre une copie d'écran de l'interface graphique de GROOVE affichant, dans la partie principale, un graphe modélisant un système à composants similaire à celui du composant composite **vmApp** de la figure 10.4. Les composants sont représentés en bleu, les interfaces requises et fournies, sont respectivement représentées en magenta

5. Dans notre modèle, nous utilisons la relation $Descendants$ qui est formellement la relation inverse de la fermeture transitive de $Parent$, i.e., $Parent^{+-1}$, et peut être déduite de $Parent$. Nous utilisons la relation $Descendants$ qui est définie en utilisant seulement la logique du premier ordre. Comme la fermeture transitive ne peut être exprimée par la logique du premier ordre, nous n'utilisons pas cette notion d'un point de vue formel, mais nous profitons du fait que GROOVE supporte la fermeture transitive afin de simplifier les règles de transformation de graphes.

et rouge, les paramètres en noir et les types (que ce soient des types d'interfaces ou de paramètres) en gris. La partie en haut à gauche montre la liste des règles de transformation de graphe ordonnées par ordre de priorité, tandis qu'en bas à gauche se trouve la liste des types que nous utilisons dans GROOVE.

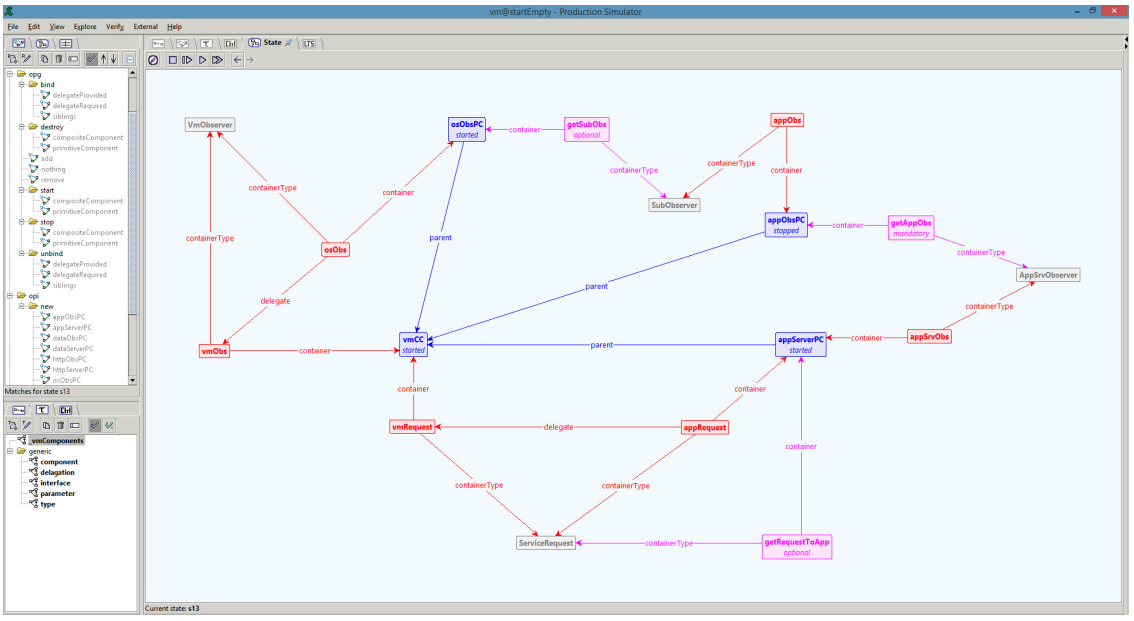


FIGURE 10.10 – Modèle du système à composants **vmApp** affiché par GROOVE

10.3.2/ LE COMPOSANT **VIRTUALMACHINE** REPRÉSENTÉ SOUS GROOVE

Nous considérons une VM représentée, comme c'est le cas figure 10.3, par un composant composite **virtualMachine** pouvant contenir des sous-composants **httpServer**, **appServer** ou **dataServer** fournissant des services qui composent une application. Cette VM peut aussi contenir des observateurs, qui sont des sous-composants utilisés pour vérifier le bon fonctionnement des services. Le sous-composant **osObs** vérifie le bon fonctionnement du système d'exploitation de la VM et peut être lié aux sous-composants **httpObs**, **appObs** ou **dataObs** respectivement utilisé pour s'assurer du bon fonctionnement des services des sous-composants **httpServer**, **appServer** ou **dataServer**.

TABLE 10.1 – Génération du code d'installation

Fonctionnalité	data	app	http	managed
Bit #	3	2	1	0
ic = 0	0	0	0	0
ic = 5	0	$2^2 = 4$	0	$2^0 = 1$
ic = 10	$2^3 = 8$	0	$2^1 = 2$	0
ic = 11	$2^3 = 8$	0	$2^1 = 2$	$2^0 = 1$

Chaque VM a sa configuration déterminée par un code d'installation (ic, pour *Install Code*) qui est un nombre entier. On considère ic comme un nombre binaire pour lequel chaque bit active ou désactive une fonctionnalité de la VM. Ceci est illustré par la table 10.1 où la première ligne affiche les fonctionnalités et la seconde les numéros des bits correspondants pour le code d'installation. Les lignes suivantes détaillent la génération de codes d'installation pour une VM ayant seulement un système d'exploitation sans aucun service (ni observateur) installé ($ic = 0$), un serveur d'application managé ($ic = 5$), tandis que les deux dernières lignes décrivent un serveur contenant un service

HTTP et un service DATA qui est soit non managé ($ic = 10$), soit managé ($ic = 11$).

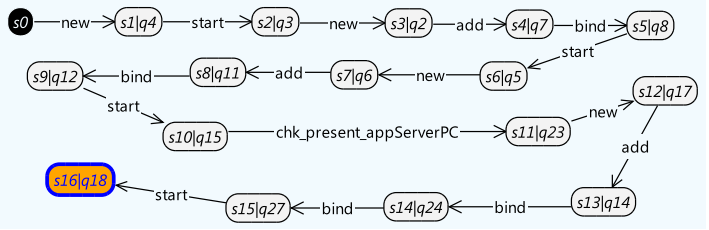


FIGURE 10.11 – Serveur d'application managé ($ic = 5$)

Notre implémentation crée la représentation d'un graphe représentant le système à composants **virtualMachine** dont la configuration est spécifiée par un code d'installation donné. La figure 10.11 décrit le système de transitions généré par GROOVE durant la création d'un serveur d'application managé ($ic = 5$) où le premier état (`s0`) représente un graphe vide et `s1` correspond à un graphe représentant le composant composite **virtualMachine** arrêté et sans aucun sous-composant. L'état `s2` désigne un graphe avec le même composant composite ayant été démarré. On voit que les transitions sont étiquetées par les opérations de reconfiguration primitives étant effectuées. Ainsi de suite, les sous-composants nécessaires sont créés, ajoutés, liés et démarrés via les opérations primitives *new*, *add*, *bind* et *start*. On notera aussi la présence d'une transition étiqueté "chk_present_appServerPC" qui représente une assertion qui permet de vérifier la présence du service APP fourni par le composant **appServer**. Ainsi, en utilisant le langage de contrôle de GROOVE, une fonction *manage*() ajoute et configure les sous-composants correspondant aux observateurs adéquats.

Pour une VM fournissant les services HTTP et DATA ($ic = 10$ or $ic = 11$), l'ordre de liaison et démarrage des sous-composants correspondant n'est pas prédéterminé comme l'illustre la figure 10.12. On y voit que l'évolution a d'abord lieu de façon déterministe de l'état `s0` à `s8`. L'état `s16`, en haut à droite représente un graphe correspondant à un code d'installation de valeur 10, i.e., une VM non managée fournissant les services HTTP et DATA. Depuis cet état on peut appliquer la fonction GROOVE *manage*() ce qui permet l'occurrence des transitions nécessaires à l'obtention d'une VM managée fournissant les services HTTP et DATA ($ic = 11$) entre les états `s23` et `s41`. On remarquera que l'évolution de `s8` à `s16` est non-déterministe. Nous avons deux chemins optimums (*shortest paths*), `s8` → `s10` → `s16` et `s8` → `s11` → `s16` qui peuvent être aisément découvert au moyen d'un parcours en largeur (*breadth-first exploration*).

La table 10.2 affiche le nombre d'états et de transitions du système de transitions de graphe pour chaque code d'installation. Le système de transition de graphe pour $ic = 11$, correspondant à la figure 10.12 a 42 états et 82 transitions. On remarque que dans notre implémentation, l'ordre des reconfigurations primitives est totalement déterministe (i.e., il y a exactement un état de plus que le nombre de transitions, ce qui revient à dire que le système de transitions de graphe est un chemin linéaire) pour $0 \leq ic \leq 5$ et $8 \leq ic \leq 9$.

TABLE 10.2 – Nombre d'états et de transitions par code d'installation

Unmanaged			**Managed**		
ic	states	transitions	*ic*	states	transitions
0	3	2	1	7	6
2	7	6	3	17	16
4	7	6	5	17	16
6	46	155	7	62	171
8	7	6	9	17	16
10	26	66	11	42	82
12	26	66	13	42	82
14	265	1456	15	288	1479

Considérons la table 10.2 pour $6 \leq ic \leq 7$ et $10 \leq ic \leq 13$, on voit que pour chaque VM ayant deux services, la version managée a 16 états et 16 transitions de plus que la version non managée. Une déduction similaire peut aussi être faite pour la dernière ligne de la

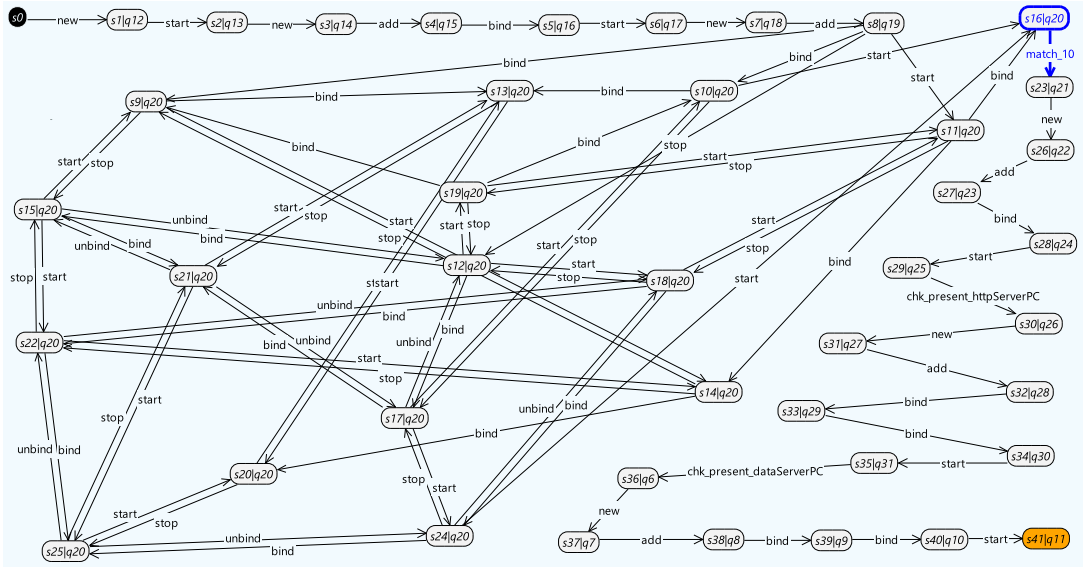


FIGURE 10.12 – VM managé fournissant les services HTTP et DATA ($ic = 11$)

table 10.2, avec cette fois une différence de 23. Ceci s'explique par le fait que la fonction GROOVE *manage*() engendre de façon déterministe les opérations primitives permettant de passer d'une VM non managée à une VM managée, comme l'illustre la figure 10.12 (entre les états `s23` et `s41`). Autrement dit, l'application de la fonction *manage*() à une VM non managée engendre un nombre de transitions supplémentaires dépendant du nombre de services de la VM :

— L'application de la fonction *manage*() à une VM non managée n'ayant aucun service ($ic = 0$) génère 4 transitions supplémentaires ;

— lorsque la fonction *manage*() est appliquée à une VM non managée ayant un seul service ($ic \in \{2, 4, 8\}$), 10 transitions supplémentaires ont lieu ;

— si la fonction *manage*() est appliquée à une VM non managée ayant deux services ($ic \in \{6, 10, 12\}$), 16 transitions supplémentaires ont lieu ; enfin

— l'application de la fonction *manage*() à une VM non managée ayant trois services ($ic = 14$) génère 23 transitions supplémentaires.

Mentionnons également que le nombre d'états et de transitions pour $ic = 6$ (VM non managé fournissant les service HTTP et APP) est différent de ceux pour $ic = 10$ (VM non managé fournissant les service HTTP et DATA) ou 12 (VM non managé fournissant les service APP et DATA). Ceci est dû au fait que contrairement aux sous-composants **httpServer** et **appServer**, **dataServer** n'a pas d'interface requise (voir figure 10.3), ce qui implique un déterminisme plus fort autorisant moins d'états et de transitions. La même différence peut être observée pour les codes d'installations similaires correspondant à des VM managées ; à savoir que le nombre d'états et de transitions pour $ic = 7$ est différent de ceux pour $ic = 11$ et $ic = 13$.

10.4/ ÉVALUATION DÉCENTRALISÉE SIMULÉE SOUS GROOVE

Dans cette section nous décrivons la façon dont nous simulons l'évaluation décentralisée sous GROOVE. Nous montrons d'abord comment nous représentons les propriétés FTPL

sous GROOVE. Ensuite nous présentons l'implémentation sous GROOVE de l'algorithme `LDMon` (voir figure 7.1) sur le modèle du système à composants **Location**.

10.4.1/ REPRÉSENTATION DES PROPRIÉTÉS FTPL SOUS GROOVE

Nous utilisons GROOVE, comme dans la section 10.3, pour modéliser la structure de systèmes à composants à l'exécution au moyen de graphes typés composés de points et de liens pouvant être étiquetés. La transformation de graphes nous permet de mettre en place la sémantique opérationnelle définie au chapitre 3. Comme nous l'avons déjà détaillé dans la section 10.3.1, les règles de transformations sous GROOVE sont représentées par *a*) des modèles (*patterns*) qui doivent être présents ou absents pour que la règle s'applique, *b*) des éléments (points et liens) à ajouter ou enlever, et *c*) des paires de points devant être fusionnés. Dans Groove, une nomenclature basée sur les couleurs et les formes permet de différencier aisément ces règles.

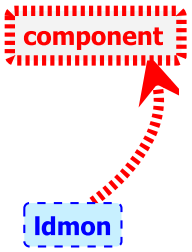


FIGURE 10.13 – Représentation dans GROOVE de la règle `removeOrphanMon`

Par exemple, l'implémentation de l'évaluation décentralisée de propriétés temporelles décrite au chapitre 7, utilise la règle `removeOrphanMon`, représentée figure 10.13, pour s'assurer qu'aucune instance de l'algorithme `LDMon` (voir figure 7.1) ne puisse exister lorsque le composant auquel elle était dédiée est supprimé. La figure 10.13 montre un point de type *component* représentant un composant, un point de type *ldmon* désignant une instance de l'algorithme `LDMon` et un lien symbolisant le fait l'instance de l'algorithme `LDMon` représentée soit un moniteur du composant. Les lignes rouges en gras et pointillés sont utilisées dans GROOVE pour marquer des éléments ne devant pas être présents dans le graphe courant, tandis les pointillés fins de couleur bleu montrent les éléments présents devant être supprimés. Ainsi la règle de la figure 10.13 exprime le fait que si une instance de l'algorithme `LDMon` est présente dans le graphe courant sans qu'elle soit le moniteur d'un composant, cette instance doit être supprimée.

Notre implémentation tire partie du typage de graphes possible sous GROOVE et utilise la priorisation des règles de graphes. Ceci nous permet de garantir que les graphes représentant des systèmes à composants soient correctement typés et qu'une règle de transformation correspondant à un graphe donné ne soit appliquée que si aucune règle de priorité plus grande correspondant à ce graphe n'existe. Notre implémentation sous GROOVE accepte en entrée un graphe contenant une formule FTPL et un graphe correspondant à un système à composants représenté en utilisant le modèle du chapitre 3.

Afin d'illustrer la représentation de formule FTPL sous forme de graphe, nous considérons le système à composants composite **Location** introduit à la section 3.1. Ce composite supporte le retrait ou l'ajout de ses sous-composants **GPS** et **Wi-Fi** pour peu qu'à un instant donné au moins un des deux soit présent. Dans certains cas, néanmoins, il peut être pertinent d'enlever l'un de ces composants. Par exemple, lorsque le niveau d'énergie des batteries du véhicule est bas, le composant **Wi-Fi** peut être retiré, puis rajouté plus tard lorsque les batteries sont rechargées. De plus, lorsque le véhicule entre dans une "zone Wi-Fi", où il n'y pas de signal GPS disponible, il est pertinent de retirer le composant **GPS** qui peut être rajouté après que le véhicule soit sorti de cette zone si le niveau des batteries le permet.

Considérons la formule FTPL suivante :

$$\varphi = \textbf{after}\ \mathit{removegps}\ \textbf{normal}\ (\textbf{before}\ \mathit{addgps}\ \textbf{normal}\ (\textbf{eventually}\ (\mathit{power} \geq 33)))$$

Nous pouvons écrire $\varphi = \textbf{after}\ e_0\ \phi$, avec $e_0 = \mathit{removegps}\ \textbf{normal}$, $\phi = \textbf{before}\ e_1\ trp$, $trp = \textbf{eventually}\ cp$, $e_1 = \mathit{addgps}\ \textbf{normal}$ et $cp = (\mathit{power} \geq 33)$. Intuitivement, φ exprime qu'après le retrait du composant **GPS**, le niveau de charge des batteries du véhicule doit atteindre au moins 33% avant que ce composant puisse être rajouté à nouveau. La figure 10.14 montre comment nous représentons la formule φ sous forme de graphe dans GROOVE.

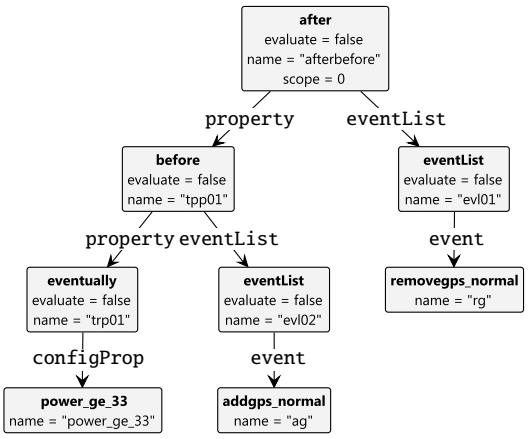


FIGURE 10.14 – Représentation sous forme de graphe de la propriété FTPL φ

On y voit que le point du haut, de type "after", nommé `afterbefore` représente une propriété temporelle de la forme "**after** $evl01\ tpp01$" où $tpp01$ est une propriété temporelle et $evl01$ est une liste d'événements qui sont chacune symbolisées par un sous-graphe associé au point `afterbefore` via des liens respectivement nommés `property` et `eventList`. La propriété de la forme "**before** $evl02\ trp01$" représentée par le point nommé `tpp01` est formée de la même façon. En revanche, le point `trp01` qui représente une propriété de la forme "**enventually** $power\_ge\_33$" est lié, par un lien libellé `configProp` à un point nommé `power_ge_33` qui correspond à la propriété de configuration $power \geq 33$. Les points nommés `evl01` et `evl02` correspondant à des listes d'événements (contenant chacune un seul événement) sont liés à des points correspondant à des événements par des liens de type "event". Ainsi, en reprenant les notations ci-dessus :

- $cp = (\mathit{power} \geq 33)$ est représenté par le point nommé `power_ge_33`,
- $trp = \textbf{eventually}\ cp$ est représenté par le point nommé `trp01`,
- $e_1 = \mathit{addgps}\ \textbf{normal}$ est représenté par le point nommé `evl02`,
- $\phi = \textbf{before}\ e_1\ trp$ est représenté par le point nommé `tpp01`,
- $e_0 = \mathit{removegps}\ \textbf{normal}$ est représenté par le point nommé `evl01` et
- $\varphi = \textbf{after}\ e_0\ \phi$ est représenté par le point nommé `afterbefore`.

10.4.2/ SIMULATION DE L'ÉVALUATION DÉCENTRALISÉ

Nous présentons dans cette section l'implémentation sous GROOVE de l'algorithme `LDMon` (voir figure 7.1) sur le modèle du système à composants **Location**. Le composant **Location** contient différents systèmes de positionnement : Wi-Fi et GPS qui sont respectivement gérés par les sous-composants **Wi-FI** et **GPS**. Le sous-composant **Merger** permet de combiner les positions obtenues tandis que le sous-composant **Controller** peut requérir la position courante auprès du sous-composant **Merger** et aussi avoir accès à des informations du véhicule comme, par exemple, le niveau de charge des batteries.

La figure 10.15 montre une copie d'écran de l'interface graphique de GROOVE affichant, dans la partie principale, un graphe modélisant le système à composants **Location**. Les

composants y sont représentés en bleu, les interfaces requises et fournies, sont respectivement représentées en magenta et rouge, les paramètres en noir et les types (que ce soient des types d'interfaces ou de paramètres) en gris.

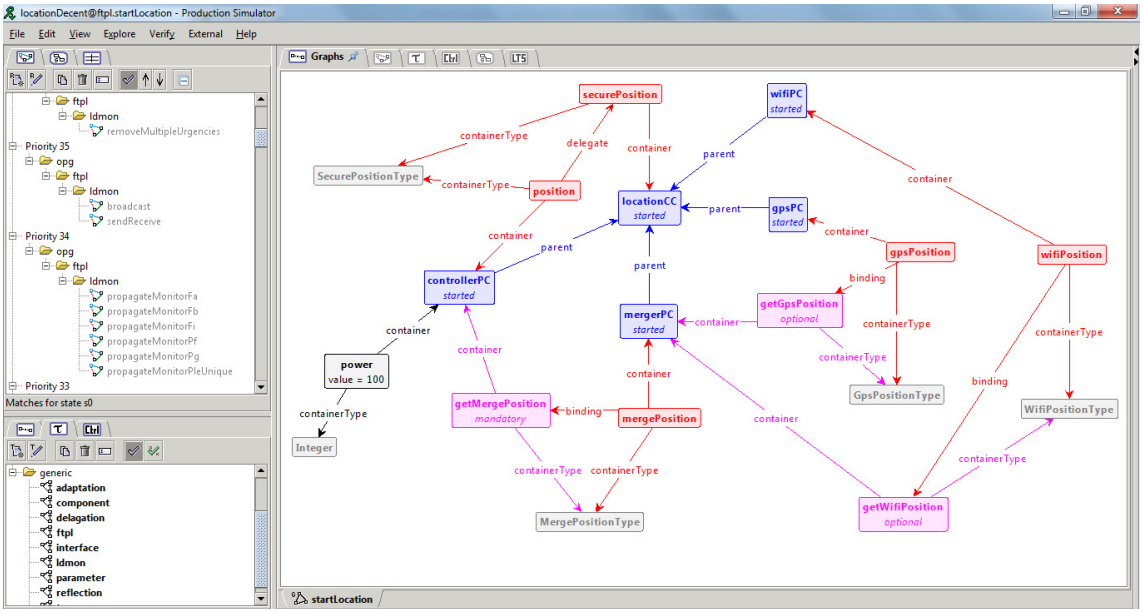


FIGURE 10.15 – Modèle du système à composants **Location** affiché par GROOVE

Grâce aux politiques d'adaptation intégrant des propriétés temporelles, le composant composite **Location** peut être modifié pour utiliser seulement le sous-composant **GPS** ou **Wi-Fi** ou bien les deux en fonction du niveau de charge des batteries ou de l'occurrence de certains événements, comme l'entrée ou la sortie d'une "zone Wi-Fi" dans laquelle aucun signal GPS ne peut être capté.

Par exemple, lorsque le niveau de charge des batteries est faible, si le véhicule est dans une "zone Wi-Fi", il peut être utile de retirer le composant **GPS**. Ceci est illustré par la figure 10.16, qui représente le suffixe σ_k d'un chemin de configuration σ. On considère que toutes les opérations de reconfiguration du suffixe σ_k sont des opérations génériques représentées par run excepté entre $\sigma(i_0 - 1)$ et $\sigma(i_0)$, d'une part, où l'opération de reconfiguration est $removegps$ et entre $\sigma(i_1 - 1)$ et $\sigma(i_1)$, d'autre part, où l'opération de reconfiguration est $addgps$.

