Composition comportementale de composants

Mikaël Beauvois

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Systèmes et Logiciels

préparée dans le Laboratoire DTL/ASR de France Telecom R&D
et dans le Laboratoire I3S de l'Université Nice Sophia-Antipolis

dans le cadre de **l'École Doctorale**
Mathématiques, Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement

par

Mikaël Beauvois

le 29 septembre 2005

Titre :

Composition comportementale de composants

Directeur de thèse :
Michel Riveill

JURY

Mme Christine Collet	Président
Mme Laurence Duchien	Rapporteur
M. Philippe Lahire	Rapporteur
M. Michel Riveill	Directeur de thèse
M. Pascal Déchamboux	Co-directeur de thèse

Résumé

Résumé

L'évolution des besoins des logiciels entraîne la croissance de la complexité des environnements répartis. La recherche effectuée dans le domaine de la conception de ces environnements vise à réduire cette complexité. Un des principaux problèmes de la conception des infrastructures réparties concerne la composition des propriétés non fonctionnelles (également appelées services techniques). Les services interagissent entre eux. Nous avons identifié deux types d'interaction : les interactions de type structurel et les interactions de type comportemental.

Il existe actuellement de nombreuses approches (académiques et industrielles) qui permettent de concevoir ces infrastructures. Dans un premier temps, nous exposons les concepts de la composition et nous étudions les mécanismes de composition mis en oeuvre dans ces approches de conception. A partir de cette étude, nous proposons une nouvelle approche de composition appelée composition comportementale qui permet de supprimer un certain nombre de limites identifiées dans les autres approches. L'approche de composition comportementale utilise le modèle de composants Fractal et introduit un modèle d'automates qui permet de décrire les comportements des composants. Les interactions de type structurel s'expriment à partir du modèle de composants et se matérialisent par des liaisons entre les interfaces des composants. Les interactions de type comportemental s'expriment à partir du modèle d'automate et se matérialisent par des contraintes d'ordonnancement. Les mécanismes de composition de notre approche mettent en oeuvre ces différents types d'interaction.

Nous avons réalisé un canevas logiciel qui implante le modèle de composant et le modèle de comportement. Le canevas a été conçu afin que les approches de conception puisse l'utiliser. L'implantation du canevas génère un environnement d'exécution basé sur le langage synchrone réactif Esterel.

Pour conclure, nous positionnons notre approche avec les autres approches de conception à partir de critères d'évaluation que nous avons définis. Quelques perspectives concernant l'approche sont données.

Mots-clés

composition, comportement, adaptabilité, canevas logiciel, composant, services techniques, intergiciel, ordonnancement

Abstract

Evolution of software needs leads to increase the complexity of distributed environment. Research in sotfware engineering aims to decrease this complexity. One of main issues in design of distributed infrastructures concerns the composition of non functional

properties (also called technical services). Services interact each other. We have identified two kinds of interaction : structural interactions and behavioral interactions.

Currently, there are a lot of academic and industrial approaches to build those infrastructures. At first, we expose concepts of composition and we study the mechanisms of composition implemented in those approaches. With this survey, we propose a new composition approach called behavioral composition that suppresses the limits identified in the other approaches. The behvioral composition approach uses the Fractal component model and introduces an automata model to describe the components behaviors. The structural interactions are expressed from the component model et are realized with bindings between the interfaces of components. The behvioral interactions are expressed from the automata model and are realized with scheduling constraints. The mechanisms of composition implement these kinds of interaction.

We made a software framework that implements the component model and the behavior model. The framework has been designed to be used by other approaches. The implementation of the framework generates an execution environment based on the reactive synchronous Esterel language.

To conclude, we evaluate our approach with evaluation criteria we have defined. Some perspectives are given.

Keywords

composition, components, behavior, non functional properties, adaptable middleware

Remerciements

Cette thèse a été effectuée dans le laboratoire DTL/ASR de France Télécom Recherche et Développement en coopération avec le laboratoire I3S de l'Université Nice Sophia-Antipolis.

Je tiens à remercier Mme Christine Collet, Professeur à l'Institut National Polytechnique de Grenoble, d'avoir accepté de présider le Jury de cette thèse. Mes remerciements vont également à Mme Laurence Duchien, Professeur de l'Université de Lille, et à M. Philippe Lahire, Maître de Conférences de l'Université Nice Sophia-Antipolis d'avoir accepté d'être les rapporteurs de cette thèse.

Je souhaite remercier tout particulièrement M. Michel Riveill, directeur de cette thèse, pour son soutien et sa confiance malgré les événements malheureux qui se sont produits durant la thèse. Sa patience et son ouverture d'esprit sont de grandes qualités.

Je remercie également Kathleen Milsted, responsable du laboratoire MAPS/AMS (anciennement DTL/ASR) de France Télécom R&D, et Pascal Déchamboux, pour leur accueil au sein du laboratoire et leur confiance qu'ils m'ont accordée tout au long de la thèse.

Je souhaite exprimer un grand merci à Mourad qui, par ses qualités humaines, m'a vraiment encouragé à aller jusqu'au bout, et à Romain, avec qui j'ai eu des discussions hautement philosophiques.

Je remercie tous mes collègues, notamment Marc, qui m'a donné tout au long de la thèse de précieux conseils et m'a aidé à prendre les bonnes décisions, Thierry, qui m'a toujours soutenu et encouragé dans toutes les situations, Eric, Aimé, Jacques, Bruno, Serge, Mohamed, Christian, Jean-Charles, Alexandre, Vincent, Nicolas, Patrice, et tous les membres du laboratoire DTL/ASR. Toutes ces personnes m'ont aidé, chacune à leur manière, à progresser dans la vie.

Je remercie également Jean-Bernard Stefani, directeur de recherche dans le projet SARDES de l'INRIA, pour sa disponibilité et son écoute.

Je tiens à exprimer une profonde amitié à tous les membres (anciens et actuels) de l'équipe Nods (LSR-IMAG) et plus particulièrement Stéphane et Patricia, qui ont toujours été présents lors des moments difficiles et qui m'ont toujours aidé à les surmonter, Luciano, Tanguy, Claudia et Khalid.

Je tiens également à remercier l'équipe Rainbow (I3S), particulièrement Mireille, Anne-Marie et Audrey.

Je souhaite saluer toutes les personnes que j'ai rencontrées dans le cadre du projet Arcad qui m'ont permis de progresser dans la thèse.

Je tiens à remercier chaleureusement Séverine, responsable de mastère spécialisé à Grenoble Ecole de Management, pour son soutien et sa confiance sans faille et qui a toujours été disponible lors des moments difficiles, ainsi qu'à mes collègues Emmanuel, Serge, Daniel-Marie, Carole, Benoît qui ont toujours cru dans mes capacités et m'ont encouragé à finir la thèse.

J'ai une profonde amitié pour Jean-Yves, Sébastien, Jean-Philippe, Gilles, Sylvain, Nicolas, Delphine qui m'ont encouragé, soutenu, réconforté durant les moments de doute et de découragement. Sans leur amitié, cette aventure n'aurait pas pu être possible. Je tiens également à saluer plus généralement tous mes amis.

Je ne terminerai pas les remerciements sans avoir une pensée pour l'ensemble des professeurs que j'ai eu durant toute ma scolarité et avec qui j'ai pu créer des liens d'amitié. Rien n'aurait été possible sans leur dévouement.

Pour finir, c'est avec une grande émotion que je dédie ce mémoire, ce travail de longue haleine, à ma douce et tendre maman qui aurait tant aimé être parmi nous pour partager ces moments, la conclusion heureuse de cette belle aventure. Les quelques moments difficiles que j'ai traversés ne représentent rien comparés aux terribles souffrances et aux combats que tu as mené contre la maladie ...

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte

Il existe de nombreux défis dans le domaine des systèmes répartis et la recherche dans ce domaine est très active. Les infrastructures logicielles réparties doivent gérer des entités hétérogènes réparties. L'évolution des besoins des logiciels entraîne la croissance de la complexité des environnements répartis. La recherche effectuée dans le domaine de la conception de ces environnements vise à réduire cette complexité. L'un des objectifs de la recherche est de rendre ces infrastructures adaptables. L'adaptabilité permet de construire les infrastructures en fonction des besoins des applications et des ressources de l'environnement d'exécution. Une des propriétés de l'adaptabilité est la composabilité des entités qui les constituent. Dans ce mémoire, nous nous focaliserons sur la composition des propriétés non fonctionnelles (également appelées services techniques) qui constituent les infrastructures réparties.

1.2 Objectifs et démarche

L'objectif de ce mémoire est de définir une approche de composition pour composer les services. Cette approche de composition repose sur un certain nombre de concepts (introduits dans la partie 2.2). L'objectif de cette approche est de palier les limites des approches de conception actuelles qui permettent de construire ces infrastructures. La première étape de ce travail est d'étudier les approches de conception et d'identifier leurs limites. La deuxième étape est de proposer une approche de composition qui permet de supprimer les limites des approches de conception étudiées dans la première étape. En ce qui concerne la troisième étape, plusieurs alternatives sont possibles :

- un travail formel autour de la nouvelle approche de composition : de nombreux travaux existent dans le cadre du domaine des composants [76, 128, 129, 40], du comportement [51, 62], et de la concurrence [123, 79, 51]. Avec ce travail, la démarche aurait été de prouver certaines propriétés, notamment celles liées à la sûreté de la composition, à la vérification, etc,
- un travail d'architecture logicielle dirigé vers la résolution d'un problème spécifique : le but de ce travail est de fournir un cadre logiciel qui permette d'implanter la nouvelle approche.

Nous avons choisi cette deuxième alternative qui répond mieux aux attentes des projets dans lesquels nous étions impliqués [10]. Cette alternative définit le déroulement et les résultats de la thèse en un prototype logiciel qui intégre les concepts définis dans la suite. Un travail formel peut maintenant être envisagé et dérivé des résultats apportés par ce mémoire, mais cela dépasse nos prérogatives initiales. Nous donnerons des perspectives dans ce domaine dans la conclusion.

Le résultat de ces travaux n'est pas de donner un modèle pour la définition de services, ni un modèle pour l'intégration de tout type de services techniques au sein d'applications, mais de définir les mécanismes de composition de services en particulier dans le cas où les services sont liés (non indépendance). Ce travail ne vise pas à prendre parti dans le cadre de la composition de services techniques pour une technologie en particulier. Le résultat doit fournir les éléments nécessaires à la composition indépendamment de la plate-forme cible. En d'autres termes, ce travail n'a pas pour objectif de fournir une infrastructure dédiée à la composition de services techniques (l'état de l'art montre qu'il n'existe pas d'architecture logicielle universelle et minimale pour réaliser n'importe quelle composition de services), mais de donner une approche et donc un cadre architectural pour composer des services, plus particulièrement lorsque ceux-ci interagissent entre eux.

Un exemple sera détaillé pour illustrer les différents mécanismes de la nouvelle approche de composition.

1.3 Exemples

1.3.1 Exemple 1

Cette partie présente un exemple qui illustre un cas de composition de propriétés non fonctionnelles. Il sera décliné dans les différentes approches détaillées dans l'étude des mécanismes de composition (chapitre 2) ainsi que dans notre approche (chapitre 3).

L'exemple est un cas de composition de deux propriétés non fonctionnelles qui sont :

1. la *persistance* : elle est implantée par un gestionnaire de stockage. Ce gestionnaire charge les données relatives à l'application. Ces données sont compressées, aussi le gestionnaire applique un algorithme de décompression avant d'assigner les valeurs.
2. la *sécurité* : elle est implantée par un gestionnaire de sécurité qui vérifie si les utilisateurs authentifiés peuvent exécuter les méthodes de l'application et enregistre toutes les opérations effectuées et un gestionnaire de cryptage qui crypte les données de l'application.

Ces gestionnaires peuvent être conçus indépendamment l'un de l'autre : la conception du gestionnaire de persistance ne dépend pas de la manière de concevoir le gestionnaire de sécurité (et inversement) ainsi que de toute autre implantation d'autres propriétés non fonctionnelles.

Ils peuvent également fonctionner de manière indépendante :

1. le gestionnaire de stockage peut charger les données sans vérifier les permissions des utilisateurs,
2. le gestionnaire de stockage ne crypte pas les données chargées,
3. le gestionnaire de sécurité ne charge pas, ni ne crypte les données,
4. le gestionnaire de cryptage, crypte les données sans savoir si elles sont stockées.

Lors de leur composition, il apparaît que l'exécution des services impose l'entrelacement de l'exécution des deux gestionnaires. En effet, le gestionnaire de stockage ne peut pas charger les données avant que le gestionnaire de sécurité ne vérifie les permissions des utilisateurs. De plus, les informations enregistrées dans le journal reposent sur les valeurs de l'application, qui sont chargées et décompressées par le gestionnaire de stockage. Il apparaît que l'exécution correcte de ces services doit générer la séquence d'exécution suivante : vérification des permissions des utilisateurs, chargement des données, décompression des données, enregistrement des opérations dans un journal.

La conception d'une infrastructure basée sur ces deux propriétés non fonctionnelles s'appuie sur des mécanismes de composition. Ces propriétés non fonctionnelles sont implantées au sein de plate-formes d'exécution répartie.

1.3.2 Exemple 2

Cet exemple montre l'utilisation de deux services : un service transactionnel et un service de gestion de fichiers. Le service de gestion de fichiers enregistre des données applicatives au sein d'un fichier. L'utilisation des deux services fait que le service de gestion de fichiers ne peut s'exécuter que dans le cadre d'une transaction. Ainsi, l'opération d'ouverture de fichier ne peut s'effectuer qu'après le début de la transaction. L'opération de fermeture du fichier doit s'effectuer avant la fin de la transaction.

1.3.3 Exemple 3

Cet exemple présente un cas de composition de deux services. Un premier service permet de gérer la mobilité des objets en fonction de la charge de la plateforme d'exécution. Un deuxième service sélectionne la plateforme la plus apte à exécuter les objets (c'est-à-dire celle dont la charge est la plus faible). Le premier service utilise les fonctionnalités du deuxième pour sélectionner la nouvelle plateforme avant de déplacer l'objet.

1.4 Plan du mémoire

Le mémoire est constitué de plusieurs chapitres :

1. Le chapitre 2 est une étude sur les mécanismes de composition des approches qui servent à la conception des infrastructures à base de services techniques.
2. Le chapitre 3 expose une nouvelle approche de composition qui résoud un certain nombre de limites des approches étudiées dans le chapitre 2 : l'approche de composition comportementale. Il détaille les modèles (composant, comportement, concurrence) qui sont utilisés dans cette proposition et illustre les mécanismes de composition mis en oeuvre.
3. Le chapitre 4 présente la mise en oeuvre de l'approche de composition comportementale sous la forme d'un canevas de composition nommé Brenda.
4. Le chapitre 5 détaille quelques points de l'implantation des canevas présentés dans le chapitre 4, et explique l'exécution de l'exemple résolu par cette nouvelle approche,
5. Le chapitre 6 évaluera notre approche selon les mêmes critères que ceux du chapitre 2 et concluera sur les contributions et les limites de notre approche et de son implantation, et donnera quelques perspectives.