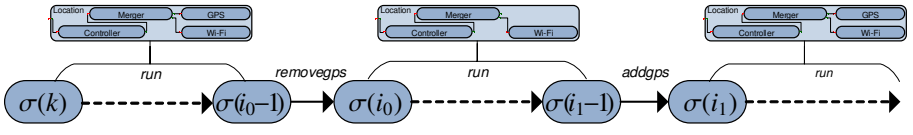


FIGURE 10.16 – Représentation du suffixe σ_k du chemin de configuration σ

Au cours de l'exécution un contrôleur de politiques d'adaptation (**APC** pour *Adaptation Policy Controller*, comme décrit dans la section 9.1.1) doit évaluer des propriétés FTPL telles que la propriété temporelle φ de la section 10.4.1. Le contrôleur de politiques d'adaptation (**APC**) utilise l'algorithme `LDMon` (voir figure 7.1) pour évaluer de telles propriétés temporelles. Nous décrivons ci-dessous l'évaluation de φ par notre implémentation sous GROOVE de l'algorithme `LDMon`.

Dans notre implémentation, un moniteur est affecté à chaque sous-composant du composant composite **Location**. Soient M_c, M_m, M_g et M_w quatre moniteurs respectivement affectés aux composants **Controller**, **Merger**, **GPS** et **Wi-Fi**. Avec les notations de la figure 10.14, le moniteur M_c a accès à la valeur de la propriété de configuration représentée par `power_ge_33`, tandis que M_m peut évaluer celles représentées par `addgps_normal` et `removegps_normal`. On notera que ces propriétés de configuration correspondent à des événements atomiques (voir section 7.1).

Dans le cadre de l'évaluation décentralisée utilisant l'algorithme `LDMon`, les moniteurs M_g et M_w n'ont accès à aucun événement atomique pouvant influencer l'évaluation de φ. Autrement dit, en utilisant les notations de la définition 63, $sus(\varphi) \cap AE_g = sus(\varphi) \cap AE_w = \emptyset$, où AE_g et AE_w contiennent les événements atomiques correspondant respectivement aux composants **GPS** et **Wi-Fi**.

Dans notre implémentation sous GROOVE, le moniteur M_{comp} d'un composant donné **Comp** est présenté par un sous-graphe lié au point représentant **Comp** par un lien de type `monitor`. Les communications entre moniteurs utilisent respectivement des liens de type `sentreceived` et `broacast` pour les envois/réceptions de messages et leurs diffusions globales.

Considérons un chemin d'évolution σ représentant les séquences des configurations du composant composite **Location** où les transitions entre les reconfigurations sont des opérations de reconfiguration. Nous rappelons que dans notre modèle, les communications entre moniteurs sont couvertes par les opérations génériques *run* car elles n'affectent pas directement l'architecture du système. Nous supposons que sur le suffixe σ_k de σ, représenté figure 10.16, toutes les opérations de reconfiguration sont *run* sauf *a*) entre $\sigma(i_0 - 1)$ et $\sigma(i_0)$ où la reconfiguration $removegps$ a lieu, et *b*) entre $\sigma(i_1 - 1)$ et $\sigma(i_1)$ où il s'agit de $addgps$.

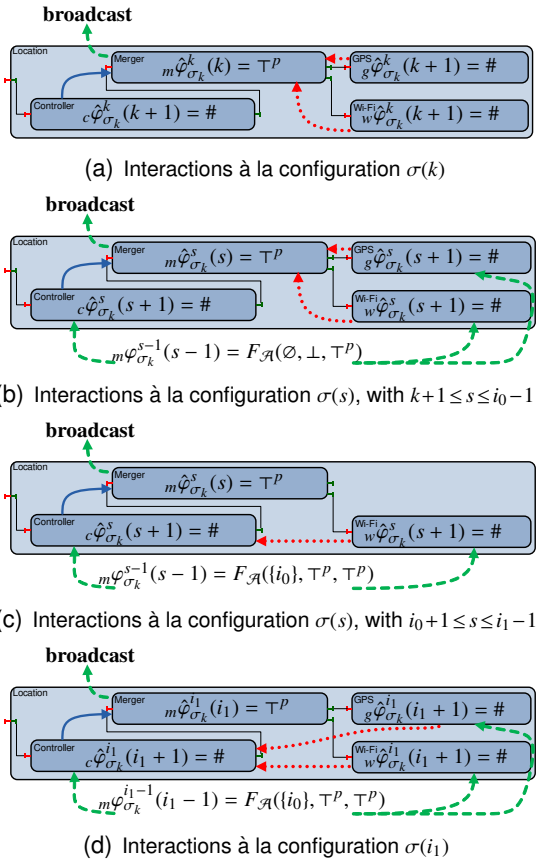
Décrivons à présent comment, à chaque configuration d'indice s, où $k \leq s \leq i_1$, la propriété φ est évaluée par l'algorithme `LDMon` avant d'être transmise au contrôleur de politiques d'adaptation (**APC**). La figure 10.17 décrit les interactions entres les moniteurs correspondant aux sous-composants du composite **Location**. Sur cette figure, les flèches vertes formées de tirets représentent les diffusions de messages (*broadcast*). Les flèches bleues en traits continus, quant à elles, symbolisent l'envoi d'une formule FTPL. Enfin, les flèches rouges en pointillés indiquent *a*) que le moniteur correspondant au composant désigné par la flèche est capable d'évaluer la sous-formule la plus urgente du moniteur correspondant au composant d'où vient la flèche, et *b*) qu'aucune communication n'a lieu entre ces moniteurs.

Puisque ni M_g et M_w n'ont accès à des événements atomiques impliqués dans l'évaluation de φ, ils n'envoient aucun message durant l'exécution détaillée ci-dessous.

À la configuration $\sigma(k)$ puisque M_m peut évaluer $e_0 = \bot$, d'après la définition 50 φ est évalué à $\top^p$ sur le suffixe σ_k et nous avons ${}_m\hat{\varphi}^k_{\sigma_k}(k) = \top^p$ en utilisant les notations de la section 7.3. Ce résultat étant établi par M_m, il est diffusé comme l'indique la flèche verte en tirets de la figure 10.17(a). Les autres moniteurs font progresser leur formule et déterminent que la sous-formule la plus urgente peut être évaluée par M_m. C'est pourquoi M_c envoie la formule à évaluer ainsi que les sous-formules qu'il a déjà pu évaluer ; ceci est symbolisé par la flèche bleu continue de la figure 10.17(a). Sur cette même figure les flèches rouges en pointillés expriment le fait que bien que les moniteurs M_g et M_w aient pu

déterminer que la sous-formule la plus urgente peut être évaluée par M_m, ils n'envoient rien car ils n'ont pu évaluer aucune sous-formule de φ.

À chaque configuration $\sigma(s)$ pour $k+1 \leq s \leq i_0 - 1$ comme aucune occurrence de e_0 n'a lieu, l'évaluation décentralisée consiste en l'évaluation de φ par M_m qui retourne et diffuse le résultat de la même façon qu'à la configuration $\sigma(k)$. De plus, les autres moniteurs reçoivent le résultat diffusé à la configuration précédente par M_m, comme l'illustrent les flèches vertes en tirets au bas de la figure 10.17(b). Le résultat ${}_m\varphi^{s-1}_{\sigma_k}(s-1)$ déterminé par M_m et est évalué à $F_{\mathcal{A}}(\varnothing, \bot, \top^p)$ ce qui, d'après la définition 59 correspond à $\top^p$. À chaque configuration $\sigma(s)$ (pour $k+1 \leq s \leq i_0 - 1$), les autres moniteurs font progresser leur formule et déterminent que la sous-formule la plus urgente peut être évaluée par M_m. C'est pourquoi M_c effectue un envoi vers M_m.



(a) Interactions à la configuration $\sigma(k)$

(b) Interactions à la configuration $\sigma(s)$, with $k+1 \leq s \leq i_0-1$

(c) Interactions à la configuration $\sigma(s)$, with $i_0+1 \leq s \leq i_1-1$

(d) Interactions à la configuration $\sigma(i_1)$

FIGURE 10.17 – Interactions entres les moniteurs du composant **Location**

À la configuration $\sigma(i_0)$ l'événement e_0 a lieu. Comme $e_0 = removegps$ **normal**, cela signifie que le composant **GPS** a été retiré. Le moniteur M_m étant averti de cette occurrence, l'évaluation de φ s'effectue en utilisant ${}_m\hat{\varphi}^{i_0}_{\sigma_k}(i_0) = {}_m\hat{\phi}^{i_0}_{\sigma_{i_0}}(i_0) = \top^p$ car d'après la définition 51, une propriété de type **before** est évaluée à $\top^p$ à la première configuration de son suffixe. Ce résultat est alors retourné et diffusé par M_m. Toujours à la configuration $\sigma(i_0)$, M_c et M_w reçoivent le résultat diffusé par M_m à la configuration précédente et M_c envoie ses formules à M_m.

À chaque configuration $\sigma(s)$ pour $i_0 + 1 \leq s \leq i_1 - 1$ comme e_0 a déjà eu lieu une fois, le moniteur M_m évalue ${}_m\hat{\varphi}^{s}_{\sigma_k}(s_0) = {}_m\hat{\phi}^{s}_{\sigma_{i_0}}(s) = \top^p$ puisque $\phi = \textbf{before}\ e_1\ trp$ et e_1 n'a pas encore eu lieu. Le résultat de cette évaluation est retourné et diffusé par M_m comme le montre la figure 10.17(c). De plus, M_c et M_w reçoivent le résultat diffusé à la configuration précédente qui contient, dans une sous-formule, l'information que e_0 a eu lieu à la configuration $\sigma(i_0)$. La formule que M_c a fait progressé contient ${}_c\hat{\phi}^{s}_{\sigma_{i_0}}(s+1)$ qui est évaluée par $\mathcal{F}_{\mathcal{B}}(\overline{\mathbf{X}}e_1, \hat{trp}^{s-1}_{\sigma_{i_0}}(s), \top^p)$, d'après la définition 60. Supposons qu'il y ait une configuration $\sigma(s')$, où $i_0 < s' < i_1$, pour laquelle le pourcentage de charge des batteries du véhicule dépasse 33%, i.e., $cp = (power > 33) = \top$. Dans ce cas, pour $s \geq s'$, $\hat{trp}^{s}_{\sigma_{i_0}}(s) = cp \sqcup \hat{trp}^{s-1}_{\sigma_{i_0}}(s) = \top$ d'après la définition 57 pour une propriété de type **eventually** et donc, l'ensemble de formules que M_c envoie à M_m (voir figure 10.17(c)) contient $\mathcal{F}_{\mathcal{B}}(\overline{\mathbf{X}}e_1, \top, \top^p)$ et $trp^{s}_{\sigma_{i_0}}(s)$.

À la configuration $\sigma(i_1)$ l'événement e_1 a lieu. Comme $e_1 = addgps$ **normal**, cela signifie que le composant **GPS** a été ajouté. M_c, M_g et M_w reçoivent le résultat diffusé à la configuration précédente, M_c et M_w se comportent comme à la configuration $\sigma(s)$ où $i_0 + 1 \leq s \leq i_1 - 1$, tandis que M_g se comporte comme M_w. Enfin, M_m évalue sa formule à ${}_m\hat{\varphi}^{i_1}_{\sigma_k}(i_1) = {}_m\hat{\phi}^{i_1}_{\sigma_{i_0}}(i_1) = \mathcal{F}_{\mathcal{B}}(\top, \overline{\mathbf{X}}trp_{\sigma_{i_0}}, \top^p) = \top^p$, en utilisant le fait que $\overline{\mathbf{X}}trp_{\sigma_{i_0}}$ avait été envoyé par M_c lors d'une configuration précédente. Ceci répond au problème TPDEP de la section 7.3.

La table 10.3 donne des informations sur les règles de transformation de graphes utilisées par notre implémentation sous GROOVE pour l'évaluation décentralisée de la propriété φ détaillée ci-dessus. Les colonnes de cette table représentent, de gauche à droite, la valeur possible de l'indice des configurations considérées, le nombre de règles de graphe utilisées pour cette configuration, la reconfiguration ayant lieu (le cas échéant) et la partie de la formule FTPL devant être évaluée pour obtenir un résultat dans $\mathbb{B}_4$.

TABLE 10.3 – Règles de transformation de graphe utilisées à la configuration $\sigma(s)$

Indice s de la configuration	Nombre de règles de transformation	Reconfiguration	Partie de la formule à évaluer
$s = k$	85		**after** *removegps* **normal** ...
$k + 1 \leq s \leq i_0 - 1$	111 – 162		**after** *removegps* **normal** ...
$s = i_0$	237	*removegps*	... **before** *addgps* **normal** ...
$i_0 + 1 \leq s \leq i_1 - 1$	149		... **before** *addgps* **normal** ...
$s = i_1$	253	*addgps*	... **eventually** $(power \geq 33)$

À la configuration $\sigma(k)$, on voit que 85 règles de transformation de graphes sont utilisées. Il s'agit principalement de règles concernant l'évaluation d'événements FTPL. En effet, tant que l'événement *removegps* **normal** n'a pas eu lieu, seule l'évaluation de la partie **after** *removegps* **normal** ... de la formule est nécessaire pour obtenir un résultat.

À la configuration $\sigma(s)$, où $k + 1 \leq s \leq i_0 - 1$, de 111 à 162 règles de transformation de graphe sont utilisées. Ce nombre varie en fonction de la population de l'historique des évaluations construite par chaque moniteur. Une fois que l'historique a atteint sa longueur maximale (à savoir $|\mathcal{M}| + 1$, où $|\mathcal{M}|$ représente le nombre de moniteurs), les règles les plus utilisées sont celles en charge du nettoyage de l'historique que l'on ne conserve pas.

À la configuration $\sigma(i_0)$, la reconfiguration *removegps* a lieu. Ainsi, tant que l'événement *addgps* **normal** n'a pas lieu, seule l'évaluation de la partie ... **before** *addgps* **normal** ... est nécessaire à l'obtention d'un résultat. La plupart des 237 règles de transformation de graphe permettent d'enlever les éléments du sous-graphe représentant le moniteur du composant **GPS** ayant été retiré.

À la configuration $\sigma(s)$, où $i_0 + 1 \leq s \leq i_1 - 1$, 149 règles de transformation de graphe sont utilisées, principalement pour le nettoyage de l'historique que l'on ne conserve pas. Enfin, à la configuration $\sigma(i_1)$, la reconfiguration *addgps* a lieu. À ce moment, l'évaluation de la partie ... **eventually** $(power \geq 33)$ de la formule à la configuration précédente est nécessaire pour obtenir un résultat. La plupart des règles de transformation de graphe sont utilisées *a*) pour le nettoyage de l'historique que l'on ne conserve pas, et *b*) pour la construction du sous-graphe représentant le moniteur du composant **GPS** ayant été ajouté.

10.5/ DISCUSSION ET CONCLUSION

Pour conclure ce chapitre, rappelons que nous avons décrit à la section 10.1 l'application de politiques d'adaptation, via notre implémentation *cbsdr*, développée en Java, au composant **Location** introduit à la section 3.1. Dans ce contexte, les propriétés temporelles utilisées par les politiques d'adaptation sont évaluées de façon centralisée. Le comportement de notre implémentation observé à l'exécution correspond à celui attendu. Nous avons aussi utilisé GROOVE (section 10.3) pour décrire notre modèle de reconfiguration. Ainsi, chaque configuration est représentée par un graphe, tandis que l'évolution d'une configuration à la suivante est effectuée au moyen de règles de transformation de graphes. Notons que cette représentation est proche de celle décrite dans [Le Métayer, 1998], mais diffère notamment du fait que nous utilisons des nœuds spécifiques pour représenter les types des interfaces des composants. Dans la section 10.4, nous avons décrit l'implémentation sous GROOVE de l'algorithme `LDMon` (voir figure 7.1) sur le modèle du système à composants **Location**.

La principale différence entre nos implémentations sous GROOVE et celle sous Java, à savoir *cbsdr*, réside dans le fait que sous GROOVE nous ne considérons que des graphes représentant des systèmes à composants, tandis que *cbsdr* permet de manipuler de véritables systèmes à composants élaborés en utilisant les modèles de systèmes à composants Fractal ou FraSCAti. En outre, *cbsdr* permet de reconfigurer dynamiquement des systèmes à composants au moyen de politiques d'adaptation basées sur des propriétés temporelles évaluées de manière centralisée. Nos implémentations sous GROOVE, en revanche, ne permettent pas l'utilisation de politiques d'adaptation mais ont d'autres usages. Ainsi, dans la section 10.3, nous avons pu produire sous forme de graphe le système de transitions des configurations d'un système à composant. Ceci permet, par exemple, de pouvoir déterminer si une configuration donnée est atteignable ou aussi de pouvoir vérifier la validité d'une propriété sur l'ensemble des configurations atteignables. Dans la section 10.4, l'implémentation sous GROOVE de l'algorithme `LDMon` nous a permis de décrire l'évaluation décentralisée de propriétés temporelles au moyen de moniteurs étant chacun associé à un composant primitif. La mise en place de l'évaluation décentralisée au sein de notre implémentation développée en Java (*cbsdr*) pourrait constituer une prochaine étape. Il conviendra néanmoins, lors du passage à l'échelle, de prendre en compte les résultats du chapitre 7, notamment que l'approche décentralisée est particulièrement performante vis-à-vis de l'approche centralisée dans le cas de systèmes comprenant un grand nombre de composants pour l'évaluation d'une propriété FTPL dont les événements atomiques sont distribués sur une faible proportion des composants.

Dans ce chapitre, nous avons appliqué les politiques d'adaptation que nous avons décrites au chapitre 8 au composant **Location** introduit à la section 3.1. L'exécution de notre implémentation permet de contrôler et guider les reconfigurations du composant **Location** en utilisant les mécanismes d'enforcement et de réflexion pour obtenir les résultats attendus. Deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti ont été présentés. Le premier modélise des machines virtuelles au sein d'un environnement *cloud* et le second s'inspire du serveur HTTP *Comanche Http Server* utilisé dans [Chauvel et al., 2009]. De plus notre modèle a été implémenté avec l'outil de transformation de graphe GROOVE [Ghamarian et al., 2012]. Une expérimentation de cette implémentation utilisant comme cas d'étude une machine virtuelle de l'environnement *cloud* a été présentée.

Enfin, nous avons décrit la façon dont nous simulons l'évaluation décentralisée sous GROOVE. Nous avons montré comment nous représentons les propriétés FTPL sous GROOVE et avons présenté l'implémentation (toujours sous GROOVE) de l'algorithme `LDMon` sur le modèle du système à composants **Location**.

CONCLUSION ET PERSPECTIVES

Les travaux de recherche effectués et présentés dans cette thèse ont pour but d'expliciter la manière dont nous pouvons guider et contrôler les reconfigurations de systèmes à composants au moyen de politiques d'adaptation basées sur des propriétés temporelles. Nous commençons par dresser le bilan de nos travaux en indiquant nos contributions. Puis, nous en présentons les perspectives futures.

SYNTHÈSE ET BILAN

Nous reprenons nos contributions en les déclinant en trois parties. Tout d'abord, nous énonçons celles concernant la modélisation des reconfigurations dynamiques. Ensuite, viennent les contributions concernant la vérification à l'exécution et le contrôle des reconfigurations. Enfin, nous décrivons comment nos contributions pratiques permettent de valider expérimentalement les contributions théoriques présentées dans cette thèse.

MODÉLISATION DES RECONFIGURATIONS DYNAMIQUES

Tout d'abord, des précisions au modèle de système à composants servant de base à nos travaux [Dormoy, 2011] ont été apportées. En effet, les éléments architecturaux $Contingencies$, $States$ et $Values$ n'étaient pas des éléments de $Elem$ dans [Dormoy, 2011]. La raison de cet ajout réside dans le fait que ces éléments sont utilisés pour définir le profil (voir définition 9) des relations architecturales contenues dans l'ensemble Rel ; Si ces éléments ne faisaient pas partie de $Elem$, la configuration $\langle Elem, Rel \rangle$ ne serait pas une signature au sens de la définition 9.

Au delà de ces changements mineurs, deux autres modifications ont été effectuées : *a*) d'une part, la relation $Binding$ qui était précédemment une relation fonctionnelle ne l'est plus pour pouvoir autoriser plusieurs interfaces requises à être liées à une interface fournie (comportement classique d'une relation client-serveur) et *b*) d'autre part, la relation fonctionnelle $Descendant$ a été ajoutée à Rel.

Ce dernier changement a permis de définir les contraintes de consistance de la table 4.1 en utilisant uniquement la logique du premier ordre et de ne plus avoir à utiliser la notion de fermeture transitive comme c'était le cas précédemment. Ce fait permettrait à présent une vérification automatique [Lanoix et al., 2011] du modèle générique.

Les reconfigurations gardées ont été introduites en se basant sur les pré/post-conditions des opérations primitives de reconfiguration. Ces reconfigurations ont permis d'utiliser les opérations primitives en tant que "briques de base" pour construire des reconfigurations non-primitives impliquant des constructions, non seulement séquentielles, mais aussi alternatives ou répétitives.

Il a été établi que l'utilisation de reconfigurations gardées garantit la consistance des configurations d'un système à composants sous la condition que les configurations initiales soient consistantes. De plus, un modèle interprété a été défini en partant du modèle abstrait présenté au chapitre 3. Le résultat de propagation de consistance des configurations a été étendu au modèle interprété.

VÉRIFICATION ET CONTRÔLE DES RECONFIGURATIONS DYNAMIQUES

Nous avons travaillé sur la spécification et la vérification de propriétés de systèmes à composants reconfigurables en étendant la logique FTPL introduite dans [Dormoy et al., 2012a] avec des événements externes. Ceci a permis de faire le lien entre les schémas temporels prenant en compte des événements externes qui avaient été introduits dans [Dormoy et al., 2010] et la logique FTPL fondée sur les travaux de Dwyers concernant les schémas de spécification [Dwyer et al., 1999]. La sémantique de cette logique a ensuite été décrite et la notion de satisfaction d'une propriété FTPL dans $\mathbb{B}_2$ a été définie.

La sémantique de la logique FTPL (étendue aux événements externes) a été établie en utilisant les quatre valeurs de $\mathbb{B}_4$ pour l'évaluation de propriétés FTPL à l'exécution sur des traces d'exécution incomplètes. Une méthode d'évaluation progressive, comme dans [Bauer et al., 2012, Bacchus et al., 1998], a été introduite, permettant ainsi d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur un historique borné des évaluations aux configurations précédentes. Ceci a permis l'évaluation de ces propriétés à l'exécution sans avoir à maintenir un historique complet des évaluations et, dans la plupart des cas, d'évaluer une propriété FTPL à une configuration donnée en se basant seulement sur son évaluation à la configuration précédente.

Une méthode, inspirée par l'évaluation décentralisée de formules LTL [Bauer et al., 2012], pour l'évaluation décentralisée de propriétés FTPL utilisant les notions de progression et d'urgence a été proposée. Le problème de l'évaluation décentralisée a aussi été posé formellement et un algorithme pour le résoudre ainsi que des résultats de correction et de terminaison ont été proposés.

Des politiques d'adaptation intégrant la logique FTPL ont été définies. Une relation de simulation sur les modèles de systèmes à composants reconfigurables a aussi été proposée et le problème de l'adaptation a été défini. Il a été établi, en utilisant cette relation de simulation, que le problème de l'adaptation est semi-décidable et un algorithme pour l'enforcement des politiques d'adaptation des systèmes à composants reconfigurables a été fourni. Enfin, l'usage du fuzzing pour le test de politiques d'adaptation a été présenté.

Les résultats de ces travaux ont fait l'objet de publications dans des conférences internationales [Kouchnarenko et al., 2014a, Kouchnarenko et al., 2014b, Kouchnarenko et al., 2015, Weber, 2016] ainsi que dans une revue nationale [Kouchnarenko et al., 2016].

CONTRIBUTIONS PRATIQUES

Les résultats décrits ci-dessus ont été implémentés et, dans cette thèse, l'implémentation permettant de guider et contrôler les reconfigurations de systèmes à composants via des politiques d'adaptation basées sur la logique temporelle FTPL a été détaillée. Cette

implémentation supporte les modèles à composants Fractal et FraSCAti. Un résultat de conformité architecturale se basant sur le modèle interprété défini précédemment a été établi. Les différentes entités qui composent notre implémentation ont été présentées et leur fonctionnement a été détaillé. Ensuite, une partie des informations contenues dans les logs générés par notre implémentation et la façon dont elles sont utilisées pour produire les sorties pour notre interface graphique ont été exposées. En outre, l'intégration de la fonctionnalité de fuzzing permettant de tester des politiques d'adaptation a été décrite.

L'exécution de notre implémentation permettant de contrôler et guider les reconfigurations d'un cas d'étude en utilisant les mécanismes d'enforcement et de réflexion a donné les résultats attendus. Deux autres cas d'étude ayant été chacun implémentés en utilisant les modèles de systèmes à composants Fractal et FraSCAti ont été présentés.

Par ailleurs, pour déterminer si une configuration donnée est atteignable ou pour vérifier la validité d'une propriété sur l'ensemble des configurations atteignables, notre modèle a été implémenté avec l'outil de transformation de graphe GROOVE [Ghamarian et al., 2012]. Une expérimentation de cette implémentation a été présentée. En outre, la façon dont l'évaluation décentralisée est simulée sous GROOVE a été décrite. Il a aussi été montré comment les propriétés FTPL sont représentées sous GROOVE. L'implémentation (toujours sous GROOVE) de l'algorithme permettant l'évaluation décentralisée de propriétés FTPL a été présentée sur un de nos cas d'étude. Ces contributions expérimentales et pratiques ont montré la pertinence des contributions théoriques grâce à l'implémentation prouvant les concepts concernés.

PERSPECTIVES

Cette section discute les perspectives envisagées qui sont organisées ci-dessous en quatre groupes. Tout d'abord, nous présentons celles liées à l'évaluation décentralisée de propriétés temporelles et aux politiques d'adaptation. Dans un second temps, nous indiquons les perspectives d'évolution de notre implémentation. Ensuite, nous nous intéressons au passage à l'échelle et au test. Enfin, nous mentionnons les systèmes cyber-physiques et micromécatroniques pour lesquels nos contributions peuvent trouver des applications directes.

ÉVALUATION DÉCENTRALISÉE ET POLITIQUES D'ADAPTATION

Nos perspectives concernant l'évaluation décentralisée de propriétés temporelles et les politiques d'adaptation sont présentées ci-dessous.

Évaluation décentralisée et reconfigurations concurrentes L'analyse du problème de l'évaluation décentralisée de propriétés temporelles peut être étendue au cas de plusieurs opérations de reconfigurations concurrentes. Ce problème serait déjà solvable dans le cas de reconfigurations s'appliquant à des configurations dont les événements atomiques n'ont pas d'influence sur l'évaluation des propriétés temporelles concernées. De telles reconfigurations sont celles qui ne modifient pas les événements atomiques de l'ensemble des sous-formules urgentes de ces propriétés temporelles. Ainsi, ces reconfigurations pourraient s'appliquer simultanément sur différentes parties indépendantes du

système à composants considéré. En revanche, dans le cas contraire, deux approches peuvent être envisagées. La première consisterait à bloquer (e.g., par un mécanisme de verrou) les reconfigurations susceptibles de modifier des événements atomiques ayant une influence sur les propriétés temporelles concernées par l'évaluation décentralisée. Dans le cadre de la seconde approche, on laisserait ces reconfigurations s'effectuer et on procéderait, si besoin, à une réécriture des sous-formules ayant progressé jusqu'à ce point. La première approche, bien que plus facile à mettre en place, pourrait mener à des interblocages (*deadlock*) ; c'est pourquoi la seconde approche serait à privilégier sur le long terme.

Combinaison de politiques d'adaptation Considérons le cas de plusieurs systèmes à composants guidés et contrôlés par des politiques d'adaptation distinctes. Si ces systèmes sont amenés à coexister au sein d'un même assemblage, il peut être opportun de combiner leurs politiques d'adaptation. En effet, des informations inférées à partir d'événements considérés comme externes par l'un des systèmes lorsqu'ils sont séparés pourraient être déterminées par la simple évaluation d'une propriété ou d'un attribut dans le cas où ces systèmes sont assemblés. Cet assemblage pourrait aussi amener un changement des priorités de certaines opérations. En effet, le freinage d'urgence d'un véhicule autonome en cas de collision imminente peut être plus important que la garantie de la précision de son positionnement. La composition de politiques d'adaptation pourrait permettre une réponse, du moins partielle, à ce type de problématique. On pourrait aussi envisager la mise en place de "méta-politiques" d'adaptation responsables de la gestion des politiques d'adaptation de plus bas niveau.

Implémentation

Les perspectives d'évolution de notre implémentation concernent la mise en place de l'évaluation décentralisée et l'utilisation d'un nouveau modèle de boucle pour nos contrôleurs.

Implémentation de l'évaluation décentralisée Une piste d'évolution de notre implémentation consiste à mettre en place la méthode d'évaluation décentralisée de propriétés temporelles et de l'intégrer dans la gestion des politiques d'adaptation. Pour l'instant, dans le cadre de l'évaluation décentralisée de propriétés temporelles, nous avons considéré des moniteurs correspondant à des composants se trouvant au même niveau architectural (i,e., des composants "frères" qui sont des sous-composants d'un même composant composite). C'est pourquoi nous envisageons de déléguer une partie de l'activité des moniteurs des composants composites aux moniteurs de leurs sous-composants.

Utilisation de boucles MAPE Les contrôleurs de notre implémentation ont été développés en suivant le modèle de la boucle de contrôle [Dobson et al., 2006, Cheng et al., 2009, de Lemos et al., 2013] qui est composée de quatre activités : collecte, analyse, décision et action. Nous envisageons de réécrire nos contrôleurs en suivant le modèle de boucle MAPE (voir section 1.2.3) qui est composé de cinq éléments, dont quatre qui effectuent des tâches similaires à celles des activités de collecte, d'analyse,

de décision et d'action du modèle de la boucle de contrôle. La différence fondamentale réside dans le rôle du cinquième élément : le gestionnaire de connaissance (*Knowledge manager*) qui est un élément contenant des informations pouvant être consultées et mises à jours par tous les éléments de la boucle MAPE. En suivant le modèle de boucle MAPE, il pourrait être possible de partager les informations du gestionnaire de connaissance entre les différents contrôleurs. Cela aurait pour effet de simplifier la synchronisation et la communication entre les contrôleurs. De plus, cela réduirait le nombre de requêtes faites par les contrôleurs auprès du système à composants pour obtenir sa configuration courante.

Passage à l'échelle et test

Les perspectives suivantes concernent le passage à l'échelle et le test.

Passage à l'échelle Le passage à l'échelle en termes de nombre de composants et de reconfigurations reste à étudier. Il serait intéressant d'expérimenter nos propositions sur des études de cas portant sur des systèmes ayant une taille importante, voire sur des systèmes paramétrés ou infinis. Des travaux dans ce sens sont en cours pour le modèle et l'environnement BIP [Falcone et al., 2017]. En effet l'évaluation décentralisée devrait être performante dans le cas de systèmes comprenant un grand nombre de composants pour l'évaluation de propriétés dont les événements atomiques sont distribués sur une faible proportion des composants.

Test Des techniques de fuzzing comportemental hors-ligne (*offline behavioural fuzzing*) pour le test de politiques d'adaptation ont déjà été mises en place. L'intégration du fuzzing en-ligne (*online fuzzing*) pourrait permettre la génération de cas de test à l'exécution en fonction de politiques de fuzzing (*fuzzy policies*). Ces politiques permettraient de générer nos cas de test lors de l'exécution de séquences spécifiques de configurations en fonction de conditions prédéterminées.

Systèmes cyber-physiques et micromécatroniques

Pour finir, nous présentons les perspectives d'intégration de nos contributions au sein des systèmes cyber-physiques et micromécatroniques.

Systèmes cyber-physiques Les systèmes cyber-physiques, ou CPS (*Cyber-Physical Systems*) peuvent être vus comme des réseaux de systèmes embarqués où un grand nombre de composants logiciels sont déployés au sein d'un environnement physique [Du et al., 2016, Lee, 2008]. Lorsque ces systèmes sont adaptables, ils peuvent évoluer tout en continuant à fonctionner correctement quand leur environnement est modifié. Dans ce contexte, nos contributions sur le contrôle des systèmes à composants au moyen de politiques d'adaptation basées sur des propriétés temporelles peuvent trouver des applications directes. En plus de ces contributions, les perspectives concernant les reconfigurations concurrentes et la combinaison de politiques d'adaptation, exposées

ci-dessus, peuvent permettre de palier au manque de méthodologies et d'outils dont souffrent les environnements de développement des CPS [El-Khoury et al., 2013].

Systèmes micromécatroniques La mécatronique [Bishop, 2005] unifie les principes de la mécanique, de l'électronique, du développement logiciel et de la théorie du contrôle pour mettre en place des systèmes automatiques. Lorsque de tels systèmes sont miniaturisés, comme, par exemple, les robots modulaires utilisés dans [Piranda et al., 2016], on parle de micromécatronique. Ces robots modulaires auto-reconfigurables ou MSR *(Modular Self-reconfigurable Robots*) sont composés de multiples modules capables de communiquer et de collaborer pour atteindre un objectif commun (i.e., l'objectif du MSR). Ainsi, la forme d'un MSR change lorsque les modules qui le composent se déplacent. Un tel déplacement coordonné de ces modules correspond à une (auto-)reconfiguration du MSR. Dans le cas où tous les modules se déplacent sur le même plan, on parle de surface intelligente (*smart surface*) ; si les déplacements se font en trois dimensions on parle de matière intelligente (*smart matter*). Les MSR peuvent être appréhendés comme des systèmes à composants auxquels nous pouvons appliquer directement nos contributions sur l'évaluation décentralisées de propriétés temporelles. Cela simplifierait les algorithmes distribués utilisés pour l'(auto-)reconfiguration des MSR en permettant l'évaluation décentralisée de propriétés décrivant la progression du système vers son objectif. Il est aussi envisageable d'utiliser des politiques d'adaptation en remplacement ou en complément de tels algorithmes.

BIBLIOGRAPHIE

[Ait Wakrime, 2015] Ait Wakrime, A. (2015). **A component-based approach for interactive visual analysis of numerical simulation results**. Theses, Université d'Orléans.

[Andrade et al., 2002] Andrade, H. A., et Sanders, B. (2002). **An approach to compositional model checking.** Dans *ipdps*.

[Bacchus et al., 1998] Bacchus, F., et Kabanza, F. (1998). **Planning for temporally extended goals**. *Annals of Mathematics and Artificial Intelligence*, 22(1) :5–27.

[Baille et al., 1999] Baille, G., Garnier, P., Mathieu, H., et Pissard-Gibollet, R. (1999). **Le cycab de l'INRIA Rhône-Alpes**. Rapport technique RT-0229, INRIA.

[Barros et al., 2007] Barros, T., Cansado, A., Madelaine, E., et Rivera, M. (2007). **Model-checking distributed components : The vercors platform**. *Electronic Notes in Theoretical Computer Science*, 182 :3–16.

[Basu et al., 2006] Basu, A., Bozga, M., et Sifakis, J. (2006). **Modeling heterogeneous real-time components in bip**. Dans *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. Ieee.

[Bauer et al., 2008] Bauer, A., et others (2008). **The theory and practice of runtime reflection : A model-based framework for dynamic analysis of distributed reactive systems**. VDM Verlag.

[Bauer et al., 2012] Bauer, A., et Falcone, Y. (2012). **Decentralised ltl monitoring**. Dans *International Symposium on Formal Methods*, pages 85–100. Springer.

[Bauer et al., 2010] Bauer, A., Leucker, M., et Schallhart, C. (2010). **Comparing ltl semantics for runtime verification**. *Journal of Logic and Computation*, 20(3) :651–674.

[Beisiegel et al., 2007] Beisiegel, M., Blohm, H., Booz, D., Edwards, M., Hurley, O., et others (2007). **Service component architecture, assembly model specification**. *SCA Version*, 1 :15.

[Bérard et al., 2013] Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., et Schnoebelen, P. (2013). **Systems and software verification : model-checking techniques and tools**. Springer Science & Business Media.

[Bergner et al., 1998] Bergner, K., Rausch, A., et Sihling, M. (1998). **Componentware–the big picture**. Dans *20th ICSE Workshop on Component-based Software Engineering*.

[Bernot et al., 1991] Bernot, G., Gaudel, M. C., et Marre, B. (1991). **Software testing based on formal specifications : a theory and a tool**. *Software Engineering Journal*, 6(6) :387–405.

[Bertolino, 2007] Bertolino, A. (2007). **Software testing research : Achievements, challenges, dreams**. Dans *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society.

[Beugnard et al., 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., et Watkins, D. (1999). **Making components contract aware**. *Computer*, 32(7) :38–45.

[Bishop, 2005] Bishop, R. H. (2005). **Mechatronics : an introduction**. CRC Press.

[Bloom et al., 1995] Bloom, B., Istrail, S., et Meyer, A. R. (1995). **Bisimulation can't be traced**. *Journal of the ACM (JACM)*, 42(1) :232–268.

[Boyer et al., 2013] Boyer, F., Gruber, O., et Pous, D. (2013). **Robust reconfigurations of component assemblies**. Dans *Proceedings of the 2013 International Conference on Software Engineering*, pages 13–22. IEEE Press.

[Bruneton et al., 2004] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., et Stefani, J.-B. (2004). **An open component model and its support in java**. Dans *International Symposium on Component-based Software Engineering*, pages 7–22. Springer.

[Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., et Stefani, J.-B. (2006). **The fractal component model and its support in java**. *Software : Practice and Experience*, 36(11-12) :1257–1284.

[Cazzola et al., 1998] Cazzola, W., Savigni, A., Sosio, A., et Tisato, F. (1998). **Architectural reflection : Bridging the gap between a running system and its architectural specification**. Dans *In proceedings of 6th Reengineering Forum (REF'98*. Citeseer.

[Chauvel et al., 2009] Chauvel, F., Barais, O., Plouzeau, N., Borne, I., et Jézéquel, J.-M. (2009). **Composition et expression qualitative de politiques d'adaptation pour les composants fractal**. Dans *Actes des Journées nationales du GDR GPL 2009*.

[Cheng et al., 2009] Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., et Whittle, J. (2009). **Software engineering for self-adaptive systems : A research roadmap**. Dans Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., et Magee, J., éditeurs, *Software Engineering for Self-Adaptive Systems*, volume 5525 de *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg.

[Chouali et al., 2010] Chouali, S., Mouelhi, S., et Mountassir, H. (2010). **Assembly of components based on interface automata and {UML} component model**. Dans *CAL'10, 4e Conf. Francophone sur les Architectures Logicielles*, pages 73–85.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., et Sistla, A. P. (1986). **Automatic verification of finite-state concurrent systems using temporal logic specifications**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., et Peled, D. (1999). **Model checking**. MIT press.

[Collet et al., 2007] Collet, P., Malenfant, J., Ozanne, A., et Rivierre, N. (2007). **Composite contract enforcement in hierarchical component systems**. Dans *International Conference on Software Composition*, pages 18–33. Springer.

[Collet et al., 2005] Collet, P., Rousseau, R., Coupaye, T., et Rivierre, N. (2005). **A contracting system for hierarchical components**. Dans *International Symposium on Component-Based Software Engineering*, pages 187–202. Springer.

[Council et al., 2001] Council, W., et Heineman, G. (2001). **Component–based software engineering : Putting the pieces together**. *Addison Weysley*.

[Dahl et al., 1972] Dahl, O.-J., Dijkstra, E. W., et Hoare, C. A. R. (1972). **Structured programming**. Academic Press Ltd.

[David, 2005] David, P.-C. (2005). **Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation**. PhD thesis, Université de Nantes.

[David et al., 2009] David, P.-C., Ledoux, T., Léger, M., et Coupaye, T. (2009). **Fpath and fscript : Language support for navigation and reliable reconfiguration of fractal architectures**. *annals of telecommunications-annales des télécommunications*, 64(1-2) :45–63.

[de Lemos et al., 2013] de Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N. M., Vogel, T., Weyns, D., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Göschka, K. M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Smith, D. B., Sousa, J. P., Tahvildari, L., Wong, K., et Wuttke, J. (2013). **Software engineering for self-adaptive systems : A second research roadmap**. Dans de Lemos, R., Giese, H., Müller, H. A., et Shaw, M., éditeurs, *Software Engineering for Self-Adaptive Systems II*, volume 7475 de *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg.

[De Nicola et al., 1995] De Nicola, R., et Vaandrager, F. (1995). **Three logics for branching bisimulation**. *Journal of the ACM (JACM)*, 42(2) :458–487.

[Delaval et al., 2010] Delaval, G., et Rutten, E. (2010). **Reactive model-based control of reconfiguration in the fractal component-based model**. Dans *International Symposium on Component-Based Software Engineering*, pages 93–112. Springer.

[DeMichiel et al., 2006] DeMichiel, L., et Keith, M. (2006). **Enterprise javabeans specification, version 3.0 : Ejb core contracts and requirements**. Rapport technique JSR-220, Sun Microsystems.

[Dijkstra, 1970] Dijkstra, E. W. (1970). **Notes on structured programming**. Rapport technique 70-WSK03, Technological University, Department of Mathematics.

[Dijkstra, 1975] Dijkstra, E. W. (1975). **Guarded commands, nondeterminacy and formal derivation of programs**. *Communications of the ACM*, 18(8) :453–457.

[Dobson et al., 2006] Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., et Zambonelli, F. (2006). **A survey of autonomic communications**. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2) :223–259.

[Dormoy, 2011] Dormoy, J. (2011). **Contributions à la spécificification et à la vérification des reconfigurations dynamiques dans les systèmes à composants**. PhD thesis, LIFC, Université de Franche-Comté.

[Dormoy et al., 2010] Dormoy, J., et Kouchnarenko, O. (2010). **Event-based adaptation policies for fractal components**. Dans *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*, pages 1–8. IEEE.

[Dormoy et al., 2011] Dormoy, J., Kouchnarenko, O., et Lanoix, A. (2011). **Runtime verification of temporal patterns for dynamic reconfigurations of components**. Dans *International Workshop on Formal Aspects of Component Software*, pages 115–132. Springer.

[Dormoy et al., 2012a] Dormoy, J., Kouchnarenko, O., et Lanoix, A. (2012a). **Using temporal logic for dynamic reconfigurations of components**. Dans Barbosa, L., et Lumpe, M., éditeurs, *FACS*, volume 6921 de *LNCS*, pages 200–217. Springer Berlin Heidelberg.

[Dormoy et al., 2012b] Dormoy, J., Kouchnarenko, O., et Lanoix, A. (2012b). **When structural refinement of components keeps temporal properties over reconfigurations**. Dans *International Symposium on Formal Methods*, pages 171–186. Springer.

[D'Souza et al., 1998] D'Souza, D. F., et Wills, A. C. (1998). **Objects, components, and frameworks with UML : the catalysis approach**. Addison-Wesley Longman Publishing Co., Inc.

[Du et al., 2016] Du, D., Huang, P., Jiang, K., Mallet, F., et Yang, M. (2016). **Marte/pccsl : Modeling and refining stochastic behaviors of cpss with probabilistic logical clocks**. Dans *FACS 2016-The 13th International Conference on Formal Aspects of Component Software*.

[Dubois et al., 2006] Dubois, C., Hardin, T., et Donzeau-Gouge, V. (2006). **Building certified components within focal**. *Trends in Functional Programming*, 5 :33–48.

[Dwyer et al., 1999] Dwyer, M. B., Avrunin, G. S., et Corbett, J. C. (1999). **Patterns in property specifications for finite-state verification**. Dans *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE.

[Eckerson, 1995] Eckerson, W. W. (1995). **Three tier client/server architectures : achieving scalability, performance, and efficiency in client/server applications**. *Open Information Systems*, 3(20) :46–50.

[El-Khoury et al., 2013] El-Khoury, J., Asplund, F., Biehl, M., Loiret, F., et Törngren, M. (2013). **A roadmap towards integrated cps development environments**. Dans *1st Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering*.

[Emerson et al., 1986] Emerson, E. A., et Halpern, J. Y. (1986). **"sometimes" and "not never" revisited : on branching versus linear time temporal logic**. *Journal of the ACM (JACM)*, 33(1) :151–178.

[Ernst et al., 2007] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., et Xiao, C. (2007). **The daikon system for dynamic detection of likely invariants**. *Science of Computer Programming*, 69(1) :35–45.

[Falcone et al., 2008] Falcone, Y., Fernandez, J.-C., et Mounier, L. (2008). **Synthesizing enforcement monitors wrt. the safety-progress classification of properties**. Dans *International Conference on Information Systems Security*, pages 41–55. Springer.

[Falcone et al., 2017] Falcone, Y., et Jaber, M. (2017). **Fully automated runtime enforcement of component-based systems with formal and sound recovery**. *International Journal on Software Tools for Technology Transfer*, 19(3) :341–365.

[Fassino et al., 2002] Fassino, J.-P., Stefani, J.-B., Lawall, J. L., et Muller, G. (2002). **Think : A software framework for component-based operating system kernels.** Dans *USENIX Annual Technical Conference, General Track*, pages 73–86.

[Garavel et al., 2011] Garavel, H., Lang, F., Mateescu, R., et Serwe, W. (2011). **Cadp 2010 : A toolbox for the construction and analysis of distributed processes**. Dans *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387. Springer.

[Garavel et al., 2013] Garavel, H., Lang, F., Mateescu, R., et Serwe, W. (2013). **Cadp 2011 : a toolbox for the construction and analysis of distributed processes**. *International Journal on Software Tools for Technology Transfer*, 15(2) :89–107.

[Garavel et al., 2007] Garavel, H., Mateescu, R., Lang, F., et Serwe, W. (2007). **Cadp 2006 : A toolbox for the construction and analysis of distributed processes**. Dans *International Conference on Computer Aided Verification*, pages 158–163. Springer.

[Ghamarian et al., 2012] Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E., et Zimakova, M. (2012). **Modelling and analysis using GROOVE**. *Int. J. on Software Tools for Technology Transfer*, 14(1) :15–40.

[Gonnord et al., 2009] Gonnord, L., et Babau, J.-P. (2009). **Quantity of resource properties expression and runtime assurance for embedded systems**. *AICCSA*.

[Goodenough et al., 1975] Goodenough, J. B., et Gerhart, S. L. (1975). **Toward a theory of test data selection**. *IEEE Transactions on software Engineering*, 1(2) :156–173.

[Hamilton, 1988] Hamilton, A. G. (1988). **Logic for mathematicians**. Cambridge University Press.

[Havelund et al., 2005] Havelund, K., et Goldberg, A. (2005). **Verify your runs**. Dans *Working Conference on Verified Software : Theories, Tools, and Experiments*, pages 374–383. Springer.

[Hoare, 1969] Hoare, C. A. R. (1969). **An axiomatic basis for computer programming**. *Communications of the ACM*, 12(10) :576–580.

[Hoare, 1978] Hoare, C. A. R. (1978). **Communicating sequential processes**. Dans *The origin of concurrent programming*, pages 413–443. Springer.

[Howden, 1976] Howden, W. E. (1976). **Reliability of the path analysis testing strategy**. *IEEE Transactions on Software Engineering*, 2(3) :208–215.

[IBM Corporation, 2006] IBM Corporation (2006). **An architectural blueprint for autonomic computing**. *IBM White Paper*, 31.

[Jackson, 2012] Jackson, D. (2012). **Software Abstractions : logic, language, and analysis**. MIT press.

[Jantsch, 2004] Jantsch, A. (2004). **Modeling embedded systems and SoCs : concurrency and time in models of computation**. Morgan Kaufmann.

[Kephart et al., 2003] Kephart, J. O., et Chess, D. M. (2003). **The vision of autonomic computing**. *Computer*, 36(1) :41–50.

[Kiczales et al., 1991] Kiczales, G., Des Rivieres, J., et Bobrow, D. G. (1991). **The art of the metaobject protocol**. MIT press.

[Kim et al., 2002] Kim, M., Lee, I., Sammapun, U., Shin, J., et Sokolsky, O. (2002). **Monitoring, checking, and steering of real-time systems**. *Electronic Notes in Theoretical Computer Science*, 70(4) :95–111.

[Kouchnarenko et al., 2014a] Kouchnarenko, O., et Weber, J.-F. (2014a). **Adapting component-based systems at runtime via policies with temporal patterns**. Dans *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, volume 8348 de *Lecture Notes in Computer Science*, pages 234–253. Springer.

[Kouchnarenko et al., 2014b] Kouchnarenko, O., et Weber, J.-F. (2014b). **Decentralised evaluation of temporal patterns over component-based systems at runtime**. Dans *Formal Aspects of Component Software - 11th International Symposium, FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers*, volume 8997 de *Lecture Notes in Computer Science*, pages 108–126. Springer.

[Kouchnarenko et al., 2015] Kouchnarenko, O., et Weber, J.-F. (2015). **Practical analysis framework for component systems with dynamic reconfigurations**. Dans *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, volume 9407 de *Lecture Notes in Computer Science*, pages 287–303. Springer.