1.5 Synthèse

Afin d'atteindre les objectifs présentés dans la partie 1.1, le travail suit la méthodologie présentée dans la partie 1.4. Pour illustrer les résultats, l'exemple de la partie 1.3.1 sera détaillé dans chaque chapitre pour suivre l'avancée des travaux. Cet exemple permettra de voir :

1. comment sont mises en oeuvre les approches du chapitre 2,
2. comment la nouvelle approche de composition présentée dans le chapitre 3 le modélise,

3. comment le canevas détaillé dans le chapitre 4 l'implante,
4. son exécution dans le chapitre 5.

Chapitre 2

Etude des mécanismes de composition

2.1 Introduction

Ce chapitre s'intéresse aux mécanismes de composition utilisés dans les approches qui servent à exécuter des applications, construites par assemblage de composants et utilisant des services techniques. Il présente les différentes approches qui proposent des mécanismes originaux pour la composition de propriétés non fonctionnelles.

Après avoir présenté brièvement les infrastructures existantes dans la section 2.1.1, un certain nombre de définitions seront données dans la section 2.2 afin d'introduire les concepts de la composition. Le fait de s'intéresser à la problématique de composition de propriétés non fonctionnelles nous fait réfléchir quant aux technologies à employer pour concevoir des infrastructures logicielles capables d'être étendues dynamiquement ou statiquement pour inclure de nouvelles propriétés non fonctionnelles. A partir des concepts de la composition, les mécanismes de composition des différentes approches seront étudiés dans la partie 2.3 et leurs limites seront mentionnées. L'exemple 1.3.1 sera décliné avec toutes les approches et illustrera les limites de ces dernières. Après avoir présenté quelques critères d'évaluation dans la partie 2.5.1, une évaluation détaillée sera réalisée sur l'ensemble de ces approches. Avant de conclure, nous mentionnerons quelques techniques d'implantation utilisées pour mettre en oeuvre ces différentes approches dans la partie 2.6.

2.1.1 Les infrastructures à base de services techniques

Les infrastructures logicielles réparties gèrent des entités (objets ou composants distribués) dans un environnement réparti (les traitements sont exécutés de manière répartie, les données sont également réparties). Ces infrastructures correspondent à la couche logicielle entre les applications et les différents sites interconnectés exécutant chacuns leur propre OS. Cette couche est appelée middleware (ou intergiciel). Elle permet de gérer la répartition et les problèmes qui en découlent tels que les accès aux objets, la préservation de l'état cohérent du système, etc.

Pour gérer la répartition, ces infrastructures utilisent des services techniques qui s'adressent chacun à un problème particulier lié à la répartition. Nous pouvons citer le service de persistance, le service de sécurité, le service de transaction, etc qui sont également utilisés dans les environnements centralisés.

Il existe actuellement différentes infrastructures à base de services techniques. Les plus répandues sont :

- CORBA [102] (Common Object Request Broker Architecture) est une architecture (définie par des entreprises et des institutionnels) qui définit un bus à objets et qui recence un certain nombre de ces fonctionnalités, elles aussi normalisées. Cette

norme a pour but de fournir un socle commun qui permet de faire interopérer des objets répartis.
- J2EE : Sun Microsystems a défini des spécifications (spécifications EJB (Enterprise Java Beans) [56] qui s'intègre dans un environnement Java. Contrairement à Corba qui s'abstrait des langages, Sun s'oriente sur une infrastructure basée sur un environnement entièrement Java. Les spécifications s'appuyent sur une architecture multi-tiers.
- .Net [7] : Microsoft a réalisé un environnement qui, contrairement à l'architecture J2EE qui s'appuye entièrement sur le langage Java, s'oriente sur une interopérabilité des langages en définissant un langage interne Microsoft Intermediate Language (MSIL) et un environnement d'exécution CLR (Common Language Runtime). L'environnement s'appuie sur la couche COM/DCOM [94] pour utiliser les services fournis par la plate-forme.

Une des principales difficultés de la conception de ces intergiciels qui permettent aux applications qu'ils supportent d'utiliser de nombreux services techniques est de les rendre extensibles et d'offrir des mécanismes et des outils pour étendre l'ensemble des services aux constructeurs d'applications.

L'idée est de pouvoir construire ces infrastructures en choisissant les propriétés non fonctionnelles que l'infrastructure doit offrir. Cet objectif ne peut être atteint qu'en définissant des techniques qui offrent la possibilité de concevoir et de composer les services techniques.

Les approches présentées dans cette partie sont utilisées dans le monde académique ou industriel pour la conception d'architectures réparties à base de services techniques. De part la complexité de conception liée à l'environnement réparti, chaque propriété non fonctionnelle (qui se traduit par l'implantation de services techniques pour les mettre en oeuvre) est étudiée et conçue indépendamment des autres. Dans un second temps, cette mise en oeuvre doit être intégrée avec l'existant.

2.1.2 Terminologie : propriété non fonctionnelle et service technique

La notion de propriété non fonctionnelle n'est pas précisément délimitée. La définition communément admise est qu'une propriété non fonctionnelle représente la caractérisation d'une entité [logicielle], cette caractérisation étant liée à l'environnement dans lequel elle s'exécute, et non pas aux fonctionnalités que cette entité doit remplir. Par exemple, la mobilité qui est la faculté que les entités applicatives ont de migrer sur des sites géographiquement éloignés au sein d'un environnement réparti est une propriété non fonctionnelle : elle est liée au contexte d'exécution des entités applicatives et non pas aux fonctionnalités auxquelles doivent répondre ces entités. En règle générale, dans le domaine des systèmes répartis, une propriété non fonctionnelle est une fonctionnalité qui est offerte aux applications fonctionnelles (constituées de propriétés fonctionnelles) et elle est fournie par l'environnement réparti dans lequel l'application s'exécute. Cette fonctionnalité est relative aux capacités de l'environnement réparti et hétérogène.

Cette notion se décline de différentes manières dans les différents paradigmes présentés dans la suite. Dans le domaine des systèmes répartis, le terme le plus employé est *service technique*. Un service technique est la réalisation d'une propriété non fonctionnelle, autrement dit une propriété non fonctionnelle peut se décliner en différents services techniques. Par exemple, la propriété de fiabilité se traduit par l'utilisation conjointe des services techniques de persistance, de duplication, et de gestion transactionnelle. Le terme service technique provient du domaine de l'intergiciel (middleware) où ces plate-formes fournissent diverses fonctionnalités pour que les applications profitent des capacités de

l'environnement d'exécution : répartition, fiabilité, optimisations à l'exécution, etc.

Dans le principe de conception *Separation of Concerns* [80] expliqué dans la partie 2.3.7, une propriété non fonctionnelle se traduit en préoccupation (*concern* en anglais). Dans l'approche par conteneur, une propriété non fonctionnelle est vue sous la forme de service technique (modules intégrés dans une infrastructure logicielle). En résumé, une propriété non fonctionnelle se traduit dans un vocabulaire différent selon le domaine étudié.

Le terme employé par la suite sera service technique pour respecter le vocabulaire du domaine.

2.1.3 Motivation de l'étude

Cette étude présentent les travaux qui concernent la composition de services techniques dans le domaine des systèmes répartis. Cette étude ne tend pas à être exhaustive car la notion de composition se retrouve dans différents domaines qui sont très denses (génie logiciel, systèmes répartis, langages, etc). Son objectif est de présenter les principes mis en oeuvre au niveau langage, au niveau des architectures logicielles, et des intergiciels pour définir, composer et intégrer les services techniques. Pour illustrer ces principes, les concepts de la composition seront définis et déclinés dans les différentes approches utilisées dans le cadre de la construction d'environnements d'exécution répartie (middleware) dans le but d'extraire les mécanismes de composition mis en oeuvre pour composer les services techniques qui sont les constituants de ces environnements.

2.2 Concepts de la composition

Cette partie vise à donner un certain nombre de définitions concernant la composition d'entités. Ces définitions ne se focalisent pas exclusivement sur les services techniques.

2.2.1 Définitions

L'ensemble de ces définitions se base sur la spécification du standard ISO RM-ODP (Reference Model of Open Distributed Processing) [83, 82]. [83, 82] utilise des techniques de description formelle pour spécifier des architectures réparties. Cette spécification s'intéresse notamment aux notions de composition, de comportement, de composition de comportements et de configuration. Ces notions seront détaillées dans cette partie. Elles seront utilisées dans la suite du mémoire.

2.2.1.1 Composition

Dans la spécification RM-ODP [82, 83], la composition est définie comme une opération qui permet de créer un nouvel objet à partir d'une combinaison de deux ou plusieurs objets. Les caractéristiques de ce nouvel objet dépendent :

- des caractéristiques de chacun des objets combinés,
- de la manière dont ces objets sont combinés.

Si nous appliquons la définition à notre étude, les objets correspondent aux services. Les caractéristiques des services sont les fonctionnalités qu'ils offrent. La composition des services consiste à créer des interactions entre les services. Le mécanisme de composition définit la manière dont les services sont combinés et se base donc sur les interactions entre les services.

Exemple L'exemple détaillé dans la partie 1.3.1 est constitué de trois gestionnaires qui doivent être composés ensemble.

Le gestionnaire de persistance a deux fonctionnalités : le chargement de données et leur décompression. Le mécanisme de composition doit permettre de définir les interactions qui existent entre l'algorithme de chargement et l'algorithme de décompression pour que le gestionnaire puisse implanter la fonction de persistance.

2.2.1.2 Comportement

Dans la spécification RM-ODP [82, 83], le comportement est défini comme une collection d'actions avec un ensemble de contraintes qui peuvent se produire.

2.2.1.3 Composition de comportements

Dans la spécification RM-ODP [82, 83], la composition des comportements est définie comme une opération qui permet de créer un nouveau comportement à partir d'une combinaison de deux ou plusieurs comportements. Les caractéristiques de ce nouveau comportement dépendent :

- des caractéristiques de chacun des comportements combinés,
- de la manière dont ces comportements sont combinés.

Si nous appliquons la définition à notre étude, les comportements correspondent aux comportements des services. Les caractéristiques des comportements sont données par le modèle de comportement. La composition des comportements des services consiste à créer les interactions entre les comportements des services. Le mécanisme de composition définit la manière dont les comportements sont combinés. Le mécanisme de composition se base donc sur les interactions qui existent entre les comportements des services.

2.2.1.4 Configuration

Dans la spécification RM-ODP [82, 83], la configuration d'objets est une collection d'objets capables d'interagir aux interfaces. Une configuration détermine l'ensemble des objets impliqués dans chaque interaction. Une spécification d'une configuration d'objet peut être statique ou dynamique avec des mécanismes qui peuvent changer la configuration.

Le résultat de la composition d'objets est une *configuration* d'objets.

2.2.2 Les interactions

Les services interagissent entre eux pour proposer une fonctionnalité issue de leur composition. Les interactions matérialisent les dépendances qui existent entre les services. A partir des définitions données dans la partie 2.2.1, nous avons identifié dans le cadre de notre travail deux types de dépendance :

1. une dépendance dite structurelle,
2. une dépendance dite comportementale.

2.2.2.1 La dépendance structurelle

D'un point de vue structurel, un service *s1* dépend du service *s2* quand, lors de la conception du service *s1*, le service *s1* a besoin des fonctionnalités fournies par le service *s2* pour fonctionner : on dit que le service *s1* dépend structurellement du service *s2*. Le service *s1* définit explicitement la dépendance vers le service *s2*.

Une dépendance structurelle se matérialise par une interaction de type structurel. Les mécanismes qui permettent de définir les dépendances structurelles entre les services seront étudiés dans la partie 2.3.

Exemple L'exemple de la partie 1.3.1 indique que le gestionnaire est composé d'un service de chargement de données et d'un service de décompression de données. L'utilisateur du gestionnaire utilise le service de chargement pour obtenir les données stockées sur un support persistant. Le service de chargement utilise la fonction de décompression offerte par le service de décompression afin de fournir à l'utilisateur du gestionnaire les données décompressées. Le service de chargement est donc dépendant du service de décompression.

2.2.2.2 La dépendances comportementale

D'un point de vue comportemental, un service *s1* dépend du service *s2* quand l'exécution du service *s1* peut influencer l'exécution du service *s2*. Le service *s1* n'a pas de dépendance structurelle avec le service *s2*, c'est-à-dire qu'il n'a pas besoin des fonctionnalités fournies par *s2* pour fonctionner, mais a une dépendance comportementale avec le service *s2*.

Une dépendance comportementale se matérialise par une interaction de type comportemental. Les mécanismes qui permettent de définir les dépendances comportementales entre les services seront étudiés dans la partie 2.3.

Exemple L'exemple de la partie 1.3.1 indique que l'exécution du gestionnaire de persistance est liée à celle du gestionnaire de sécurité. car l'exécution du chargement et de la décompression des données entre en jeu dans l'exécution de l'authentification : les données utilisées dans le cadre de l'authentification correspondent aux données qui sont chargées. Le gestionnaire de persistance n'a pas besoin des fonctionnalités du gestionnaire de sécurité, et le gestionnaire de sécurité n'a pas besoin des fonctionnalités du gestionnaire de persistance, mais leurs exécutions sont dépendantes l'une de l'autre.

2.2.2.3 Composition structurelle et composition comportementale

A partir des définitions de la composition 2.2.1 et des définitions des interactions 2.2.2, nous avons identifié deux types de composition :

1. la composition structurelle,
2. la composition comportementale.

Nous parlerons de composition structurelle lorsque les mécanismes de composition porteront sur les dépendances structurelles. Ces mécanismes permettent donc de définir des interactions de type structurel. Nous parlerons de composition comportementale lorsque les mécanismes de composition porteront sur les dépendances comportementales. Ces mécanismes permettent donc de définir des interactions de type comportemental.

Exemple Les mécanismes mis en oeuvre pour concevoir le gestionnaire de persistance sont issus de la composition structurelle (le service de chargement utilise le service de décompression). Les mécanismes mis en oeuvre pour concevoir la configuration des gestionnaires de persistance et de sécurité sont issus de la composition comportementale.

2.3 Les approches de conception

Cette partie présente une ensemble non exhaustif d'approches qui permettent de concevoir les infrastructures. Ces infrastructures donnent la possibilité aux applications de bénéficier d'un certain nombre de services techniques. La capacité de concevoir des infrastructures adaptables repose sur des techniques qui rendent ces infrastructures modulaires et composables (cf partie 2.5.3.1). Ces techniques sont issues du principe de conception *Separation of Concerns* [80].

Après avoir présenté succintement les approches qui permettent de séparer puis de composer les services techniques, les mécanismes de composition seront étudiés et un certain nombre de limites seront exposées.

2.3.1 Les approches réflexives

La réflexivité (réflexion) est la capacité d'un système à raisonner et à agir sur lui-même pendant sa propre exécution, donc de s'auto-représenter et de s'auto-modifier. [125] formalisent ce concept. Un système réflexif se décompose en deux niveaux : un niveau de base qui correspond à l'application fonctionnelle et un niveau méta (méta-niveau) qui correspond aux propriétés non fonctionnelles. L'introspection est la capacité d'un programme d'observer son propre état et donc de raisonner sur celui-ci. L'intercession est la capacité d'un programme de modifier son propre état d'exécution. La réification est le mécanisme qui offre au programme la possibilité d'agir sur son état d'exécution. Un méta-niveau peut être vu comme l'interprète qui exécute le niveau de base. Un méta-niveau est matérialisé par des méta-objets, qui implantent les fonctionnalités de l'interpréteur. Le niveau de base et le méta-niveau sont reliés par un lien méta qui symbolise la relation entre les objets du niveau de base et les méta-objets. Ce lien est matérialisé par la définition d'un MOP (Meta-Object Protocol ou protocole à méta-objets) [85, 45].