[Kouchnarenko et al., 2016] Kouchnarenko, O. B., et Weber, J.-F. (2016). **Component-based systems reconfigurations using graph grammars**. *Modelirovanie i Analiz Informatsionnykh Sistem (article publié en Russe)*, 23(6) :804–825.

[Kozen, 1983] Kozen, D. (1983). **Results on the propositional μ-calculus**. *Theoretical computer science*, 27(3) :333–354.

[Kruchten et al., 2006] Kruchten, P., Obbink, H., et Stafford, J. (2006). **The past, present, and future for software architecture**. *IEEE software*, 23(2) :22–30.

[Lamport, 1983] Lamport, L. (1983). **Specifying concurrent program modules**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2) :190–222.

[Lamport, 1996] Lamport, L. (1996). **Refinement in state-based formalisms**. *Digital Equipment Corportation*.

[Lanoix, 2005] Lanoix, A. (2005). **Systèmes à composants synchronisés : contributions à la vérification compositionnelle du raffinement et des propriétés**. PhD thesis, Université de Franche-Comté.

[Lanoix et al., 2011] Lanoix, A., Dormoy, J., et Kouchnarenko, O. (2011). **Combining proof and model-checking to validate reconfigurable architectures**. *Electronic Notes in Theoretical Computer Science*, 279(2) :43–57.

[Le Métayer, 1998] Le Métayer, D. (1998). **Describing software architecture styles using graph grammars**. *IEEE Transactions on Software Engineering*, 24(7) :521–533.

[Lee, 2008] Lee, E. A. (2008). **Cyber physical systems : Design challenges**. Dans *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE.

[Léger, 2009] Léger, M. (2009). **Fiabilité des reconfigurations dynamiques dans les architectures à composants**. PhD thesis, École Nationale Supérieure des Mines de Paris.

[Léger et al., 2010] Léger, M., Ledoux, T., et Coupaye, T. (2010). **Reliable dynamic reconfigurations in a reflective component model**. Dans *Component-Based Software Engineering*, pages 74–92. Springer.

[Ligatti et al., 2005] Ligatti, J., Bauer, L., et Walker, D. (2005). **Edit automata : Enforcement mechanisms for run-time security policies**. *International Journal of Information Security*, 4(1-2) :2–16.

[Ligatti et al., 2009] Ligatti, J., Bauer, L., et Walker, D. (2009). **Run-time enforcement of nonsafety policies**. *ACM Transactions on Information and System Security (TISSEC)*, 12(3) :19.

[Manna et al., 1992] Manna, Z., et Pnueli, A. (1992). **The Temporal Logic of Reactive and Concurrent Systems**. Springer-Verlag New York, Inc., New York, NY, USA.

[Merle et al., 2011] Merle, P., Rouvoy, R., et Seinturier, L. (2011). **A reflective platform for highly adaptive multi-cloud systems**. Dans *Adaptive and Reflective Middleware on Proceedings of the International Workshop*, pages 14–21. ACM.

[Merle et al., 2008] Merle, P., et Stefani, J.-B. (2008). **A formal specification of the Fractal component model in Alloy**. Rapport technique RR-6721, INRIA.

[Meyer, 1988] Meyer, B. (1988). **Object-oriented software construction**, volume 2. Prentice hall New York.

[Microsoft Corporation, 1995] Microsoft Corporation (1995). **The component object model specification**. Rapport technique, Microsoft Corporation and Digital Equipment Corporation.

[Milner, 1989] Milner, R. (1989). **Communication and concurrency**, volume 84. Prentice hall New York etc.

[Misra et al., 1981] Misra, J., et Chandy, K. M. (1981). **Proofs of networks of processes**. *IEEE transactions on software engineering*, 7(4) :417–426.

[Naur et al., 1969] Naur, P., et Randell, B. (1969). **Software engineering : Report of a conference sponsored by the nato science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, nato**.

[Nierstrasz, 1995] Nierstrasz, O. (1995). **Research topics in software composition.** Dans *LMO*, volume 95, pages 193–204.

[Oreizy et al., 1998a] Oreizy, P., et others (1998a). **Issues in modeling and analyzing dynamic software architectures**. Dans *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis*, pages 54–57.

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

[Oussalah, 2005] Oussalah, M. (2005). **Ingénierie des composants : concepts, techniques et outils**. Vuibert informatique.

[Papazoglou et al., 2007] Papazoglou, M. P., Traverso, P., Dustdar, S., et Leymann, F. (2007). **Service-oriented computing : State of the art and research challenges**. *Computer*, 40(11).

[Pedrycz, 1993] Pedrycz, W. (1993). **Fuzzy control and fuzzy systems (2nd**. Research Studies Press Ltd.

[Piranda et al., 2016] Piranda, B., et Bourgeois, J. (2016). **A distributed algorithm for reconfiguration of lattice-based modular self-reconfigurable robots**. Dans *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 1–9. IEEE.

[Rayner et al., 2005] Rayner, M., Hockey, B. A., Chatzichrisafis, N., et Farrell, K. (2005). **Omg unified modeling language specification**. Dans *Version 1.3,© 1999 Object Management Group, Inc*. Citeseer.

[Rogin et al., 2008] Rogin, F., Klotz, T., Fey, G., Drechsler, R., et Rulke, S. (2008). **Automatic generation of complex properties for hardware designs**. Dans *2008 Design, Automation and Test in Europe*, pages 545–548. IEEE.

[Romero, 2011] Romero, D. (2011). **Context as a Resource : A Service-Oriented Approach for Context-Awareness**. Theses, Université des Sciences et Technologie de Lille - Lille I.

[Sametinger, 1997] Sametinger, J. (1997). **Software engineering with reusable components**. Springer Science & Business Media.

[Schneider, 2000] Schneider, F. B. (2000). **Enforceable security policies**. *ACM Transactions on Information and System Security (TISSEC)*, 3(1) :30–50.

[Schneider et al., 2013] Schneider, M., Großmann, J., Schieferdecker, I., et Pietschker, A. (2013). **Online model-based behavioral fuzzing**. Dans *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 469–475. IEEE.

[Schneider et al., 2012] Schneider, M., Großmann, J., Tcholtchev, N., Schieferdecker, I., et Pietschker, A. (2012). **Behavioral fuzzing operators for uml sequence diagrams**. Dans *International Workshop on System Analysis and Modeling*, pages 88–104. Springer.

[Seinturier et al., 2009] Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., et Stefani, J.-B. (2009). **Reconfigurable sca applications with the frascati platform**. Dans *Services Computing, 2009. SCC'09. IEEE International Conference on*, pages 268–275. IEEE.

[Seinturier et al., 2012] Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., et Stefani, J.-B. (2012). **A component-based middleware platform for reconfigurable service-oriented architectures**. *Software : Practice and Experience*, 42(5) :559–583.

[Shapiro, 1969] Shapiro, E. B. (1969). **Network timetable**. Rapport technique.

[Simonot et al., 2008] Simonot, M., et Aponte, M.-V. (2008). **Une approche formelle de la reconfiguration dynamique**. *L'Objet*, 14(4) :73–102.

[Smith, 1984] Smith, B. C. (1984). **Reflection and semantics in lisp**. Dans *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35. ACM.

[Szyperski, 1995] Szyperski, C. (1995). **Component software : beyond object-oriented programming. 1998**. *Harlow, England : Addison-Wesley*.

[Takanen et al., 2008] Takanen, A., Demott, J. D., et Miller, C. (2008). **Fuzzing for software security testing and quality assurance**. Artech House.

[Tamura, 2012] Tamura, G. (2012). **QoS-CARE : A Reliable System for Preserving QoS Contracts through Dynamic Reconfiguration**. PhD thesis, Université des Sciences et Technologie de Lille-Lille I ; Universidad de Los Andes.

[Tiberghien et al., 2010] Tiberghien, A., Merle, P., et Seinturier, L. (2010). **Specifying self-configurable component-based systems with fractoy**. Dans *International Conference on Abstract State Machines, Alloy, B and Z*, pages 91–104. Springer.

[Trentelman et al., 2002] Trentelman, K., et Huisman, M. (2002). **Extending JML specifications with temporal logic**. Dans *International Conference on Algebraic Methodology and Software Technology*, pages 334–348. Springer.

[Tretmans, 1996] Tretmans, J. (1996). **Test generation with inputs, outputs and repetitive quiescence**. *Software—Concepts and Tools*, 46(TR-CTIT-96-26).

[Vardi, 1988] Vardi, M. Y. (1988). **A temporal fixpoint calculus**. Dans *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 250–259. ACM.

[Vyatkin, 2011] Vyatkin, V. (2011). **Function blocks for embedded and distributed control systems design**. *ISA, New Zealand*.

[Wang et al., 2001] Wang, N., Schmidt, D. C., et O'Ryan, C. (2001). **Overview of the corba component model**. Dans *Component-Based Software Engineering*, pages 557–571. Addison-Wesley Longman Publishing Co., Inc.

[Warnier et al., 1974] Warnier, J.-D., et Warnier, J. (1974). **Logical construction of programs (LCP)**. Stenfert Kroese.

[Weber, 2016] Weber, J.-F. (2016). **Tool support for fuzz testing of component-based system adaptation policies**. Dans *Formal Aspects of Component Software - 13th*

International Conference, FACS 2016, Besançon, France, October 19-21, 2016, Revised Selected Papers, volume 10231 de *Lecture Notes in Computer Science*, pages 231–237. Springer.

[Weyuker, 1998] Weyuker, E. J. (1998). **Testing component-based software : A cautionary tale**. *IEEE software*, 15(5) :54–59.

TABLE DES FIGURES

LISTE DES TABLES

LISTE DES DÉFINITIONS

LISTE DES THÉORÈMES

LISTE DES COROLAIRES

LISTE DES PROPOSITIONS

LISTE DES EXEMPLES

V

ANNEXES

A

SÉMANTIQUE DES RECONFIGURATIONS PRIMITIVES

En s'inspirant de [Léger, 2009] nous utilisons les notations de la section 3.2 pour établir une sémantique précise des opérations primitives de reconfiguration au moyen de pré-conditions et post-conditions.

La sémantique que nous proposons ci-dessous se compose des neufs actions suivantes :

- `new` / `destroy`, pour l'instanciation et la destruction de composants,
- `add` / `remove`, pour l'ajout et le retrait d'un composant dans une configuration donnée,
- `bind` / `unbind`, pour la connexion et la déconnexion d'interfaces,
- `start` / `stop`, pour le démarrage et l'arrêt d'un composant donné,
- `update`, pour la mise à jour d'un paramètre.

A.1/ NOTATIONS ET CONVENTIONS

Soit $comp$ un composant n'ayant ni parent ni sous-composants qui est (par défaut) arrêté, i.e., son état est $stopped$. Un tel composant est dit primitif et peut être défini par sa configuration $c_{comp} = \langle Elem_{comp}, Rel_{comp} \rangle$ au sens des définitions 27 à 29 où $Elem_{comp}$ et Rel_{comp} sont définis comme suit :

Sommaire

- $Elem_{comp}$ est composé des éléments architecturaux suivants :
  - $Components_{comp} = \{comp\}$,
  - $IProvided_{comp}$ contient les interfaces fournies du composant $comp$,
  - $IRequired_{comp}$ contient les interfaces requises du composant $comp$,
  - $Parameters_{comp}$ contient les paramètres du composant $comp$,
  - $ITypes_{comp}$ contient les types des interfaces (fournies ou requises) du composant $comp$,
  - $PTypes_{comp}$ contient les types des paramètres du composant $comp$,
  - $Contingencies_{comp}$ contient les valeurs possibles de la contingence des interfaces requises, du composant $comp$,
  - $States_{comp}$ contient les valeurs possibles des états du composant $comp$ ($stopped \in States_{comp}$ puisque, par définition, le composant $comp$ est arrêté),
  - $Values_{comp}$ est l'ensemble de toutes les valeurs pouvant être prise par les paramètres de type $t \in PTypes_{comp}$,
- Rel_{comp} est composé des relations architecturales suivantes :
  - $IProvidedType_{comp}$ associe à chaque interface fournie un type d'interface,
  - $IRequiredType_{comp}$ associe à chaque interface requise un type d'interface,
  - $Provider_{comp}$ associe à chaque interface fournie le composant $comp$ auquel elle appartient,
  - $Requirer_{comp}$ associe à chaque interface requise le composant $comp$ auquel elle appartient,
  - $Contingency_{comp}$ à chaque interface requise à la valeur de contingence lui correspondant,
  - $ParameterType_{comp}$ associe à chaque paramètre un type de paramètre,
  - $Definer_{comp}$ associe à chaque paramètre le composant $comp$ auquel il appartient,
  - $Parent_{comp} = \emptyset$,
  - $Descendant_{comp} = \emptyset$,
  - $Binding_{comp} = \emptyset$,
  - $DelegateProv_{comp} = \emptyset$,
  - $DelegateReq_{comp} = \emptyset$,
  - $State_{comp} = \{comp, stopped\}$,
  - $Value_{comp}$ est une fonction totale qui donne la valeur par défaut de chaque paramètre.

Comme dans la section 3.2, afin de simplifier les notations, on considère les simplifications suivantes, où $\uplus$ est l'opérateur d'union disjointe :

- $Interfaces = IProvided \uplus IRequired$,
- $Containers = Interfaces \uplus Parameters$,
- $Container = Provider \uplus Requirer \uplus Definer$,
- $ContainerType = IProvidedType \uplus IRequiredType \uplus ParameterType$,
- $Delegate = DelegateProv \uplus DelegateReq$,
- $hasBinding = Binding \uplus Binding^{-1} \uplus Delegate \uplus Delegate^{-1}$.

A.2/ SÉMANTIQUE DES OPÉRATIONS PRIMITIVES

Nous considérons par la suite une configuration $c = \langle Elem, Rel \rangle$ au sens des définitions 27 à 29. Les opérations primitives peuvent être exprimées en termes de pré-conditions et post-conditions utilisant des propriétés de configuration. On représente les éléments et relations d'une configuration à l'état courant en utilisant les noms d'ensembles ($Elem$) et de relations (Rel), e.g., $Components$ ou $Parent$, tandis que ces mêmes éléments et relations à l'état suivant sont primés, e.g., $Components'$ ou $Parent'$. Lorsqu'un ensemble ne change pas, e.g., $Components' = Components$, il ne sera simplement pas mentionné dans la post-condition.

A.2.1/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `new`

L'opération `new`($comp$), représenté en table A.1, permet d'instancier un nouveau composant $comp$. La pré-condition permet de s'assurer que *a*) $comp$ n'est pas déjà un élément de $Components$ ($comp \notin Components$), *b*) qu'aucune interface ou aucun paramètre de $comp$ ne porte un nom déjà utilisé ($Containers \cap Containers_{comp} = \emptyset$), *c*) que soit $comp$ est le premier composant du système ($Components = \emptyset$), soit les valeurs de contingences et d'états possibles de $comp$ sont les mêmes que celles déjà possibles pour les autres composants du système ($Contingencies = Contingencies_{comp} \wedge States = States_{comp}$) et *d*) que $comp$ possède au moins une interface fournie.

La post-condition consiste à ajouter (union ensembliste) les ensembles et relations de $comp$ à celles du système.

TABLE A.1: Opération primitive `new` : pré/post-conditions

Opération	`new`($comp$)		
Pré-condition	$comp \notin Components \wedge Containers \cap Containers_{comp} = \emptyset$ $\wedge\ (Components = \emptyset \vee (Contingencies = Contingencies_{comp} \wedge States = States_{comp}))$ $\wedge\ IProvided_{comp} \neq \emptyset$		
Post-condition	$Components'$	$=$	$Components \cup Components_{comp} = Components \cup \{comp\}$
	$IProvided'$	$=$	$IProvided \cup IProvided_{comp}$
	$IRequired'$	$=$	$IRequired \cup IRequired_{comp}$
	$Parameters'$	$=$	$Parameters \cup Parameters_{comp}$
	$ITypes'$	$=$	$ITypes \cup ITypes_{comp}$
	$PTypes'$	$=$	$PTypes \cup PTypes_{comp}$
	$Contingencies'$	$=$	$Contingencies \cup Contingencies_{comp}$
	$Values'$	$=$	$Values \cup Values_{comp}$
	$IProvidedType'$	$=$	$IProvidedType \cup IProvidedType_{comp}$
	$IRequiredType'$	$=$	$IRequiredType \cup IRequiredType_{comp}$
	$Provider'$	$=$	$Provider \cup Provider_{comp}$
	$Requirer'$	$=$	$Requirer \cup Requirer_{comp}$
	$Contingency'$	$=$	$Contingency \cup Contingency_{comp}$
	$ParameterType'$	$=$	$ParameterType \cup ParameterType_{comp}$
	$Definer'$	$=$	$Definer \cup Definer_{comp}$
	$State'$	$=$	$State \cup State_{comp} = State \cup \{comp, stopped\}$
	$Value'$	$=$	$Value \cup Value_{comp}$

A.2.2/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `destroy`

L'opération `destroy`($comp$), représenté en table A.2, permet de supprimer un composant $comp$. La pré-condition permet de s'assurer que *a*) $comp$ est un élément de $Components$ ($comp \in Components$), *b*) qu'il soit stoppé ($Components \wedge State(comp) = stopped$), *c*) qu'il n'ait ni parent ni descendant ($\forall c \in Components.(comp, c) \notin Parent \wedge (c, comp) \notin Parent$) et *d*) qu'aucune de ses interfaces ne soit liée ($\forall i, i' \in Interfaces.(i, comp) \in Container \Rightarrow (i, i') \notin hasBinding$). La post-condition consiste à retirer les ensembles et relations de $comp$ de celles du système.

TABLE A.2: Opération primitive `destroy` : pré/post-conditions

Opération	`destroy`($comp$)
Pré-condition	$comp \in Components \wedge State(comp) = stopped$ $\wedge\ (\forall c \in Components.(comp, c) \notin Parent \wedge (c, comp) \notin Parent)$ $\wedge\ (\forall i, i' \in Interfaces.(i, comp) \in Container \Rightarrow (i, i') \notin hasBinding)$
Post-condition	$Components' = Components \backslash \{comp\}$ $IProvided' = \{i \mid \forall c \in Components \backslash \{comp\}.i \in IProvided \wedge (i, c) \in Provider\}$ $IRequired' = \{i \mid \forall c \in Components \backslash \{comp\}.i \in IRequired \wedge (i, c) \in Requirer\}$ $Parameters' = \{p \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge (p, c) \in Definer\}$ $ITypes' = \{t \mid \forall c \in Components \backslash \{comp\}.i \in Interfaces \wedge t \in ITypes \Rightarrow (i, c) \in Container \wedge (i, t) \in ContainerType\}$ $PTypes' = \{t \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge t \in PTypes \Rightarrow (p, c) \in Container \wedge (p, t) \in ContainerType\}$ $Values' = \bigcup_{\{t \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge t \in PTypes \Rightarrow (p,c) \in Container \wedge (p,t) \in ContainerType\}} \{v \mid v \in t\}$ $IProvidedType' = \{(i, t) \mid \forall c \in Components \backslash \{comp\}.i \in IProvided \wedge t \in ITypes \wedge (i, c) \in Provider \wedge (i, t) \in IProvidedType\}$ $IRequiredType' = \{(i, t) \mid \forall c \in Components \backslash \{comp\}.i \in IRequired \wedge t \in ITypes \wedge (i, c) \in Requirer \wedge (i, t) \in IRequiredType\}$ $Provider' = \{(i, c) \mid \forall c \in Components \backslash \{comp\}.i \in IProvided \wedge (i, c) \in Provider\}$ $Requirer' = \{(i, c) \mid \forall c \in Components \backslash \{comp\}.i \in IRequired \wedge (i, c) \in Requirer\}$ $Contingency' = \{(i, k) \mid \forall c \in Components \backslash \{comp\}.k \in Contingencies \wedge i \in IRequired \wedge (i, k) \in Contingency\}$ $ParameterType' = \{(p, t) \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge t \in PTypes \wedge (p, c) \in Definer \wedge (p, t) \in ParameterType\}$ $Definer' = \{(p, c) \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge (p, c) \in Definer\}$ $State' = State \backslash \{comp, stopped\}$ $Value' = \{(p, v) \mid \forall c \in Components \backslash \{comp\}.p \in Parameters \wedge v \in Values \wedge (p, c) \in Definer \wedge (p, v) \in Value\}$

A.2.3/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE add

L'opération add($comp_p, comp_c$), représenté en table A.3, permet d'ajouter un composant $comp_c$ ("c" pour "child") à un composant $comp_p$ ("p" pour "parent"). La pré-condition permet de s'assurer que *a*) $comp_p$ et $comp_c$ soient des éléments distincts de $Components$ ($comp_p, comp_c \in Components$), *b*) que $comp_p$ ne soit pas un descendant de $comp_c$ ($(comp_c, comp_p) \notin Descendant$) afin d'éviter que $Parent$ ne soit une relation cyclique si ou ajoute $\{comp_c, comp_p\}$ dans $Parent$ et que *c*) $comp_p$ n'ait aucun paramètre ($\forall p \in Parameters.(comp_p, p) \notin Definer$).

La post-condition consiste à ajouter $\{(comp_c, comp_p)\}$ à la relation $Parent$; de plus on ajoute à $Descendant$ le couple $(comp_p, comp_c)$ ainsi que les ensembles suivants :

— $\{(c, comp_c) \mid \forall c \in Components, (c, comp_p) \in Descendant\}$, signifiant que tout composant ayant $comp_p$ comme descendant doit aussi avoir $comp_c$ comme descendant ;

— $\{(comp_p, c) \mid \forall c \in Components, (comp_c, c) \in Descendant\}$, afin de s'assurer que tout descendant de $comp_c$ est aussi descendant de $comp_p$;

— $\{(c, c') \mid \forall c, c' \in Components, (c, comp_p) \in Descendant \wedge (comp_c, c') \in Descendant\}$, pour décrire le fait que tout composant ayant $comp_p$ comme descendant doit aussi avoir comme descendants ceux de $comp_c$.

TABLE A.3: Opération primitive add : pré/post-conditions

Opération	add($comp_p, comp_c$)
Pré-condition	$comp_p, comp_c \in Components \wedge comp_c \neq comp_p$ $\wedge (comp_c, comp_p) \notin Descendant \wedge \forall p \in Parameters.(comp_p, p) \notin Definer$
Post-condition	$Parent' = Parent \cup \{(comp_c, comp_p)\}$ $Descendant' = Descendant \cup \{(comp_p, comp_c)\} \cup$ $\{(c, comp_c) \mid \forall c \in Components, (c, comp_p) \in Descendant\} \cup$ $\{(comp_p, c) \mid \forall c \in Components, (comp_c, c) \in Descendant\} \cup$ $\{(c, c') \mid \forall c, c' \in Components, (c, comp_p) \in Descendant$ $\wedge (comp_c, c') \in Descendant\}$

A.2.4/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE remove

L'opération remove($comp_p, comp_c$), représentée en table A.4, permet d'enlever un composant $comp_c$ ("c" pour "child") d'un composant $comp_p$ ("p" pour "parent"). La pré-condition permet de s'assurer que *a*) $comp_p$ et $comp_c$ soient des éléments de $Components$ ($comp_p, comp_c \in Components$), *b*) que $comp_p$ ainsi que tous ses éventuels descendants soient stoppés ($(comp_p, stopped) \in State \wedge (\forall c \in Components.(comp_p, c) \in Descendant \Rightarrow (c, stopped) \in State)$) et *c*) que si un composant a une interface liée à une interface de $comp_c$, ce composant n'a pas $comp_p$ pour parent[1] et n'est ni $comp_c$[2] ni $comp_p$[3] ($\forall i, i' \in Interfaces, c \in Components.(i, c) \in Container \wedge (i', comp_c) \in Container \wedge (i, i') \in hasBinding \Rightarrow c \notin \{comp_p, comp_c\} \wedge (c, comp_p) \notin Parent$).

1. Afin d'éviter les liens entres interfaces de composants frères ayant $comp_p$ comme parent.
2. Ceci permet d'éviter les liens entre deux interfaces de $comp_c$.
3. Cela permet d'éviter les liens de délégations entres interfaces de $comp_c$ et $comp_p$.

La post-condition consiste à retirer $\{(comp_c, comp_p)\}$ de $Parent$; de plus on enlève de $Descendant$ le couple $(comp_p, comp_c)$ (s'il n'existe pas de composant à la fois descendant de $comp_p$ et admettant $comp_c$ pour descendant) ainsi que certains autres couples de composants afin de s'assurer que les relations de descendance décrites entre les composants par la relation $Descendant$ soient conformes aux relations de parenté décrites par la relation $Parent$. Ainsi,nous excluons les ensembles suivants (nommés r_0, r_c, r_p et r_2 pour plus de lisibilité) de la relation $Descendant$:

— $r_0 = \{(comp_p, comp_c) \mid \forall c \in Components.(comp_p, c) \notin Descendant \vee (c, comp_c) \notin Descendant\}$ qui contient $(comp_c, comp_p)$ s'il n'existe pas de composant à la fois descendant de $comp_p$ et admettant $comp_c$ pour descendant ou est vide sinon,

— $r_c = \{(c, comp_c) \mid \forall c \in Components.(c, comp_p) \in Descendant \wedge (comp_c, c) \notin Parent \wedge (\forall c' \in Components.(comp_c, c') \in Parent \wedge (c, c') \in Descendant \Rightarrow c' = comp_p)\}$, pour s'assurer que tout composant c ayant $comp_p$ pour descendant n'aura plus $comp_c$ comme descendant sauf si

  — c est un parent de $comp_c$; ou

  — $comp_p$ n'est pas l'unique descendant de c parent de $comp_c$ et si c n'est pas un parent de $comp_c$;

— $r_p = \{(comp_p, c) \mid \forall c \in Components.(comp_c, c) \in Descendant \wedge (c, comp_p) \notin Parent \wedge (\forall c' \in Components.(c', comp_p) \in Parent \wedge (c', c) \in Descendant \Rightarrow c' = comp_c)\}$, pour signifier que tout composant c descendant de $comp_c$ ne soit plus descendant de $comp_p$ sauf si

  — $comp_p$ est aussi son parent [4] ou

  — $comp_c$ n'est pas l'unique composant ayant $comp_p$ pour parent duquel c soit un descendant ;

— $r_2 = \{(c, c') \mid \forall c, c' \in Components.(c, comp_p) \in Descendant \wedge (comp_c, c') \in Descendant \wedge (c', c) \notin Parent \wedge (\forall c'' \in Components.(c', c'') \notin Parent \vee (c'', c) \notin Parent) \wedge (\forall c'', c''' \in Components.(c''', c'') \in Parent \wedge (c, c'') \in Descendant \wedge (c''', c') \in Descendant \Rightarrow (c''', c'') = (comp_c, comp_p) \vee c'' = comp_c \vee c''' = comp_p \vee (comp_c, c'') \in Descendant \vee (c''', comp_p) \in Descendant)\}$, pour établir que tout composant descendant c' de $comp_c$ ne soit plus descendant d'aucun composant c ayant $comp_p$ pour descendant sauf si

  — c est lui-même parent de c',

  — il existe au moins un composant c'' qui est parent de c' et dont c est le parent ou

  — il existe un couple de composants (c'', c''') tel que c'' soit un parent de c''', que c'' et c' soient respectivement des descendants de c et c''' et qu'aucune des conditions de la figure A.1 ne soit remplie. Cette figure représente les différentes organisations architecturales possibles impliquant c'' et c''' pour lesquelles la relation de descendance entre c et c' dépend de la relation de parenté entre $comp_c$ et $comp_p$. Chaque forme en traits pleins représente un composant dont le nom se trouve dans le coin supérieur gauche, tandis que les formes en traits discontinus représentent une imbrication d'un nombre quelconque (pouvant être nul) de composants. Ainsi, les conditions représentée par la figure A.1 sont les suivantes.

    — $(c'', c''') = (comp_p, comp_c)$ (figure A.1(a)),

4. Cas d'un composant partagé c ayant pour parents c' et c'' lorsque c'' est parent de c'.

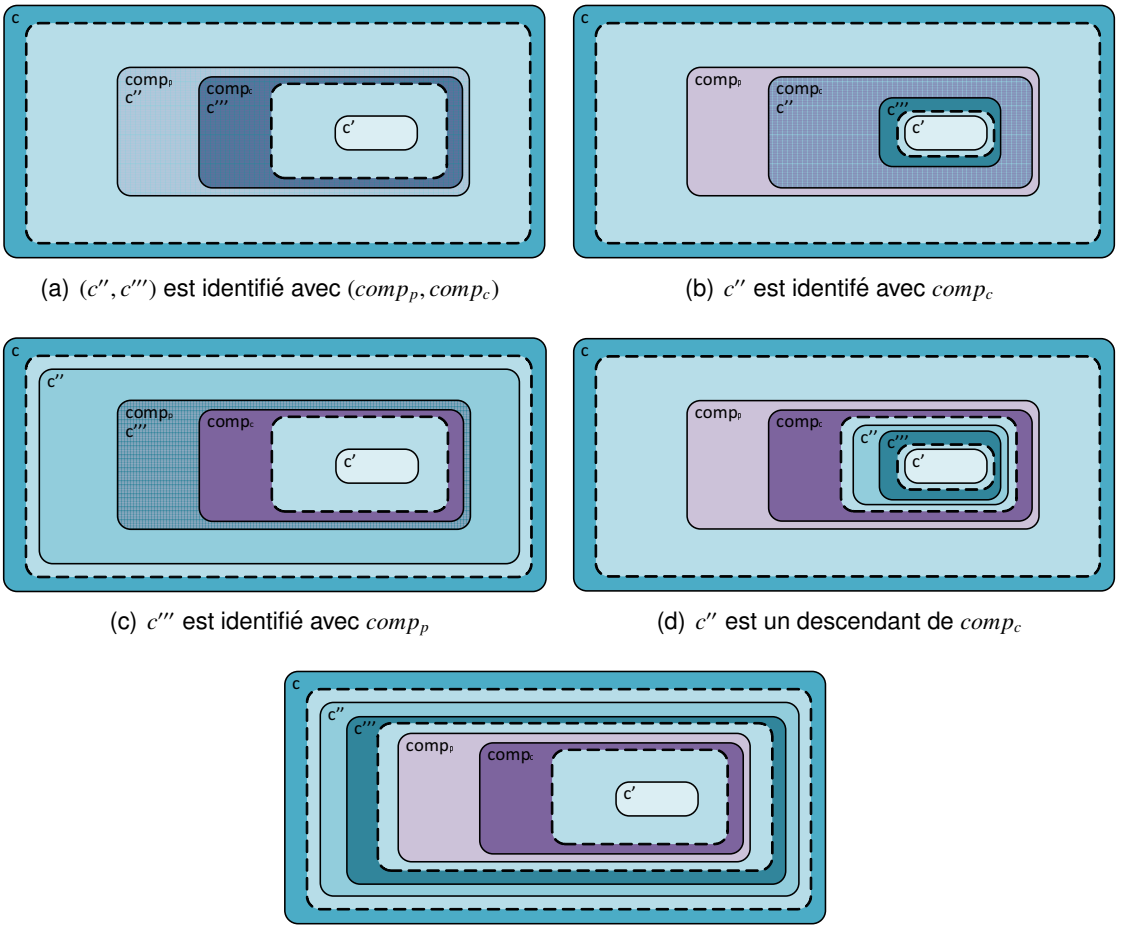
— $c'' = comp_c$ (figure A.1(b)),

— $c''' = comp_p$ (figure A.1(c)),

— $(comp_c, c'') \in Descendant$ (figure A.1(d)) ou que

— $(c''', comp_p) \in Descendant$ (figure A.1(e)).

Intuitivement, le fait qu'il existe un tel couple (c'', c''') ne remplissant aucune des conditions représentées à la figure A.1 signifie qu'il existe une suite $c_0, \ldots, c_n$ où $(c_i, c_{i-1}) \in Parent$, pour $1 \leq i \leq n$, telle que, si $c = c_n$ et $c' = c_0$, $(c_i, c_{i-1}) \neq (comp_c, comp_p)$ pour $1 \leq i \leq n$, c'est-à-dire que la suite $c_0, \ldots, c_n$ basée sur les relations de parenté entre ses termes ne dépend pas de la relation de parenté entre $comp_c$ et $comp_p$.



(a) (c'', c''') est identifié avec $(comp_p, comp_c)$

(b) c'' est identifié avec $comp_c$

(c) c''' est identifié avec $comp_p$

(d) c'' est un descendant de $comp_c$

(e) $comp_p$ est un descendant de c'''

FIGURE A.1 – Cas de figures possibles pour un couple de composants (c, c') dont la relation de descendance dépend de la relation de parenté en $comp_c$ et $comp_p$

Ainsi, la post-condition de l'opération `remove`$(comp_p, comp_c)$ concernant la relation $Descendant$ s'écrit $Descendant' = Descendant \backslash \Big(\{(comp_p, comp_c)\} \cup r_c \cup r_p \cup r_2 \Big)$. La table A.4 représente l'opération `remove`$(comp_p, comp_c)$ permettant d'enlever un composant $comp_c$ ("c" pour "child") d'un composant $comp_p$ ("p" pour "parent").

TABLE A.4: Opération primitive `remove` : pré/post-conditions

Opération	`remove`$(comp_p, comp_c)$
Pré-condition	$comp_p, comp_c \in Components \wedge (comp_p, stopped) \in State$ $\wedge\ (\forall c \in Components.(comp_p, c) \in Descendant \Rightarrow (c, stopped) \in State)$ $\wedge\ \Big(\forall i, i' \in Interfaces, c \in Components.(i, c) \in Container \wedge (i', comp_c) \in Container$ $\wedge\ (i, i') \in hasBinding \Rightarrow c \notin \{comp_p, comp_c\} \wedge (c, comp_p) \notin Parent\Big)$
Post-condition	$Parent' = Parent \backslash \{(comp_c, comp_p)\}$ $Descendant' = Descendant \backslash \Big(\{(comp_p, comp_c) \mid \forall c \in Components.$ $(comp_p, c) \notin Descendant \vee (c, comp_c) \notin Descendant\} \cup$ $\{(c, comp_c) \mid \forall c \in Components.(c, comp_p) \in Descendant \wedge$ $(comp_c, c) \notin Parent \wedge$ $(\forall c' \in Components.(comp_c, c') \in Parent \wedge$ $(c, c') \in Descendant \Rightarrow c' = comp_p)\} \cup$ $\{(comp_p, c) \mid \forall c \in Components.(comp_c, c) \in Descendant \wedge$ $(c, comp_p) \notin Parent \wedge$ $(\forall c' \in Components.(c', comp_p) \in Parent \wedge$ $(c', c) \in Descendant \Rightarrow c' = comp_c)\} \cup$ $\{(c, c') \mid \forall c, c' \in Components.(c, comp_p) \in Descendant \wedge$ $(comp_c, c') \in Descendant \wedge (c', c) \notin Parent \wedge$ $(\forall c'' \in Components.(c', c'') \notin Parent \vee$ $(c'', c) \notin Parent) \wedge$ $(\forall c'', c''' \in Components.(c''', c'') \in Parent \wedge$ $(c, c'') \in Descendant \wedge (c''', c') \in Descendant$ $\Rightarrow (c''', c'') = (comp_c, comp_p) \vee c'' = comp_c \vee$ $c''' = comp_p \vee (comp_c, c'') \in Descendant \vee$ $(c''', comp_p) \in Descendant)\}\Big)$

A.2.5/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `start`

L'opération `start`$(comp)$, représenté en table A.5, permet de démarrer un composant $comp$ ainsi que ses descendants. La pré-condition permet de s'assurer que *a*) $comp$ est un élément de $Components$ ($comp \in Components$) et que *b*) toute interface requise de $comp$ ou d'un de ses éventuels descendants soit liée ($\forall c \in Components, i \in IRequired.\ (comp, c) \in Descendant \wedge ((i, comp) \in Requirer \vee (i, c) \in Requirer) \wedge (i, mandatory) \in Contingency \Rightarrow (\exists i' \in Interfaces.(i, i') \in hasBinding)$).

La post-condition consiste à retirer les couples de la relation fonctionnelle $State$ impliquant $comp$ et ses éventuels descendants (en excluant de $State$ sa propre restriction à l'ensemble contenant $comp$ et ses éventuels descendants) et à y ajouter les couples signifiant que $comp$ et ses éventuels descendants soient démarrés.

TABLE A.5: Opération primitive `start` : pré/post-conditions

Opération	`start`$(comp)$
Pré-condition	$comp \in Components \wedge \Big(\forall c \in Components, i \in IRequired.\ (comp, c) \in Descendant$ $\wedge\ ((i, comp) \in Requirer \vee (i, c) \in Requirer) \wedge (i, mandatory) \in Contingency$ $\Rightarrow (\exists i' \in Interfaces.(i, i') \in hasBinding)\Big)$
Post-condition	$State' = State \backslash State_{\mid \{comp\} \cup \{c \in Components \mid (comp, c) \in Descendant\}} \cup \{(comp, started)\} \cup$ $\{(c, started) \mid c \in Components \wedge (comp, c) \in Descendant\}$

A.2.6/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `stop`

L'opération `stop`($comp$), représenté en table A.6, permet de stopper un composant $comp$ ainsi que ses descendants. La pré-condition permet de s'assurer que $comp$ est un élément de $Components$ ($comp \in Components$) tandis que la post-condition consiste à retirer les couples de la relation fonctionnelle $State$ impliquant $comp$ et ses éventuels descendants (en excluant de $State$ sa propre restriction à l'ensemble contenant $comp$ et ses éventuels descendants) et à y ajouter les couples signifiant que $comp$ et ses éventuels descendants soient stoppés.

TABLE A.6: Opération primitive `stop` : pré/post-conditions

Opération	`stop`($comp$)
Pré-condition	$comp \in Components$
Post-condition	$State' = State \backslash State_{\|\{comp\} \cup \{c \in Components \| (comp,c) \in Descendant\}} \cup \{(comp, stopped)\} \cup \{(c, stopped) \mid c \in Components \wedge (comp, c) \in Descendant\}$

A.2.7/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `bind`

L'opération `bind`(int_1, int_2), représenté en table A.7, permet de lier deux interfaces int_1 et int_2 par la relation $Binding$ ou un mécanisme de délégation. La pré-condition permet de s'assurer que *a*) int_1 et int_2 soient des interfaces ($int_1, int_2 \in Interfaces$), *b*) que les interfaces int_1 et int_2 ne soient pas respectivement requise et fournie ($\neg(int_1 \in IRequired \wedge int_2 \in IProvided)$) et que *c*) pour les composants c_1 et c_2 contenant respectivement int_1 et int_2 ($\forall c_1, c_2 \in Components.(int_1, c_1) \in Container \wedge (int_2, c_2) \in Container$), *c.1*) si les interfaces int_1 et int_2 sont respectivement fournie et requise c_1 et c_2 sont des composants frères ayant un parent commun ($int_1 \in IProvided \wedge int_2 \in IRequired \Rightarrow \exists c \in Components.(c_1, c) \in Parent \wedge (c_2, c) \in Parent$) ou *c.2*) si les interfaces int_1 et int_2 sont toutes deux fournies ou requises, c_2 est un parent de c_1 et si l'une de ces interfaces est déjà impliquée dans une relation de délégation c'est forcément avec l'autre ($(((int_1 \in IProvided \wedge int_2 \in IProvided) \vee (int_1 \in IRequired \wedge int_2 \in IRequired)) \Rightarrow (c_1, c_2) \in Parent \wedge \forall i \in Interface.(int_1, i) \in Delegate \Rightarrow i = int_2 \wedge (i, int_2) \in Delegate \Rightarrow i = int_1)$).

La post-condition consiste à ajouter le couple (int_1, int_2) à la relation $Binding$, $DelegateProv$ ou $DelegateReq$ en fonction de la qualité requise ou fournie des interfaces concernées.

TABLE A.7: Opération primitive `bind` : pré/post-conditions

Opération	`bind`(int_1, int_2)
Pré-condition	$int_1, int_2 \in Interfaces \wedge (\exists t \in ITypes.(int_1, t) \in Types \wedge (int_2, t) \in Types)$ $\wedge \neg(int_1 \in IRequired \wedge int_2 \in IProvided)$ $\wedge \Big(\forall c_1, c_2 \in Components, \forall int \in Interfaces.(int_1, c_1) \in Container$ $\wedge\ (int_2, c_2) \in Container.\Big(int_1 \in IProvided$ $\wedge\ int_2 \in IRequired \Rightarrow (int_1, int) \notin DelegateProv \wedge (int_2, int) \notin DelegateReq$ $\wedge \exists c \in Components.(c_1, c) \in Parent \wedge (c_2, c) \in Parent\Big)$ $\wedge\ \Big(((int_1 \in IProvided \wedge int_2 \in IProvided)$ $\vee (int_1 \in IRequired \wedge int_2 \in IRequired)) \Rightarrow\ (c_1, c_2) \in Parent$ $\wedge (int_1, int) \notin Binding \wedge (int, int_1) \notin Binding$ $\wedge \forall i \in Interface.(int_1, i) \in Delegate \Rightarrow i = int_2$ $\wedge (i, int_2) \in Delegate \Rightarrow i = int_1\Big)\Big)$
Post-condition	$Binding' = \begin{cases} Binding \cup \{(int_1, int_2)\} & \text{si } int_1 \in IProvided \wedge int_2 \in IRequired \\ Binding & \text{sinon} \end{cases}$ $DelegateProv' = \begin{cases} DelegateProv \cup \{(int_1, int_2)\} & \text{si } int_1 \in IProvided \wedge int_2 \in IProvided \\ DelegateProv & \text{sinon} \end{cases}$ $DelegateReq' = \begin{cases} DelegateReq \cup \{(int_1, int_2)\} & \text{si } int_1 \in IRequired \wedge int_2 \in IRequired \\ DelegateReq & \text{sinon} \end{cases}$

A.2.8/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `unbind`

L'opération `unbind`(int_1, int_2), représenté en table A.8, permet de retirer le lien par la relation $Binding$ ou un mécanisme de délégation des interfaces int_1 et int_2. La pré-condition permet de s'assurer que *a*) int_1 et int_2 soient des interfaces ($int_1, int_2 \in Interfaces$), *b*) que si les interfaces int_1 et int_2 sont liées par une relation de délégation, le composant qui contient int_2 (le composant parent par opposition au composant enfant contenant int_1) ainsi que ses descendants sont stoppés ($(int_1, int_2) \in Delegate \Rightarrow (\forall c, c' \in Components.(int_2, c) \in Container \wedge (c, c') \in Descendant \Rightarrow (c, stopped) \in State \wedge (c', stopped) \in State)$), et que *c*) si les interfaces int_1 et int_2 sont liées par la relation $Binding$ le composant qui contient int_1 (l'interface requise) ainsi que ses descendants sont stoppés($(int_1, int_2) \in Binding \Rightarrow (\forall c, c' \in Components.(int_1, c) \in Container \wedge (c, c') \in Descendant \Rightarrow (c, stopped) \in State \wedge (c', stopped) \in State)$).

La post-condition consiste à retirer le couple (int_1, int_2) des relations $Binding$, $DelegateProv$ ou $DelegateReq$.

TABLE A.8: Opération primitive `unbind` : pré/post-conditions

Opération	`unbind`(int_1, int_2)
Pré-condition	$int_1, int_2 \in Interfaces$ $\wedge\ ((int_1, int_2) \in Delegate \Rightarrow (\forall c, c' \in Components.(int_2, c) \in Container$ $\wedge\ (c, c') \in Descendant \Rightarrow (c, stopped) \in State \wedge (c', stopped) \in State))$ $\wedge\ ((int_1, int_2) \in Binding \Rightarrow (\forall c, c' \in Components.(int_1, c) \in Container$ $\wedge\ (c, c') \in Descendant \Rightarrow (c, stopped) \in State \wedge (c', stopped) \in State))$
Post-condition	$Binding' = Binding \backslash \{(int_1, int_2)\}$ $DelegateProv' = DelegateProv \backslash \{(int_1, int_2)\}$ $DelegateReq' = DelegateReq \backslash \{(int_1, int_2)\}$

A.2.9/ SÉMANTIQUE DE L'OPÉRATION PRIMITIVE `update`

L'opération `update`$(param, val)$, représenté en table A.9, permet de mettre à jour la valeur d'un paramètre. La pré-condition permet de s'assurer que *a*) $param$ soit un paramètre ($param \in Parameters$), *b*) val soit une valeur valide ($val \in Values$) et que *c*) la valeur de val corresponde au type du paramètre $param$ ($val \in \{t \mid t \in PTypes \wedge (p, t) \in ParameterType\}$).

La post-condition consiste à retirer de la relation fonctionnelle $Value$ le couple impliquant $param$ (en excluant de $Value$ sa propre restriction à l'ensemble contenant $param$) et à y ajouter le couple $(param, val)$.

TABLE A.9: Opération primitive `update` : pré/post-conditions

Opération	`update`$(param, val)$
Pré-condition	$param \in Parameters \wedge val \in Values$ $\wedge\ val \in \{t \mid t \in PTypes \wedge (p, t) \in ParameterType\}$
Post-condition	$Value' = Value \backslash Value_{\mid \{param\}} \cup \{(param, val)\}$

B

PRÉSERVATION DE LA CONSISTANCE PAR LES OPÉRATIONS PRIMITIVES

Nous vérifions que pour $S = \langle C, C^0, \mathcal{R}^G_{run}, \mapsto, l \rangle$, un système à composants reconfigurable, chaque contrainte de consistance de la table 4.1 est préservée par les opérations primitives susceptibles de l'affecter, lorsque S évolue de d'une configuration c à une configuration c' au moyen d'une reconfiguration primitive.

La contrainte (CC.1) peut être affectée par l'opération `new`. De même, (CC.2), (CC.3) et (CC.4) peuvent être affectées par `add` ou `remove`, tandis que l'opération `start` ne peut affecter que la contrainte (CC.12). Enfin, l'opération `bind` peut avoir une influence sur les contraintes (CC.5) à (CC.11).

Sommaire

B.1/ OPÉRATION PRIMITIVE new

Cette opération primitive ne peut affecter que la contrainte (CC.1). Considérons la création d'un composant $comp$ dans S au moyen de la reconfiguration primitive $\texttt{new}(comp)$, abrégé par new ; par hypothèse on a $(l(c) \Rightarrow CC \wedge P_{\texttt{new}}) \wedge c \xrightarrow{\texttt{new}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{new}}$.

Comme (CC.1) et $P_{\texttt{new}}$ sont satisfaites par la configuration c, on a, en utilisant les notations de la table A.1, $\forall x \in Components.(\exists ip \in IProvided.\ Container(ip) = x)$, d'une part, et $IProvided_{comp} \neq \emptyset$ d'autre part.

Par définition de l'opération primitive new, $Components' = Components \cup \{comp\}$ et $IProvided' = IProvided \cup IProvided_{comp}$. Ainsi, $\forall x \in Components'.(\exists ip \in IProvided'.\ Container'(ip) = x)$, ce qui signifie que (CC.1) est satisfaite par c'.

B.2/ OPÉRATION PRIMITIVE add

Cette opération primitive ne peut affecter que les contraintes (CC.2), (CC.3) et (CC.4). Considérons l'ajout d'un composant $comp_c$ à un composant $comp_p$ dans S au moyen de la reconfiguration primitive $\texttt{add}(comp_p, comp_c)$, abrégé par add ; par hypothèse on a donc $(l(c) \Rightarrow CC \wedge P_{\texttt{add}}) \wedge c \xrightarrow{\texttt{add}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{add}}$.

De plus, par définition de l'opération primitive add (voir table A.3), $P_{\texttt{add}} = comp_p, comp_c \in Components \wedge comp_c \neq comp_p \wedge (comp_c, comp_p) \notin Descendant \wedge \forall p \in Parameters.(comp_p, p) \notin Definer$ et $R_{\texttt{add}}$ est tel que $Parent' = Parent \cup \{(comp_c, comp_p)\}$ et $Descendant' = Descendant \cup \{(comp_p, comp_c)\} \cup \{(q, comp_c) \mid \forall q \in Components, (q, comp_p) \in Descendant\} \cup \{(comp_p, q) \mid \forall q \in Components, (comp_c, q) \in Descendant\} \cup \{(q, q') \mid \forall q, q' \in Components, (q, comp_p) \in Descendant \wedge (comp_c, q') \in Descendant\}$.

B.2.1/ PRÉSERVATION DE (CC.2) PAR add.

Comme (CC.2) et $P_{\texttt{add}}$ sont satisfaites par la configuration c, on a $\forall x, y \in Components.(x, y) \in Parent \Rightarrow \forall p \in Parameters.\ Container(p) \neq y \wedge (y, x) \in Descendant$, d'une part, et $\forall p \in Parameters.(comp_p, p) \notin Definer$ (qui, comme $Container = Provider \uplus Requirer \uplus Definer$, peut s'écrire $\forall p \in Parameters.Container(p) \neq comp_p$) d'autre part.