Il existe deux types de réification : la réification statique et la réification dynamique.

2.3.1.1 Réification statique

La réification statique est la réification totale et statique du niveau de base avant l'exécution de ce dernier. Ses structures de données internes sont réifiées.

Elle s'effectue à la compilation (compile-time). Les travaux réalisés dans ce domaine sont relatifs aux compilateurs ouverts (ou méta-compilateurs) tels que OpenJava [9, 134, 47, 133]. Les MOP définis pour chaque prototype correspondent aux points d'extension du compilateur. Généralement ces points d'extension sont en rapport avec les entités de l'arbre syntaxique du langage source : affectation, appel de méthode, création d'objets, etc. Les méta-objets sont des extensions (spécialisations) du compilateur qui le spécialisent pour générer le code adéquat. Ces méta-objets gèrent la méta-information des entités du langage.

2.3.1.2 Réification dynamique

La réification dynamique est la réification dynamique du niveau de base, c'est-à-dire la réification des entités qui entrent en jeu dans le comportement de ce niveau.

Elle s'effectue à l'exécution (run-time). Les travaux réalisés dans ce domaine sont relatifs aux environnements d'exécution (interpréteurs, machines virtuelles, environnements réflexifs) tels que MetaXa [67, 68, 66, 69], Guarana [100, 101], etc. Les Mop correspondent aux points d'extension de l'environnement d'exécution fourni par les différents prototypes. Les méta-objets représentent les extensions de cet environnement et ajoutent dynamiquement (à l'exécution) du comportement au niveau de base exécuté par cet environnement.

2.3.1.3 La composition de méta-objets

Il n'existe pas de mécanisme de composition de méta-objets défini dans cette approche. Les approches réflexives ne prennent pas en compte la composition des méta-objets, sauf en ce qui concerne le prototype Guarana [100, 101] qui définit la notion de méta-configuration : cette notion indique la que la composition des méta-objets est à la charge de l'utilisateur.

La non-orthogonalité intervient dans la composition des méta-objets. La non orthogonalité se définit par l'activation de plusieurs méta-objets lors de la réification d'une opération effectuée par le niveau de base.

Dans [108], la composition de méta-objets s'effectue à partir de relations d'ordre partiel entre méta-objets exprimées à partir de priorités. Des propriétés sont définies sur les *wrappers* (conditional, mandatory, exclusive, modificatory). A partir de ces propriétés, des contraintes d'ordre ont été élaborées pour déterminer les priorités dans l'exécution des méta-objets et ainsi définir l'ordre d'exécution des méta-objets.

Les limites de cette approche viennent du fait que la réflexivité définit les interactions entre le niveau de base et le méta-niveau, mais ne définit pas les interactions entre les méta-objets.

2.3.1.4 Conclusion

Dans cette approche, les services techniques sont conçus sous forme de méta-objets. Ils sont donc considérés comme un ajout de comportement au niveau de base. La réflexivité n'est pas une technique spécifiquement dédiée à la composition de services techniques.

La réification statique considère le niveau de base comme une structure d'entités définissant le comportement de ce dernier (affectation, appel de méthode, destruction d'objet, etc). La réification comportementale considère le niveau de base comme un flot d'exécution.

La réflexivité introduit quelques techniques de base pour concevoir et intégrer des propriétés non fonctionnelles au sein d'applications existantes. Elle a permis de représenter une application existante à partir d'un ensemble d'opérations qui peuvent être réifiées.

2.3.2 Les approches orientées aspects

La programmation orientée aspect (AOP [Aspect Oriented Programming]) introduite par [88, 93, 61, 126, 36, 111] se définit comme étant une extension de la programmation orientée objet. Elle introduit le concept d'aspect et de tissage d'aspects (aspect weaving). L'objectif de ce paradigme de programmation est de se focaliser sur la conception des préoccupations (persistance, mobilité dans le domaine des systèmes répartis) indépendamment les unes des autres et indépendamment du niveau de base (partie applicative).

A la base, les techniques orientées aspect ont pour objectif d'étendre les langages généralement orientés objet avec les implantations les plus répandues (AspectJ [86, 87, 2], JAC [114, 113, 109, 112, 115, 110], etc)) afin d'améliorer la modularité du logiciel pour en simplifier le développement et la maintenance. Le domaine des systèmes répartis souhaite tirer parti de cette modularité pour améliorer l'expression des propriétés non fonctionnelles, telles que la mobilité, la fiabilité, la sécurité, le comportement transactionnel, afin d'en simplifier leur conception.

Un aspect est défini par un ensemble d'implantations à exécuter à certains endroits du niveau de base.

Les constituants de base d'AOP sont :

- le *joinpoint* qui est un élément du programme de base (par exemple, la classe, la méthode, la structure de contrôle, etc),

- le *pointcut* qui est un ensemble de jointpoints logiquement liés ensemble,
- *l'advice* qui correspond au traitement ajouté sur un *pointcut*,
- *l'aspect* qui est l'association des *pointcuts* et des *advices*.

Le tissage d'aspects (*weaving*) est le mécanisme qui consiste à intégrer le code des aspects au niveau de base.

Les mécanismes de composition sont étudiés en fonction des implantations AspectJ [2, 87, 86], JAC [114, 113, 109, 112, 115, 110] et PROSE [119, 121, 120, 97, 98] de l'approche AOP et de l'implantation de l'approche EAOP [3, 60, 58, 59] (Event-based Aspect Oriented Programming).

2.3.2.1 AspectJ

AspectJ [2, 87, 86] est une implantation développée par l'équipe de Gregor Kiczales qui s'est notamment concentré dans l'approche à méta-objets [85]. Il propose un langage qui permet de concevoir les aspects. Ce langage inclut les notions de *pointcut* et d'*advice*. Un pointcut est une collection d'événements que correspondent à des points dans l'exécution d'un programme. Il est conçu à partir de patterns qui se définissent à l'aide de *designators*. Un *designator* correspond au type d'événements qui peuvent être capturés : l'appel à une méthode, l'accès à un champ, etc. Un *advice* correspond à une partie de l'implantation d'un aspect qui sera exécutée dans le cadre d'un point bien défini dans l'exécution d'un programme ; il est défini à partir des *pointcuts* et peut être inséré avant (*before*), après (*after*) ou à la place (*around*) du programme d'origine (écrit en langage Java).

Le prototype compile le code des aspects. A partir du code source du programme de base, il génère le bytecode comprenant le code du programme de base et celui des aspects. Il s'appuie sur des techniques de transformation de programmes.

La composition d'aspects dans AspectJ Les aspects sont définis de manière indépendante les uns des autres. AspectJ ne permet donc pas de définir des interactions de type structurel. Néanmoins, lors de la composition, l'exécution des aspects peut être ordonnée lorsque, lors du weaving, plusieurs aspects sont activés sur le même *joinpoint*. Les aspects sont dits non orthogonaux. L'ordonnancement correspond aux interactions de type comportemental. Ces interactions sont définies au niveau de l'aspect et de manière statique. L'ordonnancement entre les aspects s'effectue à l'aide du mot-clef *dominate* défini au sein du langage d'AspectJ.

Dans les approches orientées aspects, les interactions de type structurel n'existent pas, donc un aspect ne peut pas utiliser les fonctionnalités implantées dans les autres aspects. Les interactions de type comportemental correspondent à une relation d'ordre entre les aspects. Dans AspectJ, cette relation d'ordre est basée sur la notion d'exécution prioritaire d'un aspect par rapport à un autre. Cette exécution prioritaire est statique et la priorité s'appuie uniquement sur l'aspect, donc tous les advices qui constituent cet aspect auront la même priorité par rapport aux autres advices d'un autre aspect, ce qui montre la limite de l'expressivité des relations d'ordre entre les aspects.

2.3.2.2 JAC

JAC [114, 113, 109, 112, 115, 110] est une implantation développée à l'origine par l'équipe Caolac du laboratoire Cedric (CNAM) et maintenant au LIFL. Ce prototype est actuellement utilisé par la société Aopsys [1] pour promouvoir la programmation par aspects. Contrairement à AspectJ, l'objectif n'est pas la définition d'un langage d'aspects, mais la dynamicité de ce type d'approche. La dynamicité est obtenue par manipulation de bytecode à l'aide de Javassist [46] à l'exécution. Il introduit la notion de *wrapper*

dynamique qui correspond à la notion d'*advice*. L'ensemble des *wrappers* sur un même objet est appelé *wrapping chain*.

La composition d'aspects dans JAC Tout comme AspectJ, les aspects dans JAC sont définis de manière indépendante les uns des autres. L'ordonnancement s'effectue en définissant de manière statique (via un descripteur de configuration d'aspects ou via un aspect de composition) la liste ordonnée des aspects et s'appuie donc sur la notion de priorité.

Les limites de JAC dans le domaine de la composition d'aspects sont les mêmes que celles indiquées dans AspectJ.

[107] définit des propriétés sur les aspects afin d'améliorer leur composabilité (*mandatory*, *exclusive*, etc). Néanmoins, ces propriétés se focalisent uniquement sur l'aspect.

2.3.2.3 PROSE

PROSE [119, 121, 120, 97, 98] est la contraction de PROgrammable extenSion of sErvices. Le prototype a été développé par l'équipe de Gustavo Alonso (Zürich). Il s'agit d'une machine virtuelle Java [71]. PROSE fournit une API qui permet d'ajouter ou de retirer dynamiquement les extensions de la JVM, ces extensions étant des aspects. L'objectif est d'intégrer dynamiquement des aspects [48].

La composition d'aspects dans PROSE Tout comme AspectJ et JAC, les aspects dans PROSE sont définis de manière indépendante les uns des autres. Les limites de PROSE dans le domaine de la composition d'aspects sont les mêmes que celles indiquées dans AspectJ et JAC.

L'ordonnancement s'appuie sur la notion de priorités qui définit la relation de précédence dans l'exécution des aspects. Cet ordonnancement est dynamique, et les priorités peuvent dynamiquement être modifiées.

2.3.2.4 EAOP (Event-based Aspect Oriented Programming)

EAOP [3, 60, 58, 59] est une proposition de Rémi Douence et Mario Südholt. Elle correspond à une extension de l'approche AOP. L'idée est d'améliorer l'expressivité des aspects : l'approche AOP conçoit les aspects à partir de la structure du programme, l'approche EAOP préconise la conception des aspects à partir des événements produits par le programme. Ainsi, l'occurrence des événements peut être prise en compte lors de la conception des aspects (par exemple, un insert peut ainsi être exécuté que lors du troisième appel d'une méthode applicative), ce qui n'était pas le cas avec une approche AOP classique.

Le prototype est constitué d'un préprocesseur, d'un moniteur d'exécution et de trois bibliothèques pour la définition des événements et des aspects ainsi que pour la composition de ces derniers.

La composition des aspects est réalisée à partir d'un arbre binaire d'aspects et d'opérateurs de composition. Ces opérateurs sont :

1. *seq* : propage d'abord les événements sur la branche gauche du noeud de l'arbre, puis sur la branche droite ;
2. *any* : propage les événements dans un ordre arbitraire sur les branches du noeud de l'arbre ;
3. *fst* : propage les événements sur la branche gauche du noeud de l'arbre, puis, seulement dans le cas où aucun aspect de la branche gauche n'est concerné par ces derniers, les propage dans la branche droite ;

4. *cond* : propage les événements sur la branche gauche du noeud de l'arbre, puis, si il existe au moins un aspect de cette branche qui est concerné par ces derniers, les propage dans la branche droite.

Ces opérateurs peuvent être combinés pour composer plusieurs aspects (par exemple : A1 *seq* (A2 *fst* A3)). Ces opérateurs sont utilisés lors de la construction de l'arbre binaire. Ainsi, lorsqu'un événement est généré, il est propagé à l'ensemble des aspects composés sous forme d'arbre. Le moniteur d'exécution utilise les opérateurs de composition pour déterminer l'ordre d'exécution des aspects. La propagation s'effectue en profondeur d'abord (depth-first).

L'atout principal de cette approche est la définition des aspects à partir non plus à partir de points dans la structure des programmes, mais à partir des événements que ces derniers produisent. Contrairement aux autres travaux sur les approches orientées aspects, Rémi Douence et al prennent en compte le problème de l'ordonnancement des aspects lors de leur composition.

Rémi Douence et al [60, 58] définissent un cadre formel générique pour la programmation par aspects qui repose sur un calcul de processus de type CSP [39, 78]. permettant de détecter et de résoudre les conflits entre aspects. La résolution de ces conflits s'appuie sur les opérateurs de composition. La non orthogonalité est résolue avec l'ordonnancement des aspects. Cet ordonnancement s'appuie sur la propagation des événements. Cette propagation est raffinée à l'aide des opérateurs de composition. Ainsi, il appartient au responsable de la composition de choisir les opérateurs adéquats et d'agencer les aspects selon les opérateurs par construction de l'arbre.

La composition d'aspects en EAOP Tout comme AspectJ, JAC et PROSE, les aspects dans EAOP sont définis de manière indépendante les uns des autres. EAOP ne permet donc pas de définir des interactions de type structurel. Néanmoins, lors de la composition, l'exécution des aspects peut être ordonnée lorsque les aspects réagissent au même événement. L'ordonnancement s'appuie sur la sémantique des opérateurs qui permettent de construire l'arbre d'aspects. L'évaluation de l'ordre d'exécution des aspects est effectuée dynamiquement par le moniteur d'exécution.

Les avantages d'EAOP par rapport aux autres approches orientées aspects sont que les opérateurs définissent la sémantique de composition des aspects. Cette sémantique repose sur le parcours d'événements dans un arbre, qui est une structure plus élaborée qu'une liste définie par les autres approches.

Les limites d'EAOP sont que tous les événements se propagent de la même façon dans l'arbre, ce qui limite la capacité de composition des aspects. De plus, tous les *advices* d'un même aspect sont évalués de la même manière.

2.3.2.5 Conclusion

L'approche AOP définit la notion d'aspect. Elle met en oeuvre la séparation des *concerns*. Chaque prototype a sa propre technique pour intégrer les aspects au sein de l'application de base (modification de code source, modification de bytecode, machine virtuelle dédiée) : ces techniques sont dérivées des travaux du domaine de la réflexivité.

Le terme composition telle qu'il est défini dans la partie 2.2.1 a un sens différent de celui communément utilisé dans l'approche AOP. Au sein d'AOP, la composition d'aspects se rapporte au tissage (*weaving*) qui regroupe la composition et l'intégration des aspects au sein du code fonctionnel (mise en relation entre la partie fonctionnelle et la partie non fonctionnelle).

Les aspects ne peuvent pas utiliser les fonctionnalités des autres pour fonctionner. Chaque aspect est défini indépendamment des autres. Par exemple, un aspect A1 peut

implanter un service d'accès au réseau. Un aspect A2 qui implante un service transactionnel réparti pourrait utiliser les fonctionnalités d'accès implantées par l'aspect A1. Or, les approches orientées aspects ne permettent pas de le réaliser.