Nous cherchons à établir que $\forall x, y \in Components'.(x, y) \in Parent' \Rightarrow \forall p \in Parameters'.\ Container'(p) \neq y \wedge (y, x) \in Descendant'$, ce qui signifie que (CC.2) est satisfaite par c'. Soient x et y des composants tel que $(x, y) \in Parent'$. Comme $Parent' = Parent \cup \{(comp_c, comp_p)\}$, si $(x, y) \in Parent$ alors, comme c satisfait (CC.2), cette contrainte est aussi satisfaite par c'.

Dans le cas où $(x, y) = (comp_c, comp_p)$, la condition $\forall p \in Parameters'.\ Container'(p) \neq y$ est satisfaite par le respect de la pré-condition $\forall p \in Parameters.(comp_p, p) \notin Definer$. La condition $(y, x) \in Descendant'$ est une conséquence de la post-condition qui spécifie $Descendant'$ comme l'union de $Descendant$ et (entre autres) de l'ensemble $\{(comp_p, comp_c)\}$.

B.2.2/ PRÉSERVATION DE (CC.3) PAR add.

Voyons à présent comment l'opération primitive add préserve la contrainte (CC.3). Comme (CC.3) et $P_{\texttt{add}}$ sont satisfaites par la configuration c, on a $\forall x, y, z \in Components.\{(x,y),(y,z)\} \subseteq Descendant \Rightarrow (x,z) \in Descendant$, d'une part, et $comp_p, comp_c \in Components \wedge (comp_c, comp_p) \notin Descendant$ d'autre part.

Nous cherchons à établir que (CC.3) est satisfaite par c', c'est à dire que : $\forall x, y, z \in Components'.\{(x,y),(y,z)\} \subseteq Descendant' \Rightarrow (x,z) \in Descendant'$. Dans le cas où les couples (x,y) et (y,z) sont tous deux différents de $(comp_c, comp_p)$ (CC.3) est satisfaite par c' comme elle l'était par c avant l'occurrence de l'opération add. Il nous reste à établir que (CC.3) est satisfaite par c' lorsque $(x,y) = (comp_p, comp_c)$ et quand $(y,z) = (comp_p, comp_c)$ pour vérifier que l'application de l'opération add préserve (CC.3).

Si nous identifions x avec $comp_p$ et y avec $comp_c$, nous devons établir que lorsque z est un descendant de $comp_c$, i.e., $(comp_c, z) \in Descendant'$, on a $(comp_p, z) \in Descendant'$. Or, d'après la post-condition de l'opération add, $\{(comp_p, q) \mid \forall q \in Components, (comp_c, q) \in Descendant\} \subseteq Descendant'$, ce qui nous permet d'établir que $(comp_p, z) \in Descendant'$.

Si à présent nous identifions y avec $comp_p$ et z avec $comp_c$, nous devons établir que lorsque x a pour descendant $comp_p$, i.e., $(x, comp_p) \in Descendant'$, on a $(x, comp_c) \in Descendant'$. Or, d'après la post-condition de l'opération add, $\{(q, comp_c) \mid \forall q \in Components, (q, comp_p) \in Descendant\} \subseteq Descendant'$, ce qui nous permet d'établir que $(x, comp_c) \in Descendant'$. Ceci nous permet de conclure que l'application de l'opération add préserve (CC.3).

B.2.3/ PRÉSERVATION DE (CC.4) PAR add.

Considérons à présent la contrainte (CC.4). Comme (CC.4) et $P_{\texttt{add}}$ sont satisfaites par la configuration c, on a $\forall x, y \in Components.(x,y) \in Descendant \Rightarrow x \neq y \wedge ((y,x) \in Parent \vee \exists z_1, z_2, z_3 \in Components.\{(z_1,y),(x,z_2),(x,z_3),(z_3,y)\} \subseteq Descendant \wedge \{(z_1,x),(y,z_2)\} \subseteq Parent \wedge (y,x) \notin Parent)$ d'une part et $(comp_c, comp_p) \notin Descendant$ d'autre part.

Nous souhaitons établir que (CC.4) est satisfaite par c', c'est à dire que : $\forall x, y \in Components'.(x,y) \in Descendant' \Rightarrow x \neq y \wedge ((y,x) \in Parent' \vee \exists z_1, z_2, z_3 \in Components.\{(z_1,y),(x,z_2),(x,z_3),(z_3,y)\} \subseteq Descendant' \wedge \{(z_1,x),(y,z_2)\} \subseteq Parent' \wedge (y,x) \notin Parent')$. Comme $Descendant'$ est défini comme étant l'union de $Descendant$ et des ensembles $\{(comp_p, comp_c)\}$, $\{(q, comp_c) \mid \forall q \in Components, (q, comp_p) \in Descendant\}$, $\{(comp_p, q) \mid \forall q \in Components, (comp_c, q) \in Descendant\}$ et $\{(q,q') \mid \forall q, q' \in Components, (q, comp_p) \in Descendant \wedge (comp_c, q') \in Descendant\}$, il nous suffit d'établir que la condition $x \neq y \wedge ((y,x) \in Parent' \vee \exists z_1, z_2, z_3 \in Components.\{(z_1,y),(x,z_2),(x,z_3),(z_3,y)\} \subseteq Descendant' \wedge \{(z_1,x),(y,z_2)\} \subseteq Parent' \wedge (y,x) \notin Parent')$ est satisfaite pour tout couple (x,y) appartenant à l'un de ces ensembles.

De façon pratique, plutôt que de démontrer globalement que $\{(z_1,y),(x,z_2),(x,z_3),(z_3,y)\} \subseteq Descendant' \wedge \{(z_1,x),(y,z_2)\} \subseteq Parent' \wedge (y,x) \notin Parent'$, nous établirons séparément les trois assertions suivantes : *a*) $(z_1,x) \in Parent' \wedge (z_1,y) \in Descendant'$, *b*) $(x,z_2) \in Descendant' \wedge (y,z_2) \in Parent'$ et *c*) $(x,z_3) \in Descendant' \wedge (z_3,y) \in Descendant'$.

— Si $(x,y) \in Descendant$, on est dans le cas de la configuration c qui satisfait (CC.4).

— Si $(x,y) \in \{(comp_p, comp_c)\}$, on identifie x à $comp_p$ et y à $comp_c$. Ainsi, la pré-condition

$comp_c \neq comp_p$ permet de s'assurer que $x \neq y$; la post-condition $Parent' = Parent \cup \{(comp_c, comp_p)\}$ établi que $(y, x) \in Parent'$. La contrainte (CC.4) est donc satisfaite.

— Si $(x, y) \in \{(q, comp_c) \mid \forall q \in Components, (q, comp_p) \in Descendant\}$, on identifie y à $comp_c$ et on vérifie que la contrainte (CC.4) est satisfaite pour $(x, comp_c)$ lorsque x admet $comp_p$ pour descendant, i.e., $(x, comp_p) \in Descendant$. Si on avait $x = comp_c$, on aurait $(comp_c, comp_p) \in Descendant$, ce qui est en contradiction avec la pré-condition $(comp_c, comp_p) \notin Descendant$, donc $x \neq comp_c$, i.e., $x \neq y$.

En outre, comme c satisfait (CC.4), s'il existe au moins un composant x tel que $(x, comp_p) \in Descendant$ on aurait soit $(comp_p, x) \in Parent$, soit il existerait des composant z_1, z_2 et z_3 tels que $\{(z_1, x), (comp_p, z_2)\} \subseteq Parent \wedge \{(z_1, comp_p)(x, z_2), (x, z_3), (z_3, comp_p)\} \subseteq Descendant$. Examinons ces deux cas.

Dans le premier cas ($(comp_p, x) \in Parent$), il existe donc bien, pour $(x, comp_c) \in Descendant \subseteq Descendant'$, un composant $comp_p$ tel que $(comp_p, x) \in Parent' \wedge (comp_p, comp_c) \in Descendant'$, ce qui vérifie (CC.4).

Dans le second cas ($\{(z_1, comp_p)(x, z_2), (x, z_3), (z_3, comp_p)\} \subseteq Descendant$ et $\{(z_1, x), (comp_p, z_2)\} \subseteq Parent$), on utilise (CC.2) pour déduire d'une part que $(comp_p, comp_c) \in Descendant'$ en utilisant $(comp_c, comp_p) \in Parent'$ et d'autre part que $(z_2, comp_p) \in Descendant'$ en utilisant $(comp_p, z_2) \in Parent \subseteq Parent'$. On applique ensuite (CC.3) pour déduire *a*) de $(z_1, comp_p) \in Descendant \subseteq Descendant'$ et de $(comp_p, comp_c) \in Descendant'$ que $(z_1, comp_c) \in Descendant'$; *b*) de $(x, z_2) \in Descendant \subseteq Descendant'$ et de $(z_2, comp_p) \in Descendant'$ que $(x, comp_p) \in Descendant'$; *c*) de $(z_3, comp_p) \in Descendant \subseteq Descendant'$ et de $(comp_p, comp_c) \in Descendant'$ que $(z_3, comp_c) \in Descendant'$. Il existe donc bien, pour $(x, comp_c) \in Descendant'$, *a*) un composant z_1 tel que $(z_1, x) \in Parent' \wedge (z_1, comp_c) \in Descendant'$, *b*) un composant $z_2' = comp_p$ tel que $(comp_c, z_2') \in Parent' \wedge (x, z_2') \in Descendant'$; *c*) un composant z_3 tel que $(x, z_3) \in Descendant \subseteq Descendant'$ et $(z_3, comp_c) \in Descendant'$; ce qui vérifie (CC.4).

— Si $(x, y) \in \{(comp_p, q) \mid \forall q \in Components, (comp_c, q) \in Descendant\}$, on identifie x à $comp_p$ et on vérifie que la contrainte (CC.4) est satisfaite pour $(comp_p, y)$ lorsque y est un descendant de $comp_c$. Si $y = comp_p$ on aurait $(comp_c, comp_p) \in Descendant$, ce qui est en contradiction avec la pré-condition $(comp_c, comp_p) \notin Descendant$, donc $y \neq comp_p$, i.e., $x \neq y$. Ainsi, il existe bien un composant, $z_1 = comp_c$, tel que $(z_1, comp_p) \in Parent'$ et $(z_1, y) \in Descendant \subseteq Descendant'$. De plus, $z_3 = comp_c$ nous permet aussi de satisfaire $\{(comp_p, z_3), (z_3, y)\} \subseteq Descendant'$. Il nous reste à établir l'existence d'un composant z_2' tel que $(comp_p, z_2') \in Descendant' \wedge (y, z_2') \in Parent \subseteq Parent'$ ce qui permettrait à c' de satisfaire (CC.4).

Par hypothèse, on a $(comp_c, y) \in Descendant$. Comme c satisfait (CC.4), soit $(y, comp_c) \in Parent$, soit il existe un composant z_2 tel que $(comp_c, z_2) \in Descendant$ et $(y, z_2) \in Parent$. Dans le premier cas ($(y, comp_c) \in Parent$), on a bien $z_2' = comp_c$ qui vérifie $(comp_p, z_2') \in Descendant' \wedge (y, z_2') \in Parent'$. Dans le second cas ($(comp_c, z_2) \in Descendant$ et $(y, z_2) \in Parent$), on utilise (CC.3) pour déduire de $(comp_p, comp_c) \in Descendant'$ et $(comp_c, z_2) \in Descendant \subseteq Descendant'$ que $(comp_p, z_2) \in Descendant'$ et on a bien $z_2' = z_2$ qui vérifie $(comp_p, z_2') \in Descendant' \wedge (y, z_2') \in Parent'$.

— Si $(x, y) \in \{(q, q') \mid \forall q, q' \in Components, (q, comp_p) \in Descendant \wedge (comp_c, q') \in Descendant\}$, on a $(x, comp_p) \in Descendant$ et $(comp_c, y) \in Descendant$. Si $x = y$, cela signifie qu'il existe q tel que $(comp_c, q) \in Descendant$ et $(q, comp_p) \in Descendant$. En appliquant (CC.4), on a $(comp_c, comp_p) \in Descendant$ ce qui est en contradiction avec la post-condition $(comp_c, comp_p) \notin Descendant$.

Comme (CC.3) est satisfaite par c' on peut déduire de la post-condition $(comp_p, comp_c) \in Descendant'$ et de $(comp_c, y) \in Descendant \subseteq Descendant'$ que $(comp_p, y) \in Descendant'$. Ainsi, puisque $(x, comp_p) \in Descendant \subseteq Descendant'$, il existe bien un composant, $z'_3 = comp_p$ tel que $(x, z'_3) \in Descendant \wedge (z'_3, y) \in Descendant$. Il nous reste à vérifier que $(y, x) \in Parent'$ ou qu'il existe des composant z'_1 et z'_2 tels que $(z'_1, x) \in Parent' \wedge (z'_1, y) \in Descendant'$ et $(x, z'_2) \in Descendant' \wedge (y, z'_2) \in Parent'$.

Comme $(x, comp_p) \in Descendant$, en application de (CC.4) qui est satisfaite par c, on en déduit que $(comp_p, x) \in Parent$ ou qu'il existe des composant z_1, z_2 et z_3 tels que $(z_1, x) \in Parent \wedge (z_1, comp_p) \in Descendant$, $(x, z_2) \in Descendant \wedge (comp_p, z_2) \in Parent$ et $(x, z_3) \in Descendant \wedge (z_3, comp_p) \in Descendant$. De plus, on utilise le fait qu'en application de la pré-condition de `add` on a $(comp_p, comp_c) \in Descendant'$ et, comme c' satisfait (CC.3), on déduit de $(x, comp_p) \in Descendant'$ et $(comp_p, comp_c) \in Descendant'$ que $(x, comp_c) \in Descendant'$.

Dans le premier cas ($(comp_p, x) \in Parent$), comme (CC.3) est satisfaite par c' on peut déduire de la post-condition $(comp_p, comp_c) \in Descendant'$ et de $(comp_c, y) \in Descendant \subseteq Descendant'$ que $(comp_p, y) \in Descendant'$. De plus, comme $Parent \subseteq Parent'$, on a $(comp_p, x) \in Parent'$. Ainsi, il existe bien un composant, $z'_1 = comp_p$ tel que $(z'_1, x) \in Parent' \wedge (z'_1, y) \in Descendant'$. Comme c satisfait (CC.4), on peut déduire de $(comp_c, y) \in Descendant$ que soit $(y, comp_c) \in Parent$, soit il existe un composant z_2 tel que $(y, z_2) \in Parent \wedge (comp_c, z_2) \in Descendant$.

— Dans ce premier sous-cas ($(y, comp_c) \in Parent$), on a bien $z'_2 = comp_c$ qui vérifie $(x, z'_2) \in Descendant' \wedge (y, z'_2) \in Parent'$.

— Dans le second sous-cas ($(y, z_2) \in Parent \wedge (comc_p, z_2) \in Descendant$), on utilise (CC.3) pour déduire de $(x, comp_p) \in Descendant'$ et $(comp_p, z_2) \in Descendant \subseteq Descendant'$ que $(x, z_2) \in Descendant'$ et on a bien $z'_2 = z_2$ qui vérifie $(x, z'_2) \in Descendant' \wedge (y, z'_2) \in Parent'$, ce qui établit (CC.4) pour le premier cas.

Dans le second cas ($(z_1, x) \in Parent \wedge (z_1, comp_p) \in Descendant$, $(x, z_2) \in Descendant \wedge (comp_p, z_2) \in Parent$ et $(x, z_3) \in Descendant \wedge (z_3, comp_p) \in Descendant$), comme (CC.3) est satisfaite par c' on peut déduire de $(z_1, comp_p) \in Descendant \subseteq Descendant'$ et de la post-condition $(comp_p, comp_c) \in Descendant \subseteq Descendant'$ que $(z_1, comp_c) \in Descendant'$. En utilisant encore (CC.3) avec $(z_1, comp_c) \in Descendant'$ et $(comp_c, y) \in Descendant \subseteq Descendant'$, on établit $(z_1, y) \in Descendant'$. Comme $(z_1, x) \in Parent \subseteq Parent'$, il existe bien un composant, z_1 tel que $(z_1, x) \in Parent' \wedge (z_1, y) \in Descendant'$. Comme c satisfait (CC.4), on peut déduire de $(comp_c, y) \in Descendant$ que soit $(y, comp_c) \in Parent$, soit il existe un composant z_2 tel que $(y, z_2) \in Parent \wedge (comp_c, z_2) \in Descendant$.

— Dans ce premier sous-cas ($(y, comp_c) \in Parent$), on a bien $z'_2 = comp_c$ qui vérifie $(x, z'_2) \in Descendant' \wedge (y, z'_2) \in Parent'$.

— Dans le second sous-cas ($(y, z_2) \in Parent \wedge (comc_p, z_2) \in Descendant$), on utilise (CC.3) pour déduire de $(x, comp_p) \in Descendant'$ et $(comp_p, z_2) \in Descendant \subseteq Descendant'$ que $(x, z_2) \in Descendant'$ et on a bien $z'_2 = z_2$ qui vérifie $(x, z'_2) \in Descendant' \wedge (y, z'_2) \in Parent'$, ce qui établit (CC.4) pour le second cas.

B.3/ OPÉRATION PRIMITIVE `remove`

Cette opération primitive ne peut affecter que les contraintes (CC.2), (CC.3) et (CC.4). Considérons le retrait du composant $comp_c$ d'un composant $comp_p$ de S au moyen de la reconfiguration primitive `remove`$(comp_p, comp_c)$, abrégé par `remove` ; par hypothèse on a donc $(l(c) \Rightarrow CC \wedge P_{\texttt{remove}}) \wedge c \xrightarrow{\texttt{remove}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{add}}$.

De plus, par définition de l'opération primitive `remove` (voir Table A.4), on a $P_{\texttt{remove}} = comp_p, comp_c \in Components \wedge \ (comp_p, stopped) \in State \wedge (\forall c \in Components.(comp_p, c) \in Descendant \Rightarrow (c, stopped) \in State) \wedge \Big(\forall i, i' \in Interfaces, c \in Components.(i, c) \in Container \wedge (i', comp_c) \in Container \wedge \ (i, i') \in hasBinding \Rightarrow c \notin \{comp_p, comp_c\} \wedge (c, comp_p) \notin Parent\Big)$ et $R_{\texttt{remove}}$ est tel que $Parent' = Parent \backslash \{(comp_c, comp_p)\}$ et $Descendant' = Descendant \backslash (r_0 \cup r_c \cup r_p \cup r_2)$ où

- $r_0 = \{(comp_p, comp_c) \mid \forall q \in Components.(comp_p, q) \notin Descendant \vee (q, comp_c) \notin Descendant\}$,
- $r_c = \{(q, comp_c) \mid \forall q \in Components.(q, comp_p) \in Descendant \wedge (comp_c, q) \notin Parent \wedge (\forall q' \in Components.(comp_c, q') \in Parent \wedge (q, q') \in Descendant \Rightarrow q' = comp_p)\}$,
- $r_p = \{(comp_p, q) \mid \forall q \in Components.(comp_c, q) \in Descendant \wedge (q, comp_p) \notin Parent \wedge (\forall q' \in Components.(q', comp_p) \in Parent \wedge (q', q) \in Descendant \Rightarrow q' = comp_c)\}$,
- $r_2 = \{(q, q') \mid \forall q, q' \in Components.(q, comp_p) \in Descendant \wedge (comp_c, q') \in Descendant \wedge (q', q) \notin Parent \wedge (\forall q'' \in Components.(q', q'') \notin Parent \vee (q'', q) \notin Parent) \wedge (\forall q'', q''' \in Components.(q''', q'') \in Parent \wedge (q, q'') \in Descendant \wedge (q''', q') \in Descendant \Rightarrow (q''', q'') = (comp_c, comp_p) \vee q'' = comp_c \vee q''' = comp_p \vee (comp_c, q'') \in Descendant \vee (q''', comp_p) \in Descendant)\}$.

B.3.1/ PRÉSERVATION DE (CC.2) PAR `remove`.

Comme (CC.2) est satisfaite par la configuration c, on a $\forall x, y \in Components.((x, y) \in Parent \Rightarrow \forall p \in Parameters.\ Container(p) \neq y \wedge (y, x) \in Descendant)$.

Nous cherchons à établir que (CC.2) est satisfaite par c', c'est à dire que : $\forall x, y \in Components'.((x, y) \in Parent' \Rightarrow \forall p \in Parameters'.\ Container'(p) \neq y \wedge (y, x) \in Descendant')$. Pour ce faire, considérons le couple de composants $(x, y) \in Parent'$; de par l'application de l'opération primitive `remove`, (x, y) est différent de $(comp_c, comp_p)$. On voit que, comme (CC.2) est satisfaite par c, le composant composite y n'a pas de paramètres, formellement, on a $\forall p \in Parameters'.\ Container'(p) \neq y$. Il nous reste donc à établir que $(y, x) \in Descendant'$.

Comme, du fait que (CC.2) soit satisfaite par c, $(y, x) \in Descendant$, d'après la définition de $Descendant'$, il nous suffit de montrer que $(y, x) \notin r_0 \cup r_c \cup r_p \cup r_2$ ce que nous faisons en considérant chaque ensemble de cette union et en établissant que (y, x) n'appartient à aucun d'entre eux.

- r_0 **:** Il s'agit de l'ensemble contenant le couple $(comp_p, comp_c)$ s'il n'existe aucun composant à la fois descendant de $comp_p$ et admettant $comp_c$ comme descendant, sinon cet ensemble est vide. Comme, par hypothèse, (x, y) est différent de $(comp_c, comp_p)$, il est clair que $(y, x) \notin \{(comp_p, comp_c)\}$.
- r_c **:** Il s'agit de l'ensemble des couples $(q, comp_c)$ où q a $comp_p$ comme descendant mais n'est pas parent de $comp_c$ et où $comp_p$ est l'unique composant étant à la fois

descendant de q et parent de $comp_c$. En identifiant x à $comp_c$, on a par hypothèse $(comp_c, y) \in Parent$; ce qui est en contradiction avec une des conditions énoncées ci-dessus, à avoir que $q.(q, conmp_c) \in r_c$ n'est pas parent de $comp_c$. Ainsi, $(y, x) \notin r_c$.

— r_p **:** Il s'agit de l'ensemble des couples $(comp_p, q)$ où q est un descendant de $comp_c$, n'a pas $comp_p$ pour parent et où $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel q soit un descendant. En identifiant y à $comp_p$, on a par hypothèse $(x, comp_p) \in Parent$; ce qui est en contradiction avec une des conditions énoncées ci-dessus, à avoir que $q.(comp_p, q) \in r_p$ n'admet pas $comp_p$ comme parent. Ainsi, $(y, x) \notin r_p$.

— r_2 **:** Il s'agit de l'ensemble des couples (q, q') où q a $comp_p$ comme descendant et q' est un descendant de $comp_c$, q n'est pas un parent de q', il n'existe pas de composant admettant q pour parent qui soit aussi parent de q' et pour tout couple de composants (q'', q''') tel que q'' soit un parent de q''', q et q''' admettent respectivement pour descendant q'' et q' seulement si *a*) $(q'', q''') = (comp_p, comp_c)$, *b*) $q'' = comp_c$, *c*) $q''' = comp_p$, *d*) $(comp_c, q'') \in Descendant$ ou *e*) $(q''', comp_p) \in Descendant$. Or, par hypothèse, $(x, y) \in Parent$; ce qui est en contradiction avec une des conditions énoncées ci-dessus, à avoir que $(q', q) \notin Parent$ pour $(q, q') \in r_2$. Ainsi, $(y, x) \notin r_2$.

B.3.2/ PRÉSERVATION DE (CC.3) PAR `remove`.

Comme (CC.3) est satisfaite par la configuration c, on a $\forall x, y, z \in Components.\{(x, y), (y, z)\} \subseteq Descendant \Rightarrow (x, z) \in Descendant$.

Nous cherchons à établir que (CC.3) est satisfaite par c', c'est-à-dire $\forall x, y, z \in Components'.\{(x, y), (y, z)\} \subseteq Descendant' \Rightarrow (x, z) \in Descendant'$. Comme $Descendant' = Descendant \backslash (r_0 \cup r_c \cup r_p \cup r_2)$, si $\{(x, y), (y, z)\} \subseteq Descendant' \subseteq Descendant$, alors $\{(x, y), (y, z)\}$ et $r_0 \cup r_c \cup r_p \cup r_2$ sont disjoints. Comme (CC.3) est satisfaite par la configuration c, $(x, z) \in Descendant$, il suffit d'établir que $(x, z) \notin r_0 \cup r_c \cup r_p \cup r_2$ pour en déduire $(x, z) \in Descendant'$ ce qui permet de satisfaire (CC.3) par c'.

Afin de prouver que lorsque $\{(x, y), (y, z)\} \subseteq Descendant$, $(x, z) \notin r_0 \cup r_c \cup r_p \cup r_2$, nous considérons les quatre cas suivants.

— Si $(x, z) \in r_0$, on a une contradiction car $r_0 = \emptyset$ du fait qu'il existe par hypothèse un composant y tel que $\{(x, y), (y, z)\} \subseteq Descendant$.

— Si $(x, z) \in r_c$, en identifiant z avec $comp_c$, cela signifie, par définition de r_c, que $(x, comp_p) \in Descendant \wedge (comp_c, x) \notin Parent \wedge (\forall q \in Components.(comp_c, q) \in Parent \wedge (x, q) \in Descendant \Rightarrow q = comp_p)$.

  Ainsi, x admet $comp_p$ comme descendant mais n'est pas parent de $comp_c$ et $comp_p$ est l'unique composant étant à la fois descendant de x et parent de $comp_c$.

  Comme par hypothèse $\{(x, y), (y, z)\} \subseteq Descendant' \subseteq Descendant$, on a $(y, comp_c) \in Descendant' \subseteq Descendant$ et c satisfait (CC.4), on en déduit que soit $(comp_c, y) \in Parent$, soit il existe un composant q tel que $(y, q) \in Descendant \wedge (comp_c, q) \in Parent$.

  Dans le premier cas ($(comp_c, y) \in Parent$), comme $(x, y) \in Descendant$ et $comp_p$ est l'unique composant étant à la fois descendant de x et parent de $comp_c$, $y = comp_p$ et donc $(comp_p, comp_c) \in Descendant'$. Dans ce cas, on aurait $r_0 \neq \emptyset$ et il existerait un composant q tel que $(comp_c, q) \in Descendant \wedge (q, comp_p) \in Descendant$. Or, aucune

des hypothèses que nous avons utilisées ne spécifie l'existence d'un tel composant q. Ainsi, l'établissement de l'existence d'un tel composant dans le cas général constitue en soit une contradiction.

Dans le second cas ($(y,q) \in Descendant \wedge (comp_c,q) \in Parent$), puisque c satisfait (CC.3), on déduit de $(x,y) \in Descendant$ et $(y,q) \in Descendant$ que $(x,q) \in Descendant$. Comme $comp_p$ est l'unique composant étant à la fois descendant de x et parent de $comp_c$, $q = comp_p$. Donc, $(y,comp_p) \in Descendant \wedge (comp_c,comp_p) \in Parent$; de plus $(comp_c,y) \notin Parent$ et (comme y est un descendant de x admettant $comp_p$ pour descendant) $comp_p$ est l'unique composant étant à la fois descendant de y et parent de $comp_c$; cela signifie que $(y,comp_c) \in r_c$, ce qui est en contradiction avec $(y,comp_c) \in Descendant'$.

— Si $(x,z) \in r_p$, on identifie x avec $comp_p$, cela signifie, par définition de r_p, que $(comp_c,z) \in Descendant \wedge (z,comp_p) \notin Parent \wedge (\forall q \in Components.(q,comp_p) \in Parent \wedge (q,z) \in Descendant \Rightarrow q = comp_c)$.

Ainsi z est un descendant de $comp_c$, n'a pas $comp_p$ pour parent et $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel z soit un descendant.

Comme par hypothèse $\{(x,y),(y,z)\} \subseteq Descendant' \subseteq Descendant$, on a $(comp_p,y) \in Descendant' \subseteq Descendant$ et c satisfait (CC.4), on en déduit que soit $(y,comp_p) \in Parent$, soit il existe un composant q tel que $(q,comp_p) \in Parent \wedge (q,y) \in Descendant$.

Dans le premier cas ($(y,comp_p) \in Parent$), comme $(y,z) \in Descendant$ et $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel z soit un descendant, $y = comp_c$ et donc $(comp_p,comp_c) \in Descendant'$. On peut appliquer le même raisonnement que ci-dessus, lors du traitement de l'éventualité $(x,z) \in r_c$ (cas où $(comp_c,y) \in Parent$) pour en déduire une contradiction.

Dans le second cas ($(q,comp_p) \in Parent \wedge (q,y) \in Descendant$), puisque c satisfait (CC.3), on déduit de $(q,y) \in Descendant$ et $(y,z) \in Descendant$ que $(q,z) \in Descendant$. Comme $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel z soit un descendant, $q = comp_c$. Donc $(comp_c,comp_p) \in Parent \wedge (comp_c,y) \in Descendant$; de plus, $(y,comp_p) \notin Parent$ et (comme z est un descendant de y qui lui-même est un descendant de $comp_p$) $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel y soit un descendant ; cela signifie que $(comp_p,y) \in r_p$, ce qui est en contradiction avec $(comp_p,y) \in Descendant'$.

— Si $(x,z) \in r_2$, cela signifie par définition de r_2 que $(x,comp_p) \in Descendant \wedge (comp_c,z) \in Descendant \wedge (z,x) \notin Parent \wedge (\forall q \in Components.(z,q) \notin Parent \vee (q,x) \notin Parent) \wedge (\forall q,q' \in Components.(q',q) \in Parent \wedge (x,q) \in Descendant \wedge (q',z) \in Descendant \Rightarrow (q',q) = (comp_c,comp_p) \vee q = comp_c \vee q' = comp_p \vee (comp_c,q) \in Descendant \vee (q',comp_p) \in Descendant)$.

Ainsi, x a $comp_p$ comme descendant et z est un descendant de $comp_c$, x n'est pas un parent de z, il n'existe pas de composant admettant x pour parent qui soit aussi parent de z et pour tout couple de composants (q,q') tel que q soit un parent de q', x et q' admettent respectivement pour descendant q et z seulement si *a*) $(q,q') = (comp_p,comp_c)$, *b*) $q = comp_c$, *c*) $q' = comp_p$, *d*) $(comp_c,q) \in Descendant$ ou *e*) $(q',comp_p) \in Descendant$. Intuitivement, le fait que (q,q') ne remplisse aucune de ces conditions signifie qu'il existe une suite $c_0,\ldots,c_n$ où $(q_i,q_{i-1}) \in Parent$, pour $1 \leq i \leq n$, telle que, si $q = c_n$ et $q' = c_0$, $(q_i,q_{i-1}) \neq (comp_c,comp_p)$ pour $1 \leq i \leq n$, c'est-à-dire que la suite $q_0,\ldots,q_n$ basée sur les relations de parenté entre ses termes ne dépend pas de la relation de parenté entre $comp_c$ et $comp_p$.

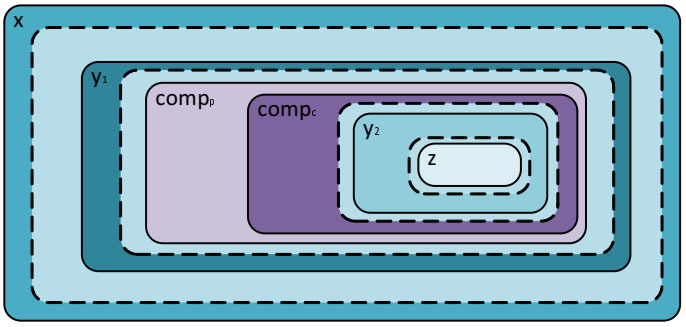


FIGURE B.1 – Organisation possible des composants lorsque $(y_1, comp_p) \in Descendant$ et $(y_1, comp_p) \in Descendant$ dans le cas $(x, z) \in r_2$ où $\{(x, y_1), (y_1, z), (x, y_2), (y_2, z)\} \in Descendant$

Les différent agencements possibles des composants x, y, z, $comp_p$ et $comp_c$ sont représentées figure B.1 où chaque forme en traits pleins représente un composant dont le nom se trouve dans le coin supérieur gauche, tandis que les formes en traits discontinus représentent une imbrication d'un nombre quelconque (pouvant être nul) de composants. Comme par hypothèse $\{(x, y), (y, z)\} \subseteq Descendant' \subseteq Descendant$, quatre cas de figure peuvent se présenter : *a*) $(y, comp_p) \in Descendant$, comme c'est le cas du composant y_1 de la figure B.1, *b*) $y = comp_p$, *c*) $y = comp_c$ ou *d*) $(comp_c, y) \in Descendant$, comme c'est le cas du composant y_2 de la figure B.1.

Dans le premier cas ($(y, comp_p) \in Descendant$), comme on a $(comp_c, z) \in Descendant$, on en déduit que $(y, z) \in r_2$. En effet, dans le cas général, on peut choisir y et z tels que $(z, y) \notin Parent$, qu'il n'y ait pas de composant q qui soit à la fois parent de z dont y soit un parent ($\forall q \in Components.(z, q) \notin Parent \vee (q, y) \notin Parent$) et que la condition $\forall q, q' \in Components.(q', q) \in Parent \wedge (y, q) \in Descendant \wedge (q', z) \in Descendant \Rightarrow (q', q) = (comp_c, comp_p) \vee q = comp_c \vee q' = comp_p \vee (comp_c, q) \in Descendant \vee (q', comp_p) \in Descendant$ soit remplie.

Dans le second cas ($y = comp_p$), comme on a $(comp_c, z) \in Descendant$, on en déduit que $(y, z) = (comp_p, z) \in r_p$. En effet, dans le cas général, on peut choisir z tel que $(z, comp_p) \notin Parent$ et où $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel z soit un descendant ($\forall q \in Components.(q, comp_p) \in Parent \wedge (q, z) \in Descendant \Rightarrow q = comp_c$).

Dans le troisième cas ($y = comp_c$), comme on a $(x, comp_p) \in Descendant$, on en déduit que $(x, y) = (x, comp_c) \in r_c$. En effet, dans le cas général, on peut choisir x tel que $(comp_c, x) \notin Parent$ et où $comp_p$ est l'unique composant étant à la fois descendant de x et parent de $comp_c$ ($\forall q \in Components.(comp_c, q) \in Parent \wedge (x, q) \in Descendant \Rightarrow q = comp_p$).

Dans le dernier cas ($(comp_c, y) \in Descendant$), comme on a $(x, comp_p) \in Descendant$, on en déduit que $(x, y) \in r_2$. En effet, dans le cas général, on peut choisir x et y tels que $(y, x) \notin Parent$, qu'il n'y ait pas de composant q qui soit à la fois parent de y dont x soit un parent ($\forall q \in Components.(y, q) \notin Parent \vee (q, x) \notin Parent$) et que la condition $\forall q, q' \in Components.(q', q) \in Parent \wedge (x, q) \in Descendant \wedge (q', y) \in Descendant \Rightarrow (q', q) = (comp_c, comp_p) \vee q = comp_c \vee q' = comp_p \vee (comp_c, q) \in Descendant \vee (q', comp_p) \in Descendant$ soit remplie.

Ainsi, dans tous les cas on a $\{(x, y), (y, z)\} \cap (r_c \cup r_p \cup r_2) \neq \emptyset$, ce qui est en contradiction avec notre hypothèse $\{(x, y), (y, z)\} \subseteq Descendant'$.

B.3.3/ PRÉSERVATION DE (CC.4) PAR `remove`.

Considérons à présent la contrainte (CC.4). Comme (CC.4) est satisfaite par la configuration c, on a $\forall x, y \in Components.(x, y) \in Descendant \Rightarrow x \neq y \wedge ((y, x) \in Parent \vee \exists z_1, z_2, z_3 \in Components.\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent \wedge (y, x) \notin Parent)$.

Nous souhaitons établir que (CC.4) est satisfaite par c', c'est à dire que : $\forall x, y \in Components'.(x, y) \in Descendant' \Rightarrow x \neq y \wedge ((y, x) \in Parent' \vee \exists z_1, z_2, z_3 \in Components.\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant' \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent' \wedge (y, x) \notin Parent')$. Puisque $Descendant' = Descendant \backslash (r_0 \cup r_c \cup r_p \cup r_2)$, si $(x, y) \in Descendant' \subseteq Descendant$, alors $(x, y) \notin r_0 \cup r_c \cup r_p \cup r_2$. Comme (CC.4) est satisfaite par la configuration c, considérons le couple $(x, y) \in Descendant' \subseteq Descendant$, qui par définition vérifie $x \neq y$. Il nous reste à établir que $x \neq y \wedge ((y, x) \in Parent' \vee \exists z_1, z_2, z_3 \in Components.\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant' \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent' \wedge (y, x) \notin Parent')$.

De façon pratique, plutôt que de démontrer globalement que $\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant' \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent' \wedge (y, x) \notin Parent'$, nous établirons séparément les trois assertions suivantes : *a*) $(z_1, x) \in Parent' \wedge (z_1, y) \in Descendant'$, *b*) $(x, z_2) \in Descendant' \wedge (y, z_2) \in Parent'$ et *c*) $(x, z_3) \in Descendant' \wedge (z_3, y) \in Descendant'$.

Comme (CC.4) est satisfaite par c, soit $(y, x) \in Parent$, soit il existe z_1, z_2 et z_3 tels que $\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent$.

Dans le premier cas ($(y, x) \in Parent$), d'après la post-condition $Parent' = Parent \backslash \{(comp_c, comp_p)\}$, pour tout couple $(x, y) \neq (comp_p, comp_c)$, la contrainte (CC.4) est satisfaite par c'. Si $(x, y) = (comp_p, comp_c)$, examinons $r_0 = \{(comp_p, comp_c) \mid \forall q \in Components.(comp_p, q) \notin Descendant \vee (q, comp_c) \notin Descendant\}$.

Si $r_0 = \{(comp_p, comp_c)\}$, alors $(comp_p, comp_c) \notin Descendant'$ ce qui permet à c' de satisfaire (CC.4). Si, au contraire, ($r_0 = \emptyset$) il existe un composant q tel que $(comp_p, q) \in Descendant \wedge (q, comp_c) \in Descendant$. Par construction $\{(comp_p, q), (q, comp_c)\} \cap (r_0 \cup r_c \cup r_p \cup r_2) = \emptyset$. En effet, le couple $(comp_p, q)$ ne peut être élément de r_0, r_c ou r_2 (voir page 228 pour la définition de ces ensembles) et si $(comp_p, q) \in r_c$, on aurait $(comp_c, q) \in Descendant$ ce qui combiné à $(q, comp_c) \in Descendant$ en utilisant (CC.3) qui est satisfaite par c donnerait $(comp_c, comp_c) \in Descendant$ qui viole la contrainte (CC.4) satisfaite par c. Symétriquement, le couple $(q, comp_c)$ ne peut être élément de r_0, r_p ou r_2 et si $(q, comp_c) \in r_p$, on aurait $(q, comp_p) \in Descendant$ ce qui combiné à $(comp_p, q) \in Descendant$ en utilisant (CC.3) qui est satisfaite par c donnerait $(comp_p, comp_p) \in Descendant$ qui viole aussi la contrainte (CC.4) satisfaite par c. Ainsi $(comp_p, q) \in Descendant' \wedge (q, comp_c) \in Descendant'$.

Du fait que $Descendant' \subseteq Descendant$ et comme (CC.4) est satisfaite par c, deux cas de figure sont possibles pour $(comp_p, q)$: soit $(q, comp_p) \in Parent$, soit $(q, comp_p) \notin Parent$. Il en est de même pour $(q, comp_c)$ et soit $(comp_c, q) \in Parent$, soit $(comp_c, q) \notin Parent$. Examinons les quatre éventualités possibles.

- Si $(q, comp_p) \in Parent$ et $(comp_c, q) \in Parent$, on a bien $z_1 = z_2 = z_3 = q$ tels que $(z_1, comp_p) \in Parent' \wedge (z_1, comp_c) \in Descendant'$, $(comp_p, z_2) \in Descendant' \wedge (comp_c, z_2) \in Parent'$ et $(comp_p, z_3) \in Descendant' \wedge (z_3, comp_c) \in Descendant'$, car $Parent' = Parent \backslash \{(comp_c, comp_p)\}$ et $q \notin \{comp_p, comp_c\}$, ce qui satisfait (CC.4).
- Si $(q, comp_p) \in Parent$ et $(comp_c, q) \notin Parent$, comme (CC.4) est satisfaite par c, il existe z_1, z_2 et z_3 tels que $(z_1, q) \in Parent \wedge (z_1, comp_c) \in Descendant$, $(q, z_2) \in$

$Descendant \wedge (comp_c, z_2) \in Parent$ et $(q, z_3) \in Descendant \wedge (z_3, comp_c) \in Descendant$. En choisissant $z'_1 = q$, on a $(z'_1, comp_p) \in Parent' \wedge (z'_1, comp_c) \in Descendant'$; si $z'_2 = z'_3 = z_2$, on a $(comp_p, z'_2) \in Descendant' \wedge (comp_c, z'_2) \in Parent'$ (en utilisant (CC.3) pour déduire de $(comp_p, q) \in Descendant$ et $(q, z_2) \in Descendant$ que $(comp_p, z_2) \in Descendant$) et $(comp_p, z'_3) \in Descendant' \wedge (z'_3, comp_c) \in Descendant'$.

— Si $(q, comp_p) \notin Parent$ et $(comp_c, q) \in Parent$, comme (CC.4) est satisfaite par c, il existe z_1, z_2 et z_3 tels que $(z_1, comp_p) \in Parent \wedge (z_1, q) \in Descendant$, $(comp_p, z_2) \in Descendant \wedge (q, z_2) \in Parent$ et $(comp_p, z_3) \in Descendant \wedge (z_3, q) \in Descendant$. En choisissant $z'_1 = z_1$, on a $(z'_1, comp_p) \in Parent' \wedge (z'_1, comp_c) \in Descendant'$ (en utilisant (CC.3) pour déduire de $(z_1, q) \in Descendant$ et $(q, comp_c) \in Descendant$ que $(z_1, comp_c) \in Descendant$) ; si $z'_2 = z'_3 = q$, on a $(comp_p, z'_2) \in Descendant' \wedge (comp_c, z'_2) \in Parent'$ et $(comp_p, z'_3) \in Descendant' \wedge (z'_3, comp_c) \in Descendant'$.

— Si $(q, comp_p) \notin Parent$ et $(comp_c, q) \notin Parent$, comme (CC.4) est satisfaite par c, il existe z_1^p, z_2^p et z_3^p tels que $(z_1^p, comp_p) \in Parent \wedge (z_1^p, q) \in Descendant$, $(comp_p, z_2^p) \in Descendant \wedge (q, z_2^p) \in Parent$ et $(comp_p, z_3^p) \in Descendant \wedge (z_3^p, q) \in Descendant$ d'une part et z_1^c, z_2^c et z_3^c tels que $(z_1^c, q) \in Parent \wedge (z_1^c, comp_c) \in Descendant$, $(q, z_2^c) \in Descendant \wedge (comp_c, z_2^c) \in Parent$ et $(q, z_3^c) \in Descendant \wedge (z_3^c, comp_c) \in Descendant$ d'autre part. En choisissant $z'_1 = z_1^p$, on a $(z'_1, comp_p) \in Parent' \wedge (z'_1, comp_c) \in Descendant'$ (en utilisant (CC.3) pour déduire de $(z_1^p, q) \in Descendant$ et $(q, comp_c) \in Descendant$ que $(z_1^p, comp_c) \in Descendant$) ; si $z'_2 = z'_3 = z_2^c$, on a $(comp_p, z'_2) \in Descendant' \wedge (comp_c, z'_2) \in Parent'$ (en utilisant (CC.3) pour déduire de $(comp_p, q) \in Descendant$ et $(q, z_2^c) \in Descendant$ que $(comp_p, z_2^c) \in Descendant$) et $(comp_p, z'_3) \in Descendant' \wedge (z'_3, comp_c) \in Descendant'$.

Dans le cas où $\{(z_1, y), (x, z_2), (x, z_3), (z_3, y)\} \subseteq Descendant \wedge \{(z_1, x), (y, z_2)\} \subseteq Parent$, on a par hypothèse $(y, x) \notin Parent$, $Parent' = Parent \backslash \{(comp_c, comp_p)\}$, $(x, y) \neq (comp_p, comp_c)$ et $x \neq y$, donc $\{(z_1, x), (y, z_2)\} \subseteq Parent'$. Afin d'établir que les couples (z_1, y) et (x, z_2) appartiennent à $Descendant'$, c'est-à-dire que ce ne sont pas des éléments de $r_0 \cup r_c \cup r_p \cup r_2$, nous considérons les huit cas suivants.

— $(z_1, y) \notin r_0$ Dans le cas contraire ($(z_1, y) \in r_0$), si $r_0 \neq \emptyset$, on aurait $z_1 = comp_p$ et $y = comp_c$. Or, le fait que $r_0 \neq \emptyset$ implique que $(comp_p, comp_c) \notin Descendant'$. Néanmoins, dans ce cas notre hypothèse $(x, y) \in Descendant'$ s'écrit $(x, comp_c) \in Descendant'$. Si $(x, comp_c) \in r_c$, on aurait $(x, comp_p) \in Descendant$ (ce qu'on peut déduire de $(z_1, x) \in Parent$ en utilisant (CC.3)), $(comp_c, x) \notin Parent$ (du fait que $(y, x) \notin Parent$) et $comp_p$ serait l'unique composant étant à la fois descendant de x et parent de $comp_c$. S'il y avait un autre composant $q \neq comp_p$ descendant de x et parent de $comp_c$, $(x, comp_c) \notin r_c$, donc $(x, comp_c) \in Descendant'$ et alors la relation de descendance entre x et $comp_c$ ne dépendrait pas de la relation de parenté entre $comp_c$ et $comp_p$ et la contrainte (CC.4) serait satisfaite par c' du fait qu'elle l'est par c. Dans le cas contraire, $(x, comp_c) \in r_c$ donc $(x, comp_c) \notin Descendant'$, ce qui satisfait (CC.4).

— $(x, z_2) \notin r_0$ Dans le cas contraire ($(x, z_2) \in r_0$), si $r_0 \neq \emptyset$, on aurait $x = comp_p$ et $z_2 = comp_c$. Dans ce cas notre hypothèse $(x, y) \in Descendant'$ s'écrit $(comp_p, y) \in Descendant'$. Si $(comp_p, y) \in r_p$, on aurait $(comp_c, y) \in Descendant$ (ce qu'on peut déduire de $(y, z_2) \in Parent$ en utilisant (CC.3)), $(y, comp_p) \notin Parent$ (du fait que $(y, x) \notin Parent$) et $comp_c$ serait l'unique composant ayant $comp_p$ pour parent duquel y soit un descendant. S'il y avait un autre composant $q \neq comp_c$ duquel y soit un descendant et ayant $comp_p$ pour parent, $(comp_p, y) \notin r_c$, donc $(comp_p, y) \in Descendant'$ et alors la relation de descendance entre $comp_p$, y ne dépendrait pas de la relation de parenté

entre $comp_c$ et $comp_p$ et la contrainte (CC.4) serait satisfaite par c' du fait qu'elle l'est par c. Dans le cas contraire, $(comp_p, y) \in r_c$ donc $(comp_p, y) \notin Descendant'$, ce qui satisfait (CC.4).

— $(z_1, y) \notin r_c$ Considérons r_c, l'ensemble des couples $(q, comp_c)$ où q a $comp_p$ comme descendant mais n'est pas parent de $comp_c$ et où $comp_p$ est l'unique composant étant à la fois descendant de q et parent de $comp_c$. Si $(z_1, y) \in r_c$, on aurait $(z_1, comp_p) \in Descendant$ et $comp_p$ serait l'unique composant étant à la fois descendant de z_1 et parent de $comp_c$. En identifiant y à $comp_c$, on a par hypothèse $(x, comp_c) \in Descendant'$. S'il existe un composant $z'_1 \neq z_1$ ayant x pour parent tel que $(z'_1, comp_p) \in Descendant$ et $comp_p$ ne soit pas l'unique composant étant à la fois descendant de z'_1 et parent de $comp_c$, alors on aurait $(z'_1, comp_p) \in Descendant'$ car $(z'_1, comp_c) \notin r_c$. En revanche, s'il n'existe pas de tel z'_1, le composant x, en tant que parent de z_1 (ou d'un autre composant similaire q tel que $(q, comp_c) \in r_c$), est tel qu'on aurait $(x, comp_c) \in r_c$ et donc $(x, comp_c) \notin Descendant'$.