La programmation orientée aspects ne définit pas de solutions concernant les aspects non orthogonaux. Elle présuppose qu'il est de la responsabilité du tisseur (*weaver*) de tisser correctement les aspects et donc de prendre en compte l'ensemble des compositions dont celles où les aspects sont non orthogonaux. Aussi, il appartient aux prototypes de cette approche de fournir leurs propres mécanismes pour traiter la non orthogonalité des aspects. Résoudre la non-orthogonalité revient à déterminer un entrelacement correct de l'exécution des aspects, ces derniers étant structurellement indépendants. La non-orthogonalité des aspects est résolue au sein de ces implantations par l'ordonnancement entre les aspects. Chaque prototype propose sa propre définition de politique d'ordonnancement. Ces mécanismes sont limités par le fait que l'expression de l'ordonnancement s'adresse aux aspects dans leur totalité et non pas sur une granularité plus fine comme par exemple l'*advice* : l'ordonnancement entre deux *advices* sera le même qu'un autre couple d'*advices* définis sur des points de jonction différents.

[89] s'intéresse aux dépendances inter-aspects. Ils définissent ainsi trois types de dépendance : *orthogonale* (où l'aspect est indépendant des autres pour s'exécuter), *unidirectionnelle* (où l'exécution de l'aspect dépend des fonctionnalités offertes par les autres aspects) et *circulaire* (où plusieurs aspects sont mutuellement dépendants). A partir de ces dépendances, un graphe de dépendance de configurations d'aspects est construit et permet donc de tisser les aspects en respectant leurs dépendances.

2.3.3 Les approches orientées patrons de conception

Les patrons de conception définis dans [63] sont des éléments pour la conception de logiciels orientés objet réutilisables. Ils sont utilisés dans les autres approches pour concevoir les techniques qui mettent en oeuvre les concepts formulés dans ces approches.

Les patrons qui peuvent être utilisés pour mettre en oeuvre la composition de services sont par exemple les patrons *Decorator*, *Strategy* et *Mediator*. Le patron *Decorator* permet de rajouter dynamiquement des responsabilités à un objet. Le patron *Strategy* permet d'implanter plusieurs algorithmes pour une même méthode. Le patron *Mediator* définit un objet qui encapsule les interactions d'un ensemble d'objets.

Les mécanismes de composition de services sont étroitement liés à la manière de combiner ces patrons pour implanter ces mécanismes. Le résultat de ces combinaisons se retrouvent dans les autres approches.

La composition de services avec les patrons Ces patrons permettent d'apporter les concepts d'architecture logicielle pour concevoir les infrastructures. Ils n'apportent aucun mécanisme de composition pour mettre en oeuvre les interactions de type structurel ou comportemental. La composition des services est à la charge du concepteur. Ils sont néanmoins cités car ils participent à la conception d'infrastructures.

Les patrons de conception ne font aucune distinction entre partie fonctionnelle et partie non fonctionnelle.

2.3.4 L'approche "interactions logicielles"

Cette approche [34, 33, 22, 21, 23] se focalise sur le problème de la dynamicité dans la composition. Elle permet de fournir une architecture pour l'intégration dynamique (à l'exécution) de services techniques au sein d'infrastructures telles que les infrastructures

à base d'EJB [56] ou de CCM [103, 127]. Le mécanisme se base sur le concept d'interaction logicielle qui décrit comment le comportement d'un objet devrait changer quand il interagit avec un autre. Une interaction est une instance d'un schéma d'interaction. Ce schéma décrit le comportement du lien entre l'application et le service technique. Il est défini à partir du langage ISL (Interaction Specification Language).

2.3.4.1 La composition d'interactions

La composition des interactions se base sur les mécanismes de fusion comportementale définis dans [22]. La fusion comportementale permet de générer le comportement issu de la composition des interactions. La fusion n'intervient que dans le cas où les interactions sont activées sur un même message déclencheur, ainsi la fusion est le mécanisme mis en oeuvre pour résoudre la non-orthogonalité.

La fusion comportementale est constituée par des règles de réécriture appliquées sur les schémas des interactions et basé sur la sémantique des opérateurs. Pour simple illustration, une des règles est qu'une fusion peut résulter en une parallélisation des appels aux services. La composition dans l'approche orientée interactions est comportementale. Les règles de réécriture définissent l'entrelacement de l'exécution des interactions posées. Le grain de l'ordonnancement s'effectue au niveau des instructions définies dans le schéma d'interaction.

Dans l'implantation de ce modèle, ces règles peuvent être exécutées dynamiquement. Le prototype actuel est limité dans le sens où il n'offre pas la possibilité de définir et d'appliquer (statiquement / dynamiquement) de nouvelles règles de fusion d'interactions. Par exemple, dans le cas d'une fusion de deux interactions séquentielles, le résultat est une exécution concurrentielle des comportements des interactions. Or il n'existe aucun mécanisme qui permette de paramétrer cette fusion, comme par exemple exécuter en séquentiel les comportements des interactions (selon un ordre défini) et non pas de manière concurrente.

Une des extensions possible à l'approche ISL serait le paramétrage de la politique de fusion des règles d'interaction (en étendant les mécanismes de fusion) afin d'affiner le résultat de la composition.

2.3.4.2 Conclusion

La composition dans l'approche par interactions repose sur une composition des interactions (appelée fusion) qui permet la composition des services techniques. L'approche par interactions logicielles diffère des autres approches dans le sens où la fusion des interactions permet d'entrelacer leurs exécutions.

2.3.5 Les approches orientées conteneur

[52] définissent le terme de *container* (conteneur en français) comme une entité responsable des propriétés non fonctionnelles au nom d'un composant logiciel.

Cette approche fait clairement la distinction entre code métier et code technique, contrairement aux autres approches précédemment étudiées. La notion de code métier fait référence aux propriétés fonctionnelles de l'application (les fonctionnalités qu'elle met en oeuvre), alors que la notion de code technique fait référence aux propriétés non fonctionnelles (les services techniques qu'elle utilise afin de réaliser cette mise en oeuvre). Les implantations de cette approche utilisent les objets d'interposition.

Au niveau industriel, cette approche est implantée dans les technologies qui permettent de déployer les composants EJB (Enterprise Java Beans) [56] (environnement J2EE (Java 2 Enterprise Edition)) et CCM (Corba Component Model) [103, 127] : leur modèle de

programmation est basé sur la notion de conteneur qui fait le lien entre le code applicatif implanté sous forme de composants (composant EJB ou composant CCM) et le code des services techniques (code non fonctionnel). La suite détaille les mécanismes de composition mis en oeuvre dans deux implantations OpenSource de la spécification J2EE à savoir JOnAS [5] et JBoss [4]. Ces mécanismes reposent sur le fait que les services techniques sont identifiés, en l'occurrence la sécurité, la persistance et la gestion des transactions et que les types de bean sont limités (*MessageDrivenBean*, *EntityBean* et *SessionBean*). La composition est donc statique. Dans le cadre de la définition d'un bean, ces services ne sont pas tous utilisés, et leur configuration s'établit à partir d'une liste de propriétés identifiées (par exemple, pour la gestion des transactions, le modèle transactionnel est fixé par les spécifications EJB et les attributs transactionnels sont choisis dans une liste pour définir le comportement transactionnel du bean). Dans ces conditions, toutes les combinaisons issues de la composition de ces services techniques ont été étudiées et donc le résultat de n'importe quelle combinaison est connu. Pour les implantations JOnAS et JBoss, leur objectif est l'intégration optimale des conteneurs dans les environnements multi-tiers.

2.3.5.1 JOnAS

JOnAS [5] est une implantation OpenSource d'un serveur J2EE, conforme aux spécifications EJB 2 [56]. Ce développement s'effectue dans le cadre du consortium ObjectWeb [11], et supporté par les entités Bull, France Télécom R&D et INRIA.

La génération du conteneur s'effectue par génération de code lors du déploiement : le code du résultat de la composition des services techniques est obtenu par un générateur. Cette génération est spécifique aux configurations des services techniques. Le principal intérêt de cette méthode est l'optimisation des performances en terme de temps d'exécution. En effet, l'ensemble du code est entièrement généré, et il n'y a pas d'indirections dûes à la prise en compte de la dynamicité.

En comparaison avec une approche orientée aspect, le fonctionnement de la génération est équivalent à un tissage spécifique des services techniques, et le générateur est équivalent à un weaver dont la composition et l'ordonnancement des services sont codés dans ce weaver. Contrairement à tout ce qui a été vu précédemment, il n'est pas possible d'intégrer de nouveaux services ni de manière statique (lors du déploiement), ni de manière dynamique (lors de l'exécution). L'ajout de nouveaux services ne peut s'effectuer que par modification du générateur (il est nécessaire d'en concevoir un autre). Pour cette raison, cette conception n'introduit pas de mécanismes de composition.

2.3.5.2 JBoss

JBoss [4] est également une implantation OpenSource d'un serveur J2EE conforme aux spécifications EJB 2 [56]. Cette implantation est réalisée par le groupe JBoss. La génération du conteneur a lieu lors de la phase de déploiement des composants EJB.

Dans la version 3 du prototype, la génération du conteneur s'effectuait dynamiquement en instantiant les classes correspondant aux configurations des services techniques décrites dans le descripteur de déploiement. Contrairement au prototype JOnAS, il n'y a pas de générateur de code. L'ajout de nouveaux services s'effectue en modifiant l'architecture logicielle du conteneur. Cette dernière a été complètement définie à partir des spécifications EJB.

Dans la version 4 du prototype, la génération du conteneur s'appuie sur une approche orientée aspect ([44]). Cette orientation a été prise pour anticiper l'introduction de nouveaux services techniques dans les futures spécifications EJB. La manipulation de bytecode (Javassist [46]) permet d'insérer les intercepteurs. Les mécanismes de composition sont ceux qui ont été présentés dans la partie 2.3.2.

2.3.5.3 OpenCCM

OpenCCM [91] est une implantation OpenSource de la spécification du modèle de composants CORBA CCM [103, 127] définie par l'Object Management Group (OMG). Ce développement s'effectue dans le cadre du consortium ObjectWeb [11]. OpenCCM permet de concevoir, implanter, compiler, assembler, déployer, configurer et exécuter les applications à base de composants.

Le conteneur fournit des interfaces pour les services Corba (sécurité, transactions, persistance, et notification). La conception du conteneur suit l'approche du POA définit dans Corba. L'introduction de nouveaux services implique une nouvelle implantation du conteneur.

2.3.5.4 La composition de services au sein des conteneurs

Les services au sein des conteneurs sont tous identifiés et décrits dans la spécification EJB [56] ou CCM [103, 127]. La conception des conteneurs suivent la spécification et ils n'ont donc pas été implantés pour prendre en compte de nouveaux services. Aussi, il n'existe aucun mécanisme de composition défini dans les approches orientées conteneur. L'entrelacement issu de la composition des services est déterminé lors de la conception des conteneurs, ce qui limite la possibilité d'ajouter de nouveaux services.

La dernière version de JBoss a toutefois pris en compte le fait que les spécifications évoluent et donc, pour faciliter l'implantation de nouveaux conteneurs, la conception de ces derniers doit intégrer des mécanismes de composition. L'approche choisie est une approche orientée aspects.

2.3.5.5 Conclusion

La notion de conteneur permet d'avoir un cadre architectural qui permet de définir les interactions entre la partie métier et la partie technique. Les implantations actuelles se focalisent sur la phase de déploiement des composants, notamment sur la génération des conteneurs qui interagissent avec les services techniques (configurés dans le descripteur de déploiement). La composition des services techniques, c'est-à-dire les interactions entre eux, est clairement identifiée car les services et leur fonctionnement sont identifiés et leur nombre est restreint.

Dans le cadre du conteneur EJB, les spécifications EJB [56] définissent uniquement le fonctionnement de base de chaque service technique (sécurité, comportement transactionnel, persistance). et un certain nombre de cas à traiter lors de la composition de ces derniers. Elles n'indiquent pas comment intégrer de nouveaux services.

2.3.6 Les approches orientées composants

L'objectif des approches orientées composants est de modéliser un système complexe en unité élémentaire de composition appelée composants. Une définition du terme composant est donnée par Szyperski : "A sofware 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" [128].

Les approches orientées composants ne sont pas utilisées pour concevoir les infrastructures. Elles sont généralement utilisées pour concevoir les applications. Néanmoins, leurs mécanismes de composition sont étudiés dans ce chapitre. Les langages de description d'architecture (ADL (Architecture Description Language) [92, 65, 64]) définissent la

notion de connecteur qui permet de définir les interactions de type structurel entre les composants.

Le modèle CCM [103, 127] introduit la notion de port. Ces ports permettent de connecter les composants CCM entre eux. La composition consiste donc à interconnecter les ports des composants.

Le modèle EJB [56] introduit des composants EJB. La création d'un lien vers un *bean* est réalisé en récupérant sa référence. La composition des composants est effectuée dynamiquement dès lors qu'un composant a besoin de la fonctionnalité offerte par un autre composant.

Le modèle Fractal [42, 41] introduit la notion de connecteur entre interfaces de composants. Ce modèle sera détaillé dans la partie 3.2. La composition des composants consiste à définir des configurations de composants en créant des liaisons entre les interfaces fournies et requises des composants.

En conclusion, les modèles de composants permettent de concevoir des assemblages de composants. Ces assemblages sont des configurations. Certains modèles peuvent définir des interactions de type structurel en créant des liaisons entre les composants. Les dépendances structurelles s'expriment par les points d'accès aux composants (port pour les composants CCM, interface pour les composants Fractal). La difficulté apparaît dès lors qu'il n'existe pas de dépendance structurelle entre les composants. L'exécution d'un composant ne doit pas influencer l'exécution des autres.

2.3.7 Synthèse

Les techniques de *Separation of Concerns,* littéralement *séparation des préoccupations*, se sont focalisées sur la définition des *concerns* (préoccupations) [80]. L'implantation des prototypes s'est appuyée sur les technologies mises en avant par la réflexivité. Dans les techniques de *Separation of Concerns*, les propriétés non fonctionnelles sont traduites par des *concerns*.

Ainsi, les mécanismes de composition de chaque approche seront dérivés soit du domaine des langages, soit du domaine des architectures logicielles. En fonction de l'approche considérée, les services techniques sont conçus :

1. soit à partir d'une orientation *langage*. Pour illustrer cette orientation, les travaux existants tendent à créer un langage spécifique pour définir les services (AspectJ [86, 87, 2]), à mettre en avant certaines propriétés du langage dans lequel est écrite l'application fonctionnelle pour appliquer les services techniques, etc.
2. soit à partir d'une orientation *architecturale*. Les travaux existants cherchent à modéliser les concepts architecturaux fondamentaux (patrons de conception [63], interactions, conteneurs, etc) pour la composition de services techniques et leur mise en relation avec la partie fonctionnelle.

La philosophie des approches orientées langage est de concevoir, composer et appliquer les propriétés non fonctionnelles à partir d'une orientation purement langage. Cette orientation cherche à déterminer quelles sont les abstractions à mettre en oeuvre au niveau langage pour séparer les propriétés non fonctionnelles des propriétés fonctionnelles. Les principales approches sont l'AOP (Aspect Oriented Programming) (partie 2.3.2), les filtres de composition (Composition Filters) [24, 12, 25, 13], SOP (Subject Oriented Programming) [75, 105, 104] et l'approche Hyperspace [106, 132]. Seule l'approche AOP a été étudiée. Les filtres de composition [24, 12, 25, 13], et le prototype ComposeJ [136], SOP [75, 105, 104], l'approche Hyperspace [106, 132] et le prototype associé HyperJ [130, 131], ne sont pas mentionnés car ces approches ne sont pas ou très peu utilisées pour la conception d'infrastructures. La réflexivité définit les points techniques clés qui sont mis en

oeuvre dans les prototypes implantés dans ces techniques. Dans un premier temps, le domaine de la réflexivité a été abordé. Les axes de recherche de la réflexivité s'orientent sur les architectures à mettre en oeuvre (compilateurs ouverts, environnement d'exécution réflexifs, etc) entre le niveau de base et le niveau méta et ainsi mettre en avant les liens méta qui existent entre ces niveaux. Les résultats de ces travaux ont permis de s'abstraire de la technologie qui sert à générer le méta-niveau pour orienter l'axe de recherche sur la conception appropriée des propriétés non fonctionnelles afin d'améliorer le pouvoir d'expression des préoccupations définies dans les techniques de *Separation of Concerns*.

Les approches orientées architecture logicielle regroupe les travaux qui se concentrent sur les concepts architecturaux de composition appliqués aux services techniques. Les principales approches sont les patrons de conception (partie 2.3.3), les interactions logicielles (partie 2.3.4), les conteneurs (partie 2.3.5) et les composants (partie 2.3.6).

2.4 Exemple

Cette partie détaille les implantations de l'exemple 1.3.1 avec les différentes approches étudiées. Afin de ne pas surcharger cette partie, le code de l'exemple est donné en annexe. L'objectif est d'illustrer les mécanismes de composition structurelle et comportementale introduits dans ces approches.

L'exemple 1.3.1 distingue un service de persistance et un service de sécurité.

Les services sont définis par trois gestionnaires : le gestionnaire de stockage, le gestionnaire de sécurité et le gestionnaire de cryptage. Cette partie montre comment ces gestionnaires sont implantés dans chaque approche.

Les gestionnaires sont conçus indépendamment les uns des autres. Lors de leur composition, un certain ordre doit être respecté :

1. les données ne peuvent être chargées en mémoire qu'à partir du moment où l'utilisateur a été authentifié,
2. les informations concernant les opérations effectuées par l'utilisateur enregistrées par le gestionnaire de sécurité contiennent des valeurs relatives à l'application : elles ne pourront donc être enregistrées qu'à partir du moment où le gestionnaire de stockage les aura chargées et décompressées.

2.4.1 Les approches réflexives

Les gestionnaires sont implantés sous la forme d'objets.

Algorithm 1 Composition des méta-objets

```
 1 public class MyMetaObjectsComposition extends OJClass {
 2  public void translateDefinition () throws MOPException {
 3    OJMethod[] methods = getDeclaredMethods();
 4    for (int i = 0; i < methods.length; i++) {
 5      Statement persistance = makeStatement
 6          ("persistanceService.load("+...);
 7      Statement security = makeStatement
 8          ("securityService.checkPermissions("+...);
 9      methods[i].getBody().insertElementAt(security,0);
10      methods[i].getBody().insertElementAt(persistance,0);
11    }
12  }
13 }
```

Le code (algorithme 1) montre que le méta-objet définit par la classe *MyMetaObjects-Composition* (ligne 1) prend en compte l'intégration des services au sein de l'application. L'appel au gestionnaire de persistance (lignes 5,6 et 10) et de sécurité (lignes et 8) est inséré au code applicatif. La composition est statique et il n'existe aucune interaction de type structurel entre les méta-objets.

2.4.2 Les approches orientées aspects

L'exemple est décliné selon les quatre implantations AspectJ, JAC, PROSE et EAOP. Chaque gestionnaire est décliné en aspect. Cette déclinaison a été choisie afin de se rapprocher le plus à une approche à base de composants.

2.4.2.1 Exemple en AspectJ

L'exemple s'implante de la manière suivante : l'aspect *StorageAspect* (algorithme 39) implante le gestionnaire de stockage qui définit le service de persistance et les aspects *SecurityAspect* (algorithme 40) et *EncryptionAspect* (algorithme 41) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

Lors du weaving, les trois aspects sont activés sur le même *joinpoint*. AspectJ permet de définir un ordre d'exécution des aspects, ce qui, dans cet exemple, ne permet pas de résoudre la non orthogonalité issue de la composition des aspects : l'aspect *SecurityAspect*, exécuté en premier pour vérifier les permissions, enregistre dans le log des informations qui ne sont pas encore disponibles (ces dernières doivent être chargées par l'aspect *StorageManager*).

L'exemple montre que l'implantation de l'algorithme de chargement et celle de décompression des données sont définies au sein de la même unité : l'aspect *StorageAspect*. L'approche ne donne pas la possibilité de modifier, ni statiquement, ni dynamiquement, l'algorithme de décompression implanté dans l'aspect. Cela réduit les capacités d'adaptabilité offertes par l'approche.

Dans l'exemple, aucun aspect n'est prioritaire à un autre. L'entrelacement s'effectue au niveau des opérations effectuées par les aspects. Le mot-clef *dominate* ne peut donc pas être utilisé sur cet exemple. Il n'est donc pas possible d'utiliser cette approche pour implanter l'exemple.

2.4.2.2 Exemple en JAC

L'exemple s'implante de la manière suivante : l'aspect *StorageAspect* (algorithme 42) implante le gestionnaire de stockage qui définit le service de persistance et les aspects *SecurityAspect* (algorithme 43) et *EncryptionAspect* (algorithme 44) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

L'ordonnancement des aspects est réalisé de manière déclarative dans un fichier (algorithme 2). Il indique également quels paramètres doivent être appliqués lors du weaving des aspects.

Algorithm 2 Activation des aspects

```
// BusinessObject.jac
...
aspects :
   storage storage.acc ...
   security security.acc ...
   encryption encryption.acc ...
```

Les limites sont les mêmes que celles indiquées pour AspectJ.

Dans l'exemple, l'entrelacement s'effectue au niveau des opérations effectuées par les aspects. La liste des aspects ne peut donc pas être utilisée sur cet exemple. Il n'est donc pas possible d'utiliser cette approche pour implanter l'exemple.

2.4.2.3 Exemple en PROSE

L'exemple s'implante de la manière suivante : l'aspect *StorageAspect* (algorithme 45) implante le gestionnaire de stockage qui définit le service de persistance et les aspects *SecurityAspect* (algorithme 46) et *EncryptionAspect* (algorithme 47) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

Après l'instantiation des aspects (lignes 2, 4 et 6), l'ordonnancement s'effectue de manière dynamique avec un *AspectManager* qui gère la liste des aspects activés (algorithme 3) à l'aide de la méthode *insert* (lignes 3, 5 et 7).

Algorithm 3 Activation des aspects

```
1  ProseSystem.startup() ;
...
2  StorageAspect storageAspect = new StorageAspect() ;
3  ProseSystem.getAspectManager().insert(storageAspect) ;
...
4  SecurityAspect securityAspect = new SecurityAspect() ;
5  ProseSystem.getAspectManager().insert(securityAspect) ;
...
6  EncryptionAspect encryptionAspect = new EncryptionAspect() ;
7  ProseSystem.getAspectManager().insert(encryptionAspect) ;
...
```

Les limites sont les mêmes que celles indiquées pour AspectJ.

Dans l'exemple, l'entrelacement s'effectue au niveau des opérations effectuées par les aspects. La liste des aspects ne peut donc pas être utilisée sur cet exemple. Il n'est donc pas possible d'utiliser cette approche pour implanter l'exemple.

2.4.2.4 Exemple en EAOP

L'exemple s'implante de la manière suivante : l'aspect *StorageAspect* (algorithme 48) implante le gestionnaire de stockage qui définit le service de persistance et les aspects *SecurityAspect* (algorithme 49) et *EncryptionAspect* (algorithme 50) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

Après l'instantiation des aspects (lignes 1, 2 et 3), l'ordonnancement s'effectue de manière dynamique avec un moniteur qui gère la structure de l'arbre d'aspects (algorithme 4, lignes 4 et 5).

Algorithm 4 Construction de l'arbre d'aspects

```
1  StorageAspect storageAspect = new StorageAspect() ;
2  SecurityAspect securityAspect = new SecurityAspect() ;
3  EncryptionAspect encryptionAspect = new EncryptionAspect() ;
4  Monitor.monitor.aspects = new Root(
5     new Seq (storageAspect, new Seq(securityAspect, encryptionAspect))) ;
```

Les limites sont les mêmes que celles indiquées pour AspectJ.

Dans l'exemple, l'entrelacement s'effectue au niveau des opérations effectuées par les aspects. L'arbre d'aspects ne peut donc pas être utilisé sur cet exemple. Il n'est donc pas possible d'utiliser cette approche pour implanter l'exemple.

2.4.3 L'approche "interactions logicielles"

L'exemple s'implante de la manière suivante : l'interaction *StorageInteraction* (algorithme 51) définit le lien entre le gestionnaire de stockage qui définit le service de persistance et l'application et les interactions *SecurityInteraction* (algorithme 52) et *EncryptionInteraction* (algorithme 53) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

L'activation des interactions est réalisée en instanciant les interactions et en les enregistrant auprès du service d'interactions (algorithme 5, lignes 5, 6 et 7).

Algorithm 5 Activation des interactions

```
1  String storageInteraction = "...";
    // ISL code of interaction StorageInteraction
2  String securityInteraction = "...";
    // ISL code of interaction SecurityInteraction
3  String encryptionInteraction = "...";
    // ISL code of interaction EncryptionInteraction
...
4  NoahServer noahServer = ...
5  noahServer.registerPattern (storageInteraction);
6  noahServer.registerPattern (securityInteraction);
7  noahServer.registerPattern (encryptionInteraction);
```

Dans l'exemple, les interactions fusionnent car elles sont activées sur un même message déclencheur. La fusion entrelace l'exécution des interactions selon les règles de fusion, mais elle ne peut pas prendre en compte l'entrelacement particulier de l'exécution des services défini à partir de l'ordre donné dans l'exemple. Cette approche ne peut donc pas être utilisée pour implanter l'exemple.

2.4.4 Les approches orientées conteneur

2.4.4.1 Exemple en JBoss-AOP

L'exemple s'implante de la manière suivante : l'intercepteur *StorageInterceptor* (algorithme 54) implante le gestionnaire de stockage qui définit le service de persistance et les intercepteurs *SecurityInterceptor* (algorithme 55) et *EncryptionInterceptor* (algorithme 56) implantent respectivement les gestionnaires de sécurité et de cryptage qui définissent le service de sécurité.

L'ordonnancement des intercepteurs s'effectue au sein d'un descripteur qui définit une pile (algorithme 6, lignes 4, 5, 6, 7 et 8) qui regroupe l'ensemble des intercepteurs qui sont activés sur le même *pointcut* (algorithme 6, ligne 3).

Algorithm 6 Déclaration des intercepteurs

```
 1  <?xml version="1.0" encoding="UTF-8">
 2  <aop>
...
 3  <pointcut name="businessmethods" expr="execution(public * *->*(..))"/>
 4  <stack name="interceptors">
 5        <interceptor class="StorageInterceptor" />
 6        <interceptor class="SecurityInterceptor" />
 7        <interceptor class="EncryptionInterceptor" />
 8  </stack>
 9  <bind pointcut="businessmethods">
10    <stack-ref name="interceptors"/>
11  </bind>
12  </aop>
```

Les limites sont les mêmes que celles indiquées pour AspectJ.

Dans l'exemple, l'entrelacement s'effectue au niveau des opérations effectuées par les gestionnaires. La liste des intercepteurs ne peut donc pas être utilisée sur cet exemple. Il n'est donc pas possible d'utiliser cette approche pour implanter l'exemple.

2.5 Evaluation

Cette partie propose une évaluation des différentes approches présentées dans ce chapitre selon un ceratin nombre de critères. La première section formalise ces critères. La deuxième section applique ces critères à ces approches afin de synthétiser la comparaison.

2.5.1 Définition des critères d'évaluation

Afin d'évaluer les différentes approches, il convient de déterminer les critères communs d'évaluation. La définition de ces critères de comparaison exige l'étude de différentes dimensions dans lesquelles interviennent les services techniques et doit répondre aux questions suivantes :

1. Comment et quand les services techniques sont-ils définis ?
   - Cette question est relative aux modèles de conception de services. Le premier critère se focalisera sur les modèles utilisés pour spécifier et implanter les services techniques. Il sera évalué dans la partie 2.5.2.
2. Comment et quand les services techniques sont-ils composés entre eux ?
   - Cette question est relative aux mécanismes de composition des services. Le deuxième critère sera axé sur les mécanismes mis en oeuvre dans les différentes approches. Cela permettra de quantifier le degré de composabilité offert par les différentes approches. Il sera évalué dans la partie 2.5.3.
3. Comment et quand les services techniques sont-ils intégrés au sein du code fonctionnel ?
   - Cette question est relative aux techniques d'intégration. Le troisième critère sera axé sur les techniques qui permettent de définir la liaison entre l'application et les services techniques. Il sera évalué dans la partie 2.5.4.

2.5.2 Evaluation selon les modèles de conception des services

Le tableau 2.1 fournit une comparaison basée sur le critère de conception des services.

Approches	Modèles de Conception
Réflexivité	Méta-objets
Aspect	Aspects
Patrons de conception	Modèle objet
Interactions	Modèle objet
Conteneur	-
Composant	Composant

TAB. 2.1 – Evaluation selon les modèles de conception

Le tableau 2.1 nous montre que chaque approche dispose de son modèle de conception sauf pour les approches orientées conteneur. En pratique, les implantations de ces approches orientées conteneurs s'appuient sur un modèle objet. Le choix d'un modèle de conception indique les capacités de composition offertes par l'approche choisie.

2.5.3 Evaluation selon le degré de composabilité

L'objectif est de concevoir des infrastructures adaptables. Le critère de comparaison sera le degré de composabilité de l'infrastructure (dans le cadre de la composition des services) fourni par les mécanismes de composition. Ce degré de composabilité est défini par la matrice de composition 2.2.

2.5.3.1 Définitions

La notion de composabilité La définition du verbe *adapter* est "rendre apte à assurer ses fonctions dans des conditions particulières ou nouvelles" [6]. La définition de *l'adaptabilité* est "la qualité de ce qui peut être adapté, de ce qui peut s'adapter" [6].

La conception d'infrastructures adaptables (présentées dans la partie 2.1.1) est relative à la notion d'adaptabilité. Cette notion a été étudiée lors du projet RNTL Arcad [10]. La capacité à supporter l'adaptabilité repose pour partie sur les notions de *modularité* et de *composabilité* [10]. La modularité est une mesure de la séparation du système en ses éléments contituants et la composabilité est une mesure des capacités d'assemblage entre les éléments constituants du système [10].

Les approches étudiées dans la partie 2.3 sont des approches qui ont été créées pour prendre en compte ces deux notions. L'approche de composition comportementale est évaluée selon le degré de composabilité qu'elle permet d'offrir aux infrastructures conçues à partir des mécanismes offerts par le canevas Brenda. Ce degré est évalué selon :

1. la dynamicité de la composition
   - La composition statique se réfère à la capacité du système à être composé lors de sa conception avant d'être déployé et exécuté. La composition dynamique se réfère à la capacité du système à être composé lors de son exécution.
   - Par exemple, la configuration du gestionnaire de persistance peut être changée dynamiquement en choisissant un autre service de décompression qui implante un algorithme de décompression plus rapide.
2. le type de composition
   - la composition structurelle se réfère à la capacité du système à définir les interactions de type structurel. La composition comportementale se réfère à la capacité du système à définir les interactions de type comportemental.