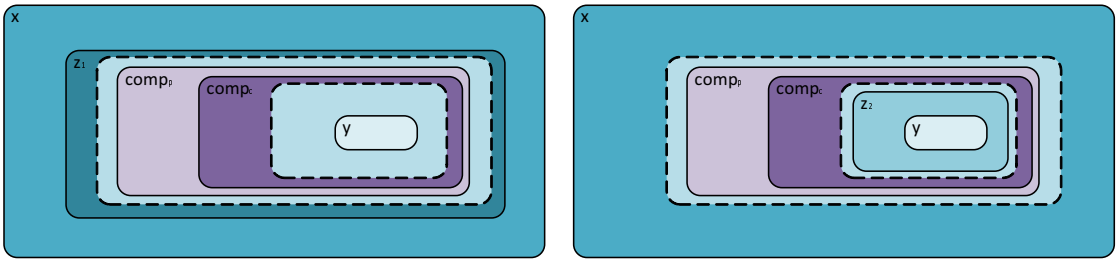
— $(x, z_2) \notin r_c$ Comme ci-dessus, si $(x, z_2) \in r_c$; on aurait $(x, comp_p) \in Descendant$ et $comp_p$ serait l'unique composant étant à la fois descendant de x et parent de $comp_c$. Si $(x, z_2) \in r_c$, on aurait $(x, comp_p) \in Descendant$ et $comp_p$ serait l'unique composant étant à la fois descendant de z_1 et parent de $comp_c$. En identifiant z_2 à $comp_c$, on a $(y, comp_c) \in Parent$. Ainsi, comme x a pour descendant $comp_p$ et y est un descendant de $comp_c$, $(x, y) \in r_2$ alors la relation de descendance entre x et y dépend de la relation de parenté entre $comp_c$ et $comp_p$. Si ce n'était pas le cas il existerait un composant $z'_2 \neq comp_c$ tel que $(y, z'_2) \in Parent' \wedge (x, z'_2) \in Descendant'$. Dans le cas contraire $(x, y) \in r_2$ et donc $(x, y) \notin Descendant'$.

— $(z_1, y) \notin r_p$ Comme r_p est l'ensemble des couples $(comp_p, q)$ où q est un descendant de $comp_c$, n'a pas $comp_p$ pour parent et où $comp_c$ est l'unique composant ayant $comp_p$ pour parent duquel q soit un descendant, si $(z_1, y) \in r_p$ on aurait $(comp_c, y) \in Descendant \wedge (y, comp_p) \notin Parent$ et $comp_c$ serait l'unique composant ayant $comp_p$ pour parent duquel y soit un descendant. En identifiant z_1 à $comp_p$, on a $(comp_p, x) \in Parent$. Ainsi, comme x a pour descendant $comp_p$ et y est un descendant de $comp_c$, $(x, y) \in r_2$ alors la relation de descendance entre x et y dépend de la relation de parenté entre $comp_c$ et $comp_p$. Si ce n'était pas le cas il existerait un composant $z'_1 \neq comp_p$ tel que $(z'_1, x) \in Parent' \wedge (z'_1, y) \in Descendant'$. Dans le cas contraire $(x, y) \in r_2$ et donc $(x, y) \notin Descendant'$.

— $(x, z_2) \notin r_p$ Comme ci-dessus, si $(x, z_2) \in r_p$ on aurait $(comp_c, z_2) \in Descendant \wedge (z_2, comp_p) \notin Parent$ et $comp_c$ serait l'unique composant ayant $comp_p$ pour parent duquel z_2 soit un descendant. En identifiant x à $comp_p$, on a par hypothèse $(comp_p, y) \in Descendant'$. S'il existe un composant $z'_2 \neq z_2$ étant un parent de y tel que $(comp_c, z'_2) \in Descendant \wedge (z'_2, comp_p) \notin Parent$ et $comp_c$ ne soit pas l'unique composant ayant $comp_p$ pour parent duquel z'_2 soit un descendant, alors on aurait $(comp_p, z'_2) \in Descendant'$ car $(comp_p, z'_2) \notin r_p$. En revanche, s'il n'existe pas de tel z'_2, le composant y, ayant z'_2 (ou un autre composant similaire q tel que $(comp_p, q) \in r_p$) pour parent, est tel qu'on aurait $(x, comp_c) \in r_c$ et donc $(x, comp_c) \notin Descendant'$.

— $(z_1, y) \notin r_2$ Comme r_2 est l'ensemble des couples (q, q') tels que q admet $comp_p$ pour descendant et q' est un descendant de $comp_c$ pour lesquels la relation de descendance entre q et q' dépend de la relation de parenté entre $comp_c$ et $comp_p$, si $(z_1, y) \in r_2$ l'organisation des composants est représenté figure B.2(a) avec les mêmes conventions que la figure B.1. S'il existe un composant $z'_1 \neq z_1$ tel que $(z'_1, x) \in Parent \wedge (z'_1, y) \in Descendant$ pour lequel la relation de descendance entre z'_1

et y ne dépend pas de la relation de parenté entre $comp_c$ et $comp_p$, alors $(z'_1, y) \notin r_2$. En revanche, si un tel composant z'_1 n'existe pas, la relation de descendance entre x et y dépend de la relation de parenté entre $comp_c$ et $comp_p$ et donc $(x, y) \in r_2$ d'où $(x, comp_c) \notin Descendant'$.



(a) Cas où $(z_1, y) \in r_2$ (b) Cas où $(x, z_2) \in r_2$

FIGURE B.2 – Représentation de l'organisation des composants dans les cas $(z_1, y) \in r_2$ et $(x, z_2) \in r_2$ de l'application de l'opération primitive `remove`

— $(x, z_2) \notin r_2$ Comme ci-dessus, si $(x, z_2) \in r_2$ l'organisation des composants est représenté figure B.2(b). S'il existe un composant $z'_2 \neq z_2$ tel que $(y, z'_2) \in Parent \wedge (x, z'_2) \in Descendant$ pour lequel la relation de descendance entre x et z'_2 ne dépend pas de la relation de parenté entre $comp_c$ et $comp_p$, alors $(x, z'_2) \notin r_2$. En revanche, si un tel composant z'_2 n'existe pas, la relation de descendance entre x et y dépend de la relation de parenté entre $comp_c$ et $comp_p$ et donc $(x, y) \in r_2$ d'où $(x, comp_c) \notin Descendant'$.

Nous avons établi que $\{(z_1, x), (y, z_2)\} \subseteq Parent'$ et que les couples (z_1, y) et (x, z_2) appartiennent à $Descendant'$. Considérons à présent $z'_3 = z_1$, comme c' satisfait (CC.3) on peut déduire de $(z'_3, x) \in Parent'$ que $(x, z'_3) \in Descendant'$. Ainsi on a bien $\{(x, z'_3), (z'_3, y)\} \subseteq Descendant'$ ce qui finit d'établir que c' satisfait (CC.4).

B.4/ OPÉRATION PRIMITIVE `bind`

Cette opération primitive ne peut affecter que les contraintes (CC.5), (CC.6), (CC.7), (CC.8), (CC.9), (CC.10) et (CC.11). Considérons deux interfaces distinctes int_1 et int_2 appartenant respectivement aux composant $comp_1$ et $comp_2$ (non nécessairement distincts) de S et la liaison de ces interfaces au moyen de la reconfiguration primitive $\texttt{bind}(int_1, int_2)$, abrégé par `bind`. Par hypothèse on a donc $(l(c) \Rightarrow CC \wedge P_{\texttt{bind}}) \wedge c \xrightarrow{\texttt{bind}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{bind}}$.

De plus, par définition de l'opération primitive `bind` (voir table A.7), $P_{\texttt{bind}} = int_1, int_2 \in Interfaces \wedge (\exists t \in ITypes.(int_1, t) \in Types \wedge (int_2, t) \in Types) \wedge \neg(int_1 \in IRequired \wedge int_2 \in IProvided) \wedge \Big(\forall q_1, q_2 \in Components, \forall int \in Interfaces.(int_1, q_1) \in Container \wedge (int_2, q_2) \in Container.\Big(int_1 \in IProvided \wedge int_2 \in IRequired \Rightarrow (int_1, int) \notin DelegateProv \wedge (int_2, int) \notin DelegateReq \wedge \exists q \in Components.(q_1, q) \in Parent \wedge (q_2, q) \in Parent\Big) \wedge \Big(((int_1 \in IProvided \wedge int_2 \in IProvided) \vee (int_1 \in IRequired \wedge int_2 \in IRequired)) \Rightarrow (q_1, q_2) \in Parent \wedge (int_1, int) \notin Binding \wedge (int, int_1) \notin Binding \wedge \forall i \in Interface.(int_1, i) \in Delegate \Rightarrow i = int_2 \wedge (i, int_2) \in Delegate \Rightarrow i =$

$int_1)\Big)$ et $R_{\texttt{bind}}$ est tel que $Binding' = Binding \cup \{(int_1, int_2)\}$ si int_1 et int_2 sont respectivement des interfaces fournies et requises, $DelegateProv' = DelegateProv \cup \{(int_1, int_2)\}$ si int_1 et int_2 sont des interfaces fournies ou $DelegateReq' = DelegateReq \cup \{(int_1, int_2)\}$ si int_1 et int_2 sont des interfaces requises.

B.4.1/ PRÉSERVATION DE (CC.5) PAR bind.

Comme (CC.5) est satisfaite par la configuration c, on a $\forall ip \in IProvided, \forall ir \in IRequired.(ip, ir) \in Binding \Rightarrow \Big(ContainerType(ip) = ContainerType(ir) \wedge (\exists c \in Components.\{(Container(ip), c), (Container(ir), c)\} \subseteq Parent)\Big)$.

Nous souhaitons établir que (CC.5) est satisfaite par c', c'est-à-dire $\forall ip \in IProvided, \forall ir \in IRequired.(ip, ir) \in Binding' \Rightarrow \Big(ContainerType(ip) = ContainerType(ir) \wedge (\exists c \in Components.\{(Container(ip), c), (Container(ir), c)\} \subseteq Parent)\Big)$.

Si $(ip, ir) \in Binding \subseteq Binding'$, alors c' satisfait (CC.5) du fait que c satisfait aussi cette contrainte. Dans le cas où $(ip, ir) \in Binding'$ mais $(ip, ir) \notin Binding$, comme $Binding' = Binding \cup \{(int_1, int_2)\}$ si int_1 et int_2 sont respectivement des interfaces fournies et requises, on a $(ip, ir) = (int_1, int_2)$.

Les parties de la pré-condition $\exists t \in ITypes.(int_1, t) \in Types \wedge (int_2, t) \in Types$ et $\forall q_1, q_2 \in Components.(int_1, q_1) \in Container \wedge (int_2, q_2) \in Container.\Big(int_1 \in IProvided \wedge int_2 \in IRequired \Rightarrow \exists q \in Components.(q_1, q) \in Parent \wedge (q_2, q) \in Parent\Big)$ vérifient respectivement les conditions $ContainerType(int_1) = ContainerType(int_2)$ et $\exists c \in Components.\{(Container(int_1), c), (Container(int_2), c)\} \subseteq Parent$ qui utilisent une notation fonctionnelle. Ceci établit que c' satisfait (CC.5).

B.4.2/ PRÉSERVATION DE (CC.6) PAR bind.

Comme (CC.6) est satisfaite par la configuration c, on a $\forall ip \in IProvided, \forall ir \in IRequired, \forall id \in Interfaces.((ip, ir) \in Binding \Rightarrow Delegate(ip) \neq id \wedge Delegate(ir) \neq id)$.

Nous souhaitons établir que (CC.6) est satisfaite par c', c'est-à-dire $\forall ip \in IProvided, \forall ir \in IRequired, \forall id \in Interfaces.((ip, ir) \in Binding' \Rightarrow Delegate'(ip) \neq id \wedge Delegate'(ir) \neq id)$.

Si $(ip, ir) \in Binding \subseteq Binding'$, alors c' satisfait (CC.6) du fait que c satisfait aussi cette contrainte. Dans le cas où $(ip, ir) \in Binding'$ mais $(ip, ir) \notin Binding$, comme $Binding' = Binding \cup \{(int_1, int_2)\}$ si int_1 et int_2 sont respectivement des interfaces fournies et requises, on a $(ip, ir) = (int_1, int_2)$. La partie de la pré-condition $int_1 \in IProvided \wedge int_2 \in IRequired \Rightarrow (int_1, int) \notin DelegateProv \wedge (int_2, int) \notin DelegateReq$ permet, comme $Delegate = DelegateProv \uplus DelegateReq$, de conclure que c' satisfait (CC.6).

B.4.3/ PRÉSERVATION DE (CC.7) PAR bind.

Comme (CC.7) est satisfaite par la configuration c, on a $\forall i, i' \in Interfaces.Delegate(i) = i' \Rightarrow \forall ip \in IProvided.(ip, i) \notin Binding \wedge \forall ir \in IRequired.(i, ir) \notin Binding$.

Nous souhaitons établir que (CC.7) est satisfaite par c', c'est-à-dire $\forall i, i' \in Interfaces.Delegate'(i) = i' \Rightarrow \forall ip \in IProvided.(ip, i) \notin Binding' \wedge \forall ir \in IRequired.(i, ir)'$.

Comme $c \xrightarrow{\texttt{bind}} c'$, le système évolue de c à c' via l'opération primitive $\texttt{bind}(int_1, int_2)$; d'après la pré-condition $P_{\texttt{bind}}$, soit int_1 et int_2 sont respectivement des interfaces fournies et requises, soit elles sont toutes deux fournies ou toutes deux requises. Considérons ces trois cas.

— Dans le cas $int_1 \in IProvided \wedge int_2 \in IRequired$, on a $Delegate' = Delegate$ (puisque $DelegateProv' = DelegateProv$ et $DelegateReq' = DelegateReq$) et $Binding' = Binding \cup \{(int_1, int_2)\}$. Considérons deux interfaces i et i', comme $Delegate' = Delegate$, on a $Delegate'(i) = i' \Rightarrow Delegate(i) = i \Rightarrow \forall ip \in IProvided.(ip, i) \notin Binding \wedge \forall ir \in IRequired.(i, ir) \notin Binding$. Il nous reste à vérifier que $Delegate(i) = i' \Rightarrow (int_1, i) \notin Binding \wedge (i, int_2) \notin Binding$, ce qui est garanti par la partie $int_1 \in IProvided \wedge int_2 \in IRequired \Rightarrow (int_1, int) \notin DelegateProv \wedge (int_2, int) \notin DelegateReq$ de la pré-condition $P_{\texttt{bind}}$.

— Dans le cas $\{int_1, int_2\} \subseteq IRequired$, on a $Delegate' = Delegate \cup \{(int_1, int_2)\}$ (puisque $DelegateProv' = DelegateProv$ et $DelegateReq' = DelegateReq \cup \{(int_1, int_2)\}$) et $Binding' = Binding$. Comme int_1 est une interface requise, on doit s'assurer que $\forall ip \in IProvided.(ip, int_1) \notin Binding$, ce qui est garanti par la partie $(int, int_1) \notin Binding$ de la pré-condition $P_{\texttt{bind}}$.

— Dans le cas $\{int_1, int_2\} \subseteq IProvided$, on a $Delegate' = Delegate \cup \{(int_1, int_2)\}$ (puisque $DelegateProv' = DelegateProv \cup \{(int_1, int_2)\}$ et $DelegateReq' = DelegateReq$) et $Binding' = Binding$. Comme int_1 est une interface fournie, on doit s'assurer que $\forall ir \in IRequired.(int_1, ir) \notin Binding$, ce qui est garanti par la partie $(int_1, int) \notin Binding$ de la pré-condition $P_{\texttt{bind}}$, ce qui nous permet de conclure que c' satisfait (CC.7).

B.4.4/ PRÉSERVATION DE (CC.8) ET (CC.9) PAR bind.

Comme (CC.8) et (CC.9) sont satisfaites par la configuration c, on a $\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in IProvided \Rightarrow i' \in IProvided)$ et $\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in IRequired \Rightarrow i' \in IRequired)$.

Nous souhaitons établir que (CC.8) et (CC.9) sont satisfaites par c', c'est-à-dire $\forall i, i' \in Interfaces.(Delegate'(i) = i' \wedge i \in IProvided \Rightarrow i' \in IProvided)$ et $\forall i, i' \in Interfaces.(Delegate'(i) = i' \wedge i \in IRequired \Rightarrow i' \in IRequired)$. Comme $c \xrightarrow{\texttt{bind}} c'$, le système évolue de c à c' via l'opération primitive $\texttt{bind}(int_1, int_2)$; d'après la pré-condition $P_{\texttt{bind}}$, soit int_1 et int_2 sont respectivement des interfaces fournies et requises et dans ce cas la relation $Delegate$ est inchangée, soit elles sont toutes deux fournies ou toutes deux requises, ce qui permet, dans chacun de ces cas, à c' de satisfaire (CC.8) et (CC.9).

B.4.5/ PRÉSERVATION DE (CC.10) PAR bind.

Comme (CC.10) est satisfaite par la configuration c, on a $\forall i, i' \in Interfaces.(Delegate(i) = i' \Rightarrow ContainerType(i) = ContainerType(i') \wedge (Container(i), Container(i')) \in Parent)$.

Nous souhaitons établir que (CC.10) est satisfaite par c', c'est-à-dire $\forall i, i' \in Interfaces.(Delegate'(i) = i' \Rightarrow ContainerType(i) = ContainerType(i') \wedge (Container(i), Container(i')) \in Parent)$. Comme $c \xrightarrow{\texttt{bind}} c'$, le système évolue de c à c' via l'opération primitive $\texttt{bind}(int_1, int_2)$; d'après la pré-condition $P_{\texttt{bind}}$, soit int_1 et int_2 sont respectivement des interfaces fournies et requises et dans ce cas la relation

$Delegate$ est inchangée, soit elles sont toutes deux fournies ou toutes deux requises. Dans chacun de ces deux derniers cas, on a $\exists t \in ITypes.(int_1, t) \in Types \wedge (int_2, t) \in Types$ et $(q_1, q_2) \in Parent$ où $(int_1, q_1) \in Container \wedge\ (int_2, q_2) \in Container$, ce qui permet à c' de satisfaire (CC.10).

B.4.6/ PRÉSERVATION DE (CC.11) PAR `bind`.

Comme (CC.11) est satisfaite par la configuration c, on a $\forall i, i', i'' \in Interfaces.((Delegate(i) = i' \wedge Delegate(i) = i'' \Rightarrow i' = i'') \wedge (Delegate(i) = i'' \wedge Delegate(i') = i'' \Rightarrow i = i'))$.

Nous souhaitons établir que (CC.11) est satisfaite par c', c'est-à-dire $\forall i, i', i'' \in Interfaces.((Delegate'(i) = i' \wedge Delegate'(i) = i'' \Rightarrow i' = i'') \wedge (Delegate'(i) = i'' \wedge Delegate'(i') = i'' \Rightarrow i = i'))$. Comme $c \xrightarrow{\texttt{bind}} c'$, le système évolue de c à c' via l'opération primitive $\texttt{bind}(int_1, int_2)$; d'après la pré-condition $P_{\texttt{bind}}$, soit int_1 et int_2 sont respectivement des interfaces fournies et requises et dans ce cas la relation $Delegate$ est inchangée, soit elles sont toutes deux fournies ou toutes deux requises. Dans chacun de ces deux derniers cas, on a $\forall i \in Interface.(int_1, i) \in Delegate \Rightarrow i = int_2 \wedge (i, int_2) \in Delegate \Rightarrow i = int_1$, ce qui permet à c' de satisfaire (CC.11).

B.5/ OPÉRATION PRIMITIVE `start`

Cette opération primitive ne peut affecter que la contrainte (CC.12) Considérons le démarrage d'un composant $comp$ dans S au moyen de la reconfiguration primitive $\texttt{start}(comp)$, abrégé par `start` ; par hypothèse on a $(l(c) \Rightarrow CC \wedge P_{\texttt{start}}) \wedge c \xrightarrow{\texttt{start}} c'$ et on cherche à établir que $l(c') \Rightarrow R_{\texttt{start}}$.

Comme (CC.12) et $P_{\texttt{start}}$ sont satisfaites par la configuration c, on a, en utilisant les notations de la table A.5, $\forall ir \in IRequired.\Big(State(Container(ir)) = started \wedge Contingency(ir) = mandatory \Rightarrow \exists i \in Interfaces.((i, ir) \in Binding \vee Delegate(i) = ir \vee Delegate(ir) = i)\Big)$ d'une part et $\forall comp \in Components \wedge \Big(\forall q \in Components, i \in IRequired.\ (comp, q) \in Descendant \wedge\ ((i, comp) \in Requirer \vee (i, q) \in Requirer) \wedge (i, mandatory) \in Contingency \Rightarrow (\exists i' \in Interfaces.(i, i') \in hasBinding)\Big)$ d'autre part. De plus, $R_{\texttt{start}}$ est tel que $State' = State \backslash State_{|\{comp\} \cup \{c \in Components | (comp,c) \in Descendant\}} \cup \{(comp, started)\} \cup \{(c, started) \mid c \in Components \wedge (comp, c) \in Descendant\}$.

Nous souhaitons établir que (CC.12) est est satisfaite par c', c'est-à-dire $\forall ir \in IRequired.\Big(State(Container(ir)) = started \wedge Contingency(ir) = mandatory \Rightarrow \exists i \in Interfaces.((i, ir) \in Binding' \vee Delegate'(i) = ir \vee Delegate'(ir) = i)\Big)$. Pour ce faire, considérons un composant x ayant une interface requise i_x tel que $(i_x, mandatory) \in Contingency \wedge (x, started) \in State'$. Si $(x, started) \in State$, alors c' satisfait (CC.12) car c'est aussi, par hypothèse, le cas de c. Si $(x, started) \in State' \backslash State$, alors $(x, started) \in \{(comp, started)\} \cup \{(c, started) \mid c \in Components \wedge (comp, c) \in Descendant\}$, comme on a $\forall q \in Components, i \in IRequired.\ (comp, q) \in Descendant \wedge\ ((i, comp) \in Requirer \vee (i, q) \in Requirer) \wedge (i, mandatory) \in Contingency \Rightarrow (\exists i' \in Interfaces.(i, i') \in hasBinding)$, c' satisfait (CC.12) du fait que $hasBinding = Binding \uplus Binding^{-1} \uplus Delegate \uplus Delegate^{-1}$.

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/

Résumé :

Notre objectif principal est de permettre l'utilisation de propriétés temporelles dans une politique d'adaptation en tenant compte des spécificités de la vérification à l'exécution.
Pour y répondre, nous définissons un modèle de système à composants supportant les reconfigurations dynamiques. Nous introduisons les reconfigurations gardées qui nous permettent d'utiliser des opérations primitives en tant que "briques" pour construire des reconfigurations plus élaborées impliquant des constructions, non seulement, séquentielles, mais aussi alternatives ou répétitives, tout en garantissant la consistance des configurations du système.
En outre, nous étendons (aux événements externes) la logique temporelle utilisée précédemment pour exprimer des contraintes architecturales sur des configurations. Avec une sémantique, dite progressive, nous pouvons, dans la plupart des cas, évaluer (de façon centralisée ou décentralisée) une expression temporelle pour une configuration donnée à partir d'évaluations déjà réalisées à la seule configuration précédente. Nous utilisons cette logique dans des politiques d'adaptation permettant de guider et contrôler les reconfigurations de systèmes à composants à l'exécution.
Enfin, l'implémentation de politiques d'adaptation a été expérimentée dans divers cas d'études sur les plateformes Fractal et FraSCAti. Nous utilisons notamment le fuzzing comportemental afin de pouvoir tester des aspects spécifiques d'une politique d'adaptation.

Mots-clés : Composants, Adaptation, Reconfiguration, Reconfiguration dynamique, Propriétés temporelles

Abstract:

Our main goal is to allow the usage of temporal properties within an adaptation policy while taking into account runtime verification specificities.
In order to reach it, we define a component-based system model that supports dynamic reconfigurations. We introduce guarded reconfigurations in order to use primitive operations as "building blocks" to craft more elaborated reconfigurations involving, not only sequential, but also, alternate and repetitive constructs while ensuring the system's configurations consistency.
Furthermore, we extend (to external events) the temporal logic previously used to express architectural constraints on configurations. Using, so called, progressive semantics, we can, in most of the cases, evaluate (in a centralised or decentralised fashion) a temporal expression for a given configuration using evaluations performed only at the previous configuration. We use this logic within adaptation policies enabling the steering and control of dynamic reconfigurations at runtime.
Finally, we implemented such adaptation policies in various case studies using frameworks such as Fractal and FraSCAti. We also use behavioural fuzzing to test various specific aspects of a given adaptation policy.

Keywords: Components, Adaptation, Reconfiguration, Dynamic reconfiguration, Temporal properties

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