Matrice de composition Les approches seront étudiées selon la matrice 2.2. Les mécanismes de composition doivent être abordés selon le degré de composabilité qu'ils procurent au système qui doit être composé. La composition structurelle peut être soit statique, soit dynamique (idem pour la composition comportementale).

1. La composition structurelle statique correspond à la capacité du système à définir les interactions de type structurel lors de la conception du système.
2. La composition structurelle dynamique correspond à la capacité du système à définir les interactions de type structurel lors de l'exécution du système.
3. La composition comportementale statique correspond à la capacité du système à définir les interactions de type comportemental lors de la conception du système. Lors de l'exécution du système, il est impossible de modifier les interactions définies lors de la conception.
4. La composition comportementale dynamique correspond à la capacité du système à définir les interactions de type comportemental lors de l'exécution du système.

TAB. 2.2 – Matrice de composition

Composition	statique	dynamique
structurelle	1	2
comportementale	3	4

La matrice de composition définit le degré de composabilité de l'infrastructure conçue à partir des mécanismes de composition des approches détaillées dans ce chapitre. Ce degré de composabilité ne se réfère qu'à la capacité de composition de l'infrastructure.

Les tableaux 2.3 et 2.4 présentent pour chaque approche la manière dont la dépendance structurelle entre les services est définie. Les tableaux 2.5 et 2.6 présentent pour chaque approche la manière dont la dépendance comportementale entre les services est définie.

2.5.3.2 La composition structurelle

Les tableau 2.3 et 2.4 présentent pour chaque approche les notions d'interaction de type structurel. Une interaction est étudiée selon sa nature , c'est-à-dire sa manière de se caractériser, son modèle d'expression (tableau 2.3), la manière dont elle est implantée et sa dynamicité, qui correspond à la capacité de l'approche de définir et appliquer les interactions statiquement ou dynamiquement (tableau 2.4).

Approches / Interaction	Nature	Expression
Réflexivité	-	-
Aspect	-	-
Patron de conception	-	-
Interactions	-	-
Conteneur	-	-
Composant (Fractal)	Appel de méthode sur interface	Définition d'une liaison entre deux interfaces des composants

TAB. 2.3 – Nature et expression des interactions de type structurel

Approche	Implantation	Dynamicité
Réflexivité	-	-
Aspect	-	-
Patron de conception	-	
Interactions	-	-
Conteneur	-	-
Composant (Fractal)	Création de la liaison avec un *BindingController*	Statique et dynamique

TAB. 2.4 – Implantation et dynamicité des interactions de type structurel

Selon ces tableaux, seule l'approche basée sur les composants Fractal permet de définir les interactions de type structurel. L'implantation de ces interactions est réalisée dans Julia [42, 8] et permet de créer statiquement et dynamiquement ces interactions en définissant des liaisons entre les interfaces des composants.

Les autres approches ne définissent aucun concept d'interaction de type structurel et donc ces approches ne permettent pas d'exprimer les dépendances structurelles entre les services. Les services sont donc conçus indépendamment les uns des autres.

2.5.3.3 La composition comportementale

Les tableaux 2.5 et 2.6 présentent pour chaque approche les notions d'interaction de type comportemental. Une interaction est étudiée selon les mêmes critères que pour la composition structurelle.

Approches	Nature	Expression
Réflexivité	-	-
Aspects AspectJ JAC PROSE EAOP	Ordonnancement des aspects	Dépend de l'implantation Mot clef *dominate* du langage Liste ordonnée d'aspects Priorité d'exécution des aspects Opérateurs pour propagation d'événements
Patrons de conception	-	-
Interactions	Fusion des interactions basée sur des règles de fusion	-
Conteneur	-	-
Composants (Fractal)	-	-

TAB. 2.5 – Nature et expression des interactions de type comportemental

Approches	Implantation	Dynamicité
Réflexivité	-	-
Aspects		
AspectJ	Weaver	statique
JAC	Weaver	dynamique
PROSE	AspectManager	dynamique
EAOP	Arbre d'aspects	dynamique
Patrons de conception	-	-
Interactions	Génération de code issu de la fusion	dynamique
Conteneur	-	-
Composants (Fractal)	-	-

TAB. 2.6 – Implantation et dynamicité des interactions de type comportemental

Selon ces tableaux, seuls les approches orientées aspects et interactions logicielles prennent en compte les interactions de type comportemental, qui conduisent à un entrelacement des exécutions des services. La différence entre les approches orientées aspects et l'approche interactions logicielles concerne le déterminisme et la granularité de l'ordonnancement.

Le déterminisme Les dépendances comportementales sont décomposées en deux groupes :

1. le comportement déterminé à partir de la dépendance est généré et donc produira une séquence d'exécution unique. Cela vient du fait que l'expressivité des dépendances qui engendrent ce comportement est limitée et ne prend pas en compte les séquences d'exécution de l'ensemble des services. Ce comportement sera déterministe. Les approches orientées aspects sont dans ce groupe :
   (a) AspectJ génère du comportement qui donne une séquence d'exécution unique (le code des aspects est inséré dans le code applicatif et donc la séquence d'exécution est unique). L'insertion du code de trois aspects A1, A2 et A3 définit un ordonnancement unique et statique.
   (b) JAC effectue l'ordonnancement des aspects à partir d'une liste ordonnée. La séquence d'exécution issue de la composition des aspects sera également unique.
   (c) PROSE s'appuie sur un *AspectManager* qui permet d'ordonnancer les aspects dynamiquement. L'ordonnacement est défini à partir de priorités. Pour un même joinpoint, la séquence d'exécution issue de l'ordonnancement des aspects sera unique.
   (d) EAOP se base sur un arbre d'aspects qui est construit à partir d'opérateurs qui définissent la propagation des événements qui déclenchent les aspects. Cet arbre peut être modifié dynamiquement (modification des opérateurs et des aspects). Pour un même message déclencheur, la séquence d'exécution issue de l'ordonnancement des aspects sera unique (l'ordonnancement est défini à partir des opérateurs, la propagation des événements définit l'ordre d'exécution des aspects).
2. le comportement déterminé à partir de la dépendance est interprété et donc produira de multiples séquences d'exécution. Cela vient du fait que l'expressivité des dépendances qui engendrent ce comportement prend en compte les séquences d'exécution de l'ensemble des services. Ce comportement est non déterministe. L'approche interaction logicielle est dans ce groupe :

(a) L'implantation de cette approche permet d'instantier de nouvelles interactions dynamiquement. Pour un même message déclencheur, les schémas des interactions qui sont activées par ce message sont fusionnés. Le nouveau schéma est instantié. Le code généré à partir de la fusion comportementale produit une exécution concurrentielle des comportements des interactions qui est non déterministe.

Le grain d'ordonnancement Le grain d'ordonnancement dans les approches orientées aspects est l'aspect. Le grain d'ordonnancement dans l'approche interaction logicielle varie selon les règles de fusion qui seront appliquées sur les schémas.

2.5.4 Evaluation selon les techniques d'intégration

Le tableau 2.7 analyse les techniques d'intégration mises en oeuvre dans les différentes approches. Ces techniques sont utilisées pour définir le lien entre la partie applicative et la partie technique (services techniques). Le tableau indique également le moment où ces techniques sont utilisées, c'est-à-dire le moment où le lien entre la partie fonctionnelle et la partie non fonctionnelle est créé.

Approches	Techniques d'intégration	Moment de l'intégration
Réflexivité Réification statique Réification dynamique	 Extension du compilateur (compilateur ouvert) Extension de la machine virtuelle	 Compilation du code applicatif Exécution du code applicatif
Aspects AspectJ JAC PROSE EAOP	Weaving Compilation avec tisseur Modification du bytecode (wrapping) Modification de l'interprétation avec machine virtuelle dédiée Insertion d'appels de méthode pour génération d'événements	Dépend de l'implantation Compilation Chargement Exécution Compilation
Patrons de conception	-	-
Interactions	Service d'interactions	Activation de l'interaction
Conteneur JOnAS JBoss OpenCCM	Dépend de l'implantation Génération du code de l'objet d'interposition Instantiation des classes du conteneur Instantiation du conteneur	Déploiement
Composants	voir approches orientées conteneur	-

TAB. 2.7 – Evaluation des techniques d'intégration

La réflexivité La réification statique permet par extension du compilateur de modifier le code (source Java ou bytecode pour AspectJ et JAC). Les techniques d'intégration repose sur la capacité du compilateur à être étendu. Le code des méta-objets sont donc

insérés dans le code applicatif lors de la compilation de ce dernier. La réification dynamique permet par extension de la machine virtuelle de modifier l'interprétation des instructions (bytecode). Les techniques d'intégration reposent sur la capacité de la machine virtuelle à être étendue. La machine virtuelle exécute le code des méta-objets lors de l'exécution du code applicatif.

Les approches orientées aspects Les techniques d'intégration sont liées au processus de *weaving*. Elles dépendent de l'implantation de cette approche :

- Dans AspectJ, le weaver interprête les aspects (écrits dans un langage qui définit les pointcut, les advices, etc) et intègre le code des advices dans le code applicatif. Ce processus s'effectue lors de la compilation du code applicatif.
- Dans JAC, un mécanisme de wrapping est utilisé pour intégrer le code des aspects lors du chargement du code applicatif.
- Dans PROSE, le code des aspects est exécuté lors de l'exécution du code applicatif. L'intégration se base sur les mécanismes introduits dans la réification dynamique (machine virtuelle dédiée).
- Dans EAOP, l'intégration se base sur les mécanismes introduits dans la réification statique : elle insère des appels de méthode au sein du code applicatif. L'implantation de ces méthodes correspond à la génération d'événements qui vont se propager dans l'arbre des aspects.

L'approche interactions logicielles Cette approche repose sur un service d'interactions qui gère l'ensemble des interactions actives. Les techniques d'intégration reposent sur un mécanisme de *wrapper* [22] qui permet d'intégrer les appels au service d'interactions.

Les approches orientées conteneur Les techniques d'intégration dépendent de l'implantation de cette approche. Pour des raisons de performances d'exécution, l'implantation JOnAS s'est orientée vers la génération du code des conteneurs à partir des informations contenues dans le fichier de description de déploiement. L'implantation JBoss s'est orientée vers une approche architecturale : le conteneur est donc créé par instantiation des classes qui définissent l'architecture du conteneur (et qui implantent les services techniques). Pour ces deux implantations, l'intégration s'effectue lors du déploiement des composants.

Les approches orientées composant Les modèles de composant introduits par les composants EJB ou CCM délèguent la création du lien entre la partie fonctionnelle et la partie non fonctionnelle au conteneur.

2.5.5 Synthèse

Cette partie s'est intéressée à évaluer les différentes approches selon différents critères. En conclusion, chaque approche dispose de son propre modèle de conception pour implanter les services. Les approches ont différents degrés de composabilité qui caractérisent les mécanismes de composition mettent en oeuvre avec plus ou moins d'expressivité les interactions entre les services. Les techniques d'intégration sont multiples et n'influent pas sur les mécanismes de composition des approches.

Selon cette évaluation, les mécanismes de composition structurelle sont clairement définis dans les approches orientées composants. Les mécanismes de composition comportementale sont très différents selon les approches et se basent tous sur la notion d'ordonnancement.

2.6 Techniques d'implantation des approches

Chaque approche dispose de propriétés particulières qui permettent d'avoir un degré de composabilité plus ou moins important. Des recherches qui visent à utiliser les propriétés intéressantes de chaque approche pour les appliquer aux autres ont été effectuées. Le tableau 2.8 indique quelques-uns des ces travaux : les lignes du tableau recencent les approches qui sont utilisées pour concevoir les approches données dans les colonnes du tableau. Ce tableau n'est pas exhaustif : il vise à montrer comment certains travaux ont été intégrés aux différentes approches pour bénéficier de leurs avantages. L'objectif de ces travaux est de développer une approche en utilisant les propriétés clairement définies dans les autres, ceci dans un but d'accroitre l'adaptabilité des infrastructures.

Approches	Réflexivité	Aspects	Patrons de conception	Interactions	Composant	Conteneur
Réflexivité	X	(1)	-	(4)	(5)	(8)
Aspects	-	X	(3)	-	(6)	(9)
Patrons de conception	-	(2)	X	-	-	-
Interactions	-	-	-	X	-	(10)
Composant	-	-	-	-	X	(11)
Conteneur	-	-	-	-	(7)	X

TAB. 2.8 – Techniques d'implantation des approches

2.6.1 Les aspects

Pour implanter les approches orientées aspects, les travaux utilisés sont :

- La réflexivité (1)
  La réflexivité est utilisée dans les approches orientées aspects pour réaliser les implantations qui mettent en oeuvre les concepts d'AOP. Dans AspectJ, l'architecture du *weaver* est inspirée de celle d'un compilateur ouvert tel que défini dans la réification statique. Les aspects correspondent aux extensions de ce compilateur. Dans PROSE, l'architecture de la machine virtuelle est directement dérivée des travaux de la réification dynamique. Les implantations de l'approche orientée aspects utilisent les mécanismes d'introspection et d'intercession introduits dans la réflexivité. La conception de ces implantations bénéficient directement des apports de la réification statique (AspectJ) et dynamique (dans PROSE, il est possible d'ajouter de nouveaux aspects à l'exécution avec les mécanismes de réification dynamique).
- Les patrons de conception (2)
  [53] propose une architecture basée sur un modérateur (*Moderator*). Ce modérateur permet de coordonner les interactions entre les composants et les aspects. Il montre à partir d'un patron de conception, comment il étend le concept de weaving.

2.6.2 Les patrons de conception

Des travaux cherchent à utiliser les approches orientées aspects pour implanter certains patrons de conception (3) [99]. [90] définit le patron de conception *Visitor* [63] comme un aspect qu'il applique à l'application.

2.6.3 Les interactions logicielles

L'implantation de l'approche "interactions logicielles" s'appuie sur la réflexivité (4).

Les mécanismes de la réflexivité sont utilisés pour déclencher l'exécution des interactions [22]. Le mécanisme mis en oeuvre dans l'implantation est la réification des appels de méthode dans le code applicatif. Le code généré par cette réification est en plus constitué d'appels pour déclencher les interactions.

2.6.4 Les composants

Pour implanter les composants, les travaux utilisés sont :

- La réflexivité (5)
  L'implantation Julia du modèle Fractal utilise des mécanismes issus des approches réflexives pour implanter les contrôleurs (comme par exemple les contrôleurs *AttributeController* et *LifeCycleController*). Ainsi, l'implantation Julia offre des mécanismes de composition structurel à la fois statique et dynamique.
- Les aspects (6)
  [117, 116] intègre au sein du modèle Fractal les mécanimes définis dans JAC qui implante l'approche orientée aspect. L'objectif est de définir les propriétés non fonctionnelles sous forme de composants et de les intégrer au sein des composants Fractal avec une approche orientée aspect.
- Les conteneurs (7)
  Les composants EJB [56] et CCM [103, 127] sont basés sur une architecture de type conteneur qui réalise le lien entre la partie fonctionnelle (code applicatif) et la partie non fonctionnelle (partie services techniques). Ainsi les implantations des composants se focalisent uniquement sur la partie fonctionnelle.

2.6.5 Les conteneurs

Pour implanter les conteneurs, les travaux utilisés sont :

- La réflexivité (8)
  Ce rapprochement est étroitement lié à l'intégration de l'approche orientée aspects au sein des conteneurs.
- Les aspects (9)
  JBoss-Aop [44] définit les services techniques définis dans les conteneurs EJB sous forme d'aspect afin de concevoir le conteneur à partir d'une approche orientée aspect. L'objectif est d'accroître la modularité du conteneur.
- Les interactions (10)
  [20] détaille l'intégration d'un service d'interactions au sein d'un conteneur EJB pour permettre au conteneur d'intégrer dynamiquement de nouveaux services. Les mécanismes de composition de ces services s'appuient sur les mécanismes de fusion des interactions.
- Les composants (11)
  Une piste de travail effectué au sein du laboratoire DTL/ASR est de concevoir les services techniques définis JOnAS sous forme de composants Fractal. L'objectif de ce travail est d'implanter les conteneurs par des composants techniques. Le modèle

Fractal donne un degré d'adaptabilité à l'architecture du conteneur définie dans JOnAS. De plus, le modèle Fractal offre de nouveaux mécanismes de composition de services au conteneur ([41, 42]).

2.6.6 Conclusion

Le tableau 2.8 montre que le champ d'investigation est encore très vaste et que les possibilités d'implanter les approches de différentes manières sont encore importantes. Il tend également à montrer que les approches sont utilisées pour implanter de nouvelles approches pour améliorer l'adaptabilité des infrastructures. L'exemple qui illustre le mieux cette constatation concerne les conteneurs, qui sont largement déployés dans le monde industriel : les travaux identifiés cherchent à améliorer leur adaptabilité pour que les composants utilisent au mieux les services offerts par la plate-forme d'exécution.

2.7 Conclusion

Ce chapitre a présenté les concepts de la composition et les a déclinés aux différentes approches de conception actuellement utilisées pour implanter les infrastructures logicielles à base de services techniques. Les approches étudiées sont :

1. la réflexivité, où l'approche ne définit que les mécanismes de réification. Il n'existe pas de mécanismes de composition de méta-objets,
2. les approches orientées aspects, où la composition d'aspects se résume à un ordonnancement de l'exécution de ces aspects, ordre exprimé à l'aide de structures simples, comme une liste, ou de structures plus complexes comme un arbre,
3. les patrons de conception, où la composition des services est réalisée avec quelques patrons, mais ces derniers ne fournissent aucun mécanismes de composition pour créer les interactions entre les services,
4. les interactions logicielles, où la composition correspond à une fusion d'interactions qui s'appuie sur la sémantique du langage pour entrelacer l'exécution des interactions, ce qui limite les capacités de l'entrelacement,
5. les composants, où la composition consiste à définir une configuration de composants à partir d'un assemblage,
6. les conteneurs, qui offrent des mécanismes pour que les composants métier puisse utiliser les services techniques offerts par l'infrastructure, mais dont la conception est réalisée sans qu'aucun mécanisme de composition ne soit utilisé pour composer les services techniques.

Une évaluation a été réalisée sur la base de critères définis à partir de questions élémentaires sur la composition de services. En résumé, il apparaît que

1. chaque approche définit un modèle spécifique pour implanter un service,
2. chaque approche a un degré de composabilité qui est caractérisé par les mécanismes de composition qu'elle propose,
3. l'intégration des services au sein de l'infrastructure diffère selon les approches.

L'étude menée dans ce chapitre a également montré qu'il n'existe pas d'approche prédominante par rapport aux autres. La partie 2.6 illustre également le fait que la tendance actuelle est d'implanter une approche avec les mécanismes identifiés dans les autres afin d'offrir à cette dernière des mécanismes de conception évolués pour la construction d'infrastructures.

Le chapitre 3 va présenter une nouvelle approche de composition dont l'objectif est de :

1. définir les mécanismes de composition qui implantent les concepts introduits dans la partie 2.2,
2. repousser les limites des mécanismes de composition des différentes approches afin de rendre les infrastructures plus adaptables.

Cette nouvelle approche doit prendre en compte le fait de pouvoir être utilisée dans les autres approches pour bénéficier de ses capacités de composition.

Chapitre 3

Une nouvelle approche de composition

3.1 Introduction

Le chapitre 2 a présenté les limites actuelles des différentes approche de conception dans le domaine de la composition. Elles implantent chacune leurs propres mécanismes de composition. Ces mécanismes sont implantés de manière partielle : chaque approche ne définit que quelques concepts de composition présentés dans la partie 2.2.

L'objectif est, à partir de l'identification des limites des approches, de proposer une nouvelle approche qui prend en compte l'ensemble des capacités offertes par ces dernières et qui repousse leurs limites. Cette nouvelle approche doit définir un cadre architectural qui met en oeuvre ces mécanismes de composition. La nouvelle approche offre un modèle qui permet d'exprimer un entrelacement entre opérations à partir de relations d'ordre qui ne se basent pas sur la sémantique d'un langage.

Pour présenter cette approche, ce chapitre est constitué de six parties :

1. la première partie décrit et formalise notre approche en définissant les concepts qui seront détaillés dans les parties suivantes,
2. la deuxième et troisième partie présentent le modèle de conception qui est utilisé pour implanter les services,
3. la quatrième partie s'intéresse plus particulièrement à la modélisation des interactions de type comportemental,
4. la cinquième partie détaille les mécanismes de composition mis en oeuvre avec les différents modèles présentés dans les parties précédentes,
5. la dernière partie implante l'exemple donné dans la partie 1.3.1.

Pour décrire cette nouvelle approche, il nous faut auparavant rappeler les limites des autres approches.

3.1.1 Comment étendre les capacités des approches de conception ?

Les approches réflexives ne définissent pas de mécanisme de composition. Seul le prototype Guarana [100, 101] définit la notion de méta-configuration qui correspond à l'ensemble des méta-objets. La notion de configuration se retrouve dans les concepts de composition. La nouvelle approche de composition doit donc définir une configuration de services.

Dans les approches orientées aspects, les aspects sont indépendants les uns des autres. Les limites de ces approches sont que les relations d'ordre entre les aspects ont un modèle d'expression limité (liste, arbre). De plus, le grain d'ordonnancement est l'aspect, alors

que, comme l'a montré l'exemple, il est plus approprié de pouvoir définir un ordonnancement sur les opérations des aspects afin d'avoir un entrelacement au niveau des opérations et non pas au niveau des aspects. L'ordonnancement des aspects n'intervient que lorsque ces derniers sont non orthogonaux.

Dans l'approche interactions logicielles, la fusion des interactions s'appuie sur la sémantique du langage ISL, ce qui limite les possibilités d'entrelacer le code des interactions. La nouvelle approche doit donc offrir un modèle qui permet d'exprimer un entrelacement entre opérations à partir de relations d'ordre qui ne se basent pas sur la sémantique d'un langage.

Les approches orientées composants sont utilisées pour concevoir des assemblages de composants. Dans une configuration de composants, un composant peut utiliser les fonctionnalités d'un autre selon les contrats qui existent entre eux. Les limites de cette approche correspond au fait qu'il n'existe dans ces approches que des contrats sur les interactions de type structurel. La nouvelle approche doit donc prendre en compte l'ajout de nouveaux services dont les interactions ne sont pas uniquement de type structurel.

Dans les approches orientées conteneurs, la principale limite concerne l'adaptabilité. L'architecture logicielle des conteneurs ne prend en compte que très partiellement la configuration de services. La nouvelle approche doit donc être capable d'être implantée dans un conteneur pour qu'ils puissent bénéficier des concepts de la composition (voir les techniques d'implantation de la partie 2.6).

3.1.2 Introduction de la nouvelle approche

La nouvelle approche doit définir un certain nombre de modèles pour résoudre les limites exposées dans la partie précédente. Ils doivent permettre d'exprimer les interactions de type structurel et comportemental.

Il existe différents cas de composition de services :

1. le premier concerne la composition des services qui, pour offrir une fonctionnalité, utilise les fonctionnalités d'autres services. Cette composition s'appuie sur la définition d'interactions de type structurel. Il est pris en compte par les approches orientées composants.
2. le deuxième concerne la composition des services qui fonctionnent indépendamment des autres, mais dont leur exécution influe sur celle des autres : cette composition s'appuie sur la définition d'interactions de type comportemental. Il est pris en compte par les approches orientées aspects et interactions logicielles. La limite vient de la résolution de la non orthogonalité. La non orthogonalité est gérée par la définition de relations d'ordre entre aspects dans les approches orientées aspects. Elle est gérée par la fusion des interactions dans l'approche interactions logicielles. Or, l'entrelacement de l'exécution des services nécessite un grain plus fin que celui de ces approches.

Chaque approche ne considère que l'un des deux cas de composition : les approches orientées composants ne permettent pas de définir d'interactions de type comportemental. Les approches orientées aspects et interaction logicielle ne définissent pas d'interactions de type structurel. Contrairement à ces dernières, notre approche doit prendre en compte ces différents cas de composition.

Afin de mettre en oeuvre les interactions de type structurel, notre approche définit un modèle de composants qui permettra de concevoir les services. Ce modèle sera détaillé dans la partie 3.2.

Afin de mettre en oeuvre les interactions de type comportemental, il est nécessaire d'avoir un moyen d'expression de l'ordonnancement pour déterminer l'entrelacement de

l'exécution des services. Cette expression permet de définir les relations d'ordre . Le modèle d'expression sera détaillé dans la partie 3.4. Les relations d'ordre doivent pouvoir s'exprimer à partir de l'exécution des services. Décrire l'exécution d'un service consiste à modéliser son comportement. Un modèle de comportement doit être défini. Ce modèle sera détaillé dans la partie 3.3. Le modèle de comportement doit intégrer la notion de parallélisme d'exécution pour définir le fonctionnement indépendant des services.

Afin d'avoir un modèle de conception homogène, les relations d'ordre seront définies sous la forme de contraintes d'ordonnancement. Une contrainte sera modélisée sous la forme d'un composant. La sémantique de la relation d'ordre sera définie sous la forme d'un automate.

Les mécanismes de composition décrits dans la partie 3.5 permettent de créer ces interactions.

Notre approche doit pouvoir être implantée dans les autres approches, notamment celle des conteneurs (partie 2.3.5) afin qu'ils puissent obtenir un haut degré de composabilité. Notre approche permet de concevoir une configuration de composants qui est le résultat de la composition des composants. Pour utiliser le résultat de la composition des services au sein des autres approches, il suffit d'utiliser la configuration des composants.

La composition des services se traduit par la composition des composants et des contraintes d'ordonnancement. La composition des composants se traduit par la composition des comportements de ces derniers. Le résultat de la composition est une configuration. Le comportement de la configuration est le résultat de la composition des comportements des composants de la configuration.

En conclusion, notre approche doit permettre de définir les interactions de type structurel et les interactions de type comportemental en construisant des configurations de composants. Par la suite, nous appellerons cette nouvelle approche *approche de composition comportementale*.

3.2 Modèle de composants

Dans la partie 2.3.6, quelques modèles de composants ont été exposés. [76, 128, 129, 40] ont travaillé sur les approches à composants. Pour introduire le modèle de comportement, il apparaît que le modèle de composants Fractal est le choix le plus approprié pour mettre en oeuvre l'approche de composition comportementale. Le modèle Fractal a été conçu pour permettre d'étendre les mécanismes de composition, et donc de définir différents types de composition [55]. La composition comportementale s'intègre idéalement dans cette conception.

Le modèle de composants Fractal se caractérise par un certain nombre de propriétés [42] qui sont conservées dans notre approche telles que :

- la dynamicité : les composants sont des unités d'exécution et peuvent donc être ajoutés, modifiés ou supprimés à l'exécution,
- l'encapsulation : les composants interagissent à travers des points d'accès clairement définis (interfaces),
- l'identité : un composant est identifié de manière non ambigüe des autres composants,
- la composition imbriquée : un composant peut contenir des sous-composants,
- le partage : un composant peut être contenu dans différents composites,
- le contrôle : les mécanismes réflexifs définis dans Fractal permettent d'ajouter et supprimer les composants et de créer ou détruire les liaisons entre leurs interfaces.

Un composant Fractal dispose d'interfaces pour interagir avec les autres composants. Il existe deux types d'interface : les interfaces serveur qui offre les fonctionnalités du composant aux autres et les interfaces client qui spécifie les fonctionnalités que le composant

a besoin pour fonctionner. Dans la projection du modèle en Java, ces interfaces correspondent à des interfaces Java. L'interaction entre composants est rendue possible grâce aux liaisons. Une liaison est un lien orienté entre une interface client et une interface serveur. L'interaction est initiée par l'interface client.

Un composant Fractal est constitué d'une membrane et d'un contenu. Le contenu d'un composant est soit un graphe d'objets s'il s'agit d'un composant primitif, soit un ensemble de sous-composants s'il s'agit d'un composant composite. La membrane d'un composant intercepte les interactions sur les interfaces du composant. Elle peut exhiber des interfaces de contrôle qui permettent de gérer le composant.

La propriété de partage ne sera pas mise en oeuvre dans l'implantation de l'approche.

3.3 Modèle de comportement

Le modèle de comportement présenté dans cette partie permet de décrire le comportement d'un composant. Après avoir présenté quelques modèles dans la partie 3.3.1, nous détaillerons un modèle adapté à notre problématique et basé sur les automates dans la partie 3.3.2. Un formalisme graphique sera donné dans la partie 3.3.3 afin de l'appliquer à l'exemple donné dans la partie 1.3.1. Les caractéristiques du modèle sont étudiées dans les parties 3.3.4 et 3.3.5. Pour finir, la partie 3.3.6 montrera comment sont intégrés les automates au modèle de composants.

3.3.1 Quelques modèles de comportement

Les automates Il existe de nombreux travaux qui utilisent les automates. Notre approche s'assimile à la mise en oeuvre de processus qui s'exécutent en parallèle et qui interagissent entre eux. [17, 18] dit que "La plupart des travaux sur la sémantique des processus dits "parallèles", ou "communicants", ou "concurrents", ou "interagissants", reposent sur la notion d'automate. [...] cette notion permet de modéliser plus ou moins finement le comportement d'un système de processus. [...] Bien que la théorie des automates finis ait été principalement développée en liaison avec la théorie des langages et ses applications à l'analyse lexicale et syntaxique, cette théorie a aussi été utilisée dans d'autres domaines, en particulier pour étudier la sémantique des programmes séquentiels itératifs [81, 124]."

Concernant la structure de l'automate, [18] indique que "De manière plus générale, un automate fini, constitué d'états et de transitions étiquetées entre ces états, permet de décrire un système dont l'état évolue au cours du temps. [...] chaque action que peut exécuter le programme ne peut l'être que s'il est dans certains états et son exécution entraîne, le cas échéant, un changement d'état ; les actions sont donc représentées par des transitions dans l'automate. Cette façon de représenter par un automate un système dont l'état évolue consécutivement à l'exécution de certaines actions ou à l'occurrence de certains événements s'applique à beaucoup de situations, parmi lesquelles figurent tout naturellement les systèmes de processus."

Les réseaux de Petri [18] dit que "D'autres modèles de systèmes de processus ont été proposés parmi lesquels il faut mentionner les réseaux de Petri [38, 122, 118] et les calculs de processus à la Milner comme CCS [95, 77, 96] et Meije [35]. Sans entrer dans les détails on peut admettre que tous ces modèles reviennent en fait à spécifier les ensembles d'états et des transitions entre ces états. Ainsi les état d'un réseau de Petri sont ses marquages et les transitions entre états sont réalisées par le tir, simultané ou non selon la sémantique donnée au réseau, de transitions du réseau. Dans les algèbres de processus les états du système sont des termes et les transitions sont définies par la sémantique opérationnelle

du calcul qui indique comment et sous quelles conditions un terme du calcul se transforme en un autre terme."

3.3.2 Définition

Le paragraphe précédent 3.3.1 présentait succinctement les différents modèles qui sont utilisés pour décrire le comportement d'entités logicielles. Dans notre approche, un modèle inspiré des automates a été conçu. Le choix des automates est justifié par [17, 18].

Un automate est défini par l'ensemble {S, T, S0} où

1. S est l'ensemble des états de l'automate,
2. T est l'ensemble des transitions possibles,
3. S0 est l'état initial.

Une transition est définie par un événement, une garde et une action.

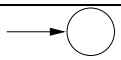
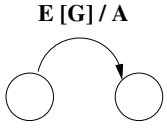
Un événement correspond à un appel d'une opération définie sur une interface serveur du composant. Une garde est définie à partir des méthodes booléennes qui sont dans l'implantation du composant. Une action correspond :

1. soit à l'exécution d'une opération appartenant à l'implantation du composant,
2. soit à l'émission d'un événement qui correspond à une demande d'exécution d'une opération définie sur une interface cliente du composant,
3. soit à l'exécution séquentielle de deux actions A1 et A2 notée A1 ; A2
4. soit à l'exécution parallèle de deux actions A1 et A2 notée A1 || A2.

Nous ne donnerons pas de définition formelle de l'automate.

3.3.3 Formalisation graphique

La formalisation graphique du modèle est illustrée dans le tableau 3.1. Chaque élément du modèle a une correspondance graphique. Ce formalisme est dérivé des *statecharts* [73, 74].

Eléments du modèle	Représentation graphique
Etat	
Etat initial (S0)	
Transition constituée d'un événement E d'une garde G d'une action A	

TAB. 3.1 – Formalisation graphique

3.3.4 Automate minimal

L'automate minimal est un automate qui n'a qu'un état *S0* (initial et final) et une seule transition effectuée à la réception de l'événement *e*. La minimalité de cet automate indique que tout composant peut être représenté par un automate.

La figure 3.1 montre cet automate.

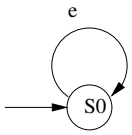


FIG. 3.1 – Automate minimal

3.3.5 Modèle hiérarchique

Le modèle est hiérarchique car un automate peut être défini à partir de plusieurs automates. Ainsi, l'état d'un automate peut encapsuler la composition de plusieurs automates. Ce modèle hiérarchique permet de décrire très finement le comportement d'un composant.

La figure 3.2 donne un exemple qui illustre ce modèle.

L'état initial de l'automate est S0. Si un événement e4 se produit, l'automate passe dans l'état S1. Si un événement e3 se produit, l'automate passe dans l'état S2. L'état S0 est décrit par l'automate composé de l'état initial S3 et de l'état S4. L'automate passe de S3 à S4 lorsque l'événement e se produit. L'automate passe de S4 à S3 lorsque l'événement e5 se produit. L'état S4 est constitué de l'état initial S5 et de l'état S6. L'automate passe de l'état S5 à S6 lorsque l'événement e2 se produit. Il accepte l'événement e3 lorsqu'il est dans l'état S6. L'état S1 est décrit par l'automate constitué de l'état initial S7 qui accepte l'événement e. L'état S2 est décrit par l'automate composé de l'état initial S8 et de l'état S9. Il accepte l'événement e lorsque l'automate est dans l'état S8 ou S9 et passe de l'état S8 à S9 lorsque l'événement e2 se produit.

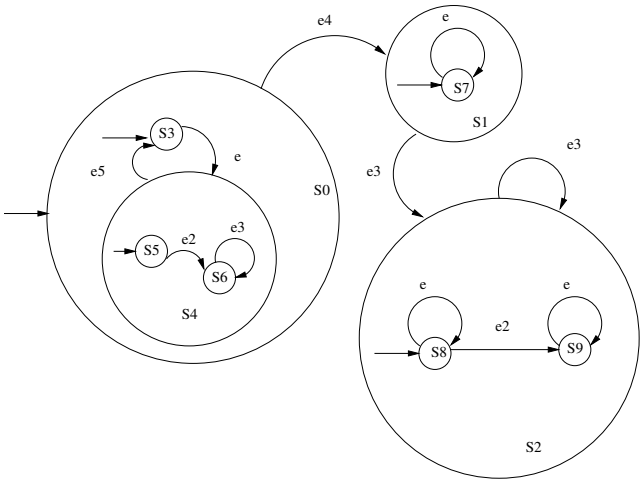


FIG. 3.2 – Modèle hiérarchique d'automates

3.3.6 Automates et composant

Le comportement d'un composant est défini par un ou plusieurs automates. Le comportement d'un composant est la composition des automates qui le décrivent. La composition des automates est le résultat du produit synchrone des automates [17, 18].

Le rôle de la membrane du composant (voir partie 3.2) est l'interception des interactions. Dans notre approche, la membrane sera dédiée à la gestion des automates. La membrane d'un composant primitif gère le lien qui existe entre l'automate du composant

et le graphe d'objets du contenu du composant (l'automate peut exécuter certaines opérations offertes par ce graphe d'objets). La membrane d'un composant composite gère l'ensemble des automates des sous-composants. Cette gestion inclue la création, la composition et l'instantiation des automates.

3.3.7 Conclusion

Cette partie a présenté un modèle de comportement à base d'automates pour décrire le comportement des composants. Ce modèle est minimal et hiérarchique. Il s'intègre dans le modèle de composants défini dans la partie 3.2. Ce modèle sera utilisé pour modéliser les relations d'ordre. La partie suivante expose le modèle d'expression des relations d'ordre.

3.4 Modélisation des relations d'ordre

Les relations d'ordre expriment les dépendances comportementales entre les actions des automates. Cette partie s'intéresse à la modélisation des contraintes d'ordonnancement qui définissent les relations d'ordre. Afin de conserver un modèle de conception homogène, les contraintes d'ordonnancement sont modélisées sous la forme de composants. Le modèle est détaillé dans la partie 3.4.1. Le comportement de la contrainte définit la sémantique de l'ordonnancement. Le modèle de comportement est défini dans la partie 3.4.2.

3.4.1 Modèle de composant

Les contraintes d'ordonnancement qui définissent la sémantique de l'ordonnancement sont, dans notre approche, représentées sous forme de composants. Le modèle d'une contrainte est le modèle Fractal. L'application d'une relation d'ordre se traduira, dans notre modèle, par l'ajout d'un composant qui représente la contrainte qui définit cette relation d'ordre dans la configuration des composants.

Les propriétés de ce modèle sont donc appliquées aux contraintes d'ordonnancement :

- la dynamicité : l'ajout ou la suppression d'une contrainte peut s'effectuer lors de l'exécution. Les relations d'ordre peuvent donc s'appliquer dynamiquement.
- l'encapsulation : les contraintes communiquent avec les autres composants par l'intermédiaire des canaux d'événements,
- la composition imbriquée : un composant peut contenir des sous-composants, que ce soit des composants qui implantent les services, ou des contraintes,
- l'identité,
- le contrôle.

La propriété de partage ne sera pas mise en oeuvre dans l'implantation de l'approche.

3.4.2 Modèle de comportement

Une dépendance comportementale entre deux composants définit une relation d'ordre dans les séquences d'exécution de ces composants. Ces séquences d'exécution sont modélisées par les automates dont le modèle est défini dans la partie 3.3.2. Les relations d'ordre sont donc définies à partir des opérations des automates qui décrivent le comportement des services techniques.

Cette partie se consacre au modèle d'expression de ces relations d'ordre. L'expressivité d'une relation d'ordre constitue un des points les plus importants de l'approche : plus le pouvoir d'expression de la relation d'ordre est important, plus l'ordonnancement sera réalisé de manière fine.

L'ordonnancement est étudié dans de nombreux domaines informatiques (parallélisation du calcul scientifique, robotique, etc) mais également organisationnels (plannification de tâches et de sous-tâches, etc). Les objectifs de l'ordonnancement diffèrent selon les domaines : dans le domaine du calcul réparti, l'objectif est de déterminer le meilleur ordonnancement pour obtenir le résultat le plus rapidement possible alors que dans le domaine de la planification de tâches, l'objectif est de déduire le chemin critique afin de le diminuer pour réduire les coûts. Dans notre approche, l'objectif est de fournir un modèle qui permette d'exprimer l'ordonnancement au niveau des opérations effectuées par les automates afin de déterminer les séquences d'exécution qui sont valides.

Pour conserver un modèle homogène, une contrainte d'ordonnancement est modélisée sous la forme d'un composant. Dans la même idée, pour conserver un modèle homogène, le comportement d'une contrainte d'ordonnancement est modélisé sous la forme d'automates. Le comportement de la contrainte décrit la sémantique de la relation d'ordre.

3.4.2.1 Eléments du modèle

Un automate d'une contrainte est défini par l'ensemble {S, T, S0} où

1. S est l'ensemble des états de l'automate,
2. T est l'ensemble des transitions possibles (une transition est définie par un événement et une garde),
3. S0 est l'état initial.

Les relations d'ordre sont exprimées à partir des opérations que peuvent effectuer les automates des composants. Ces opérations sont :

1. le changement d'état : entrée dans un nouvel état, sortie d'un état,
2. le début de l'exécution d'une action, la fin de l'exécution d'une action.

Les automates des contraintes réagissent aux opérations effectuées par les automates des composants. Un événement d'un automate d'une contrainte correspond soit à un changement d'état d'un automate, soit au début ou la fin de l'exécution d'une action d'un automate. Une garde est définie à partir des méthodes booléennes de l'implantation de la contrainte.

En ce qui concerne les actions des automates, une relation d'ordre peut s'exprimer soit à partir d'une action, soit à partir d'un ensemble d'actions dont la signature est la même.

Une contrainte peut s'appliquer sur un nombre quelconque d'automates, et sur un nombre quelconque d'actions : une contrainte peut ordonner les actions de plusieurs automates qui proviennent de composants différents.

3.4.2.2 Sémantique de l'automate

Exprimer une relation d'ordre entre opérations consiste à définir le moment où les opérations peuvent s'effectuer. Dans un modèle d'automate, ce moment se traduit en état. Ainsi, un état d'un automate d'une contrainte d'ordonnancement représente le moment où une opération d'un automate de composant peut s'exécuter. Une opération est soit un changement d'état, soit l'exécution d'une action.

La transition qui fait rentrer l'automate dans l'état qui autorise l'exécution de l'opération *op* représente la condition pour laquelle l'automate d'un composant a la possibilité d'effectuer à partir de ce moment cette opération *op*. La transition qui fait sortir l'automate d'un état qui autorise l'exécution de l'opération *op* représente la condition pour laquelle l'automate d'un composant n'a pas la possibilité d'effectuer à partir de ce moment cette opération *op*.

La formalisation graphique est la même que celle illustrée dans le tableau 3.1. Les états d'un automate d'une contrainte d'ordonnancement ne traduisent pas tous le moment où les opérations peuvent s'effectuer. Ainsi, le formalisme graphique d'un état qui ne traduit pas le moment où les opérations peuvent s'effectuer est ○ alors que le formalisme d'un état qui traduit le moment où les opérations peuvent s'effectuer est ●. La figure 3.3 illustre l'utilisation de ces deux types d'état.

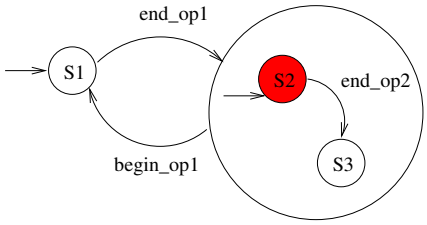


FIG. 3.3 – Exemple d'un automate de contrainte d'ordonnancement

La figure 3.3 montre un automate qui définit une relation d'ordre entre deux opérations *op1* (définie dans l'automate A1) et *op2* (définie dans l'automate A2). L'état initial est l'état S1. A la réception de l'événement qui correspond à la fin de l'exécution de l'opération *op1* (noté *end_ op1*), l'automate rentre dans l'état S2 qui correspond au moment où l'opération *op2* peut s'exécuter. L'automate passe dans l'état S3 à la réception de l'événement qui correspond à la fin de l'exécution de l'opération *op2*. A tout moment, l'automate peut revenir dans l'état S1 lors de la réception de l'événement qui correspond au début de l'exécution de l'opération *op1* (noté *begin_ op1*). Cet automate permet ainsi l'exécution de l'opération *op2* qu'après l'exécution de l'opération *op1*. La séquence d'exécution ordonnée par cet automate est donc : [[*op1*]+ *op2*]*.

3.4.2.3 Modèle hiérarchique

Le modèle hiérarchique permet de définir des contraintes complexes et de pouvoir assembler des sous-contraintes. La figure 3.3 donne une représentation qui illustre le fait que le modèle hiérarchique augmente le pouvoir d'expression des relations d'ordre.

3.4.3 Modèle de concurrence

D'après [51], "la concurrence s'occupe des aspects fondamentaux des systèmes d'agents multiples simultanément actifs qui interagissent entre eux." [51] présente également un état de l'art des travaux du domaine de la concurrence.

Les différents automates définis dans le modèle de comportement s'exécutent de manière concurrente. Ils interagissent entre eux par échange d'événements. Les relations d'ordre définissent le moment où les opérations des automates peuvent s'effectuer. Ce moment où une opération peut s'effectuer se traduit par l'activation d'une barrière de synchronisation. Une barrière de synchronisation permet de contrôler l'instant où ue opération peut être exécutée. Une barrière se matérialise dans notre approche par des points de synchronisation qui sont introduits au sein des automates des composants lors de l'application des contraintes. Ces points de synchronisation sont gérés par les automates des contraintes d'ordonnancement.

Lorsque l'automate d'une contrainte d'ordonnancement entre dans un état qui représente le moment où une opération peut être exécutée, alors l'automate émet un événement à destination de l'automate dont est issue l'opération. Cet événement correspond au point de synchronisation.

L'expression de plusieurs relations d'ordre sur une même opération se traduit par la mise en parallèle de ces points de synchronisation.

Ces points de synchronisation permettent de déterminer l'ordre d'exécution des actions effectuées par les automates, ce qui correspond à l'entrelacement des séquences d'exécution des composants. Cet entrelacement est déterminé par l'ordonnanceur qui est généré en fonction des relations d'ordre appliquées lors de l'ajout des contraintes d'ordonnancement dans la configuration des composants.

Modèle d'exécution Le modèle d'exécution qui implante ce modèle de concurrence est relatif à la mise en oeuvre de notre approche (chapitre 5). Aussi le modèle d'exécution est détaillé dans la partie 5.5.1.1. Cette partie présente comment les modèles d'automates définis dans ce chapitre sont traduits dans le modèle d'exécution.

3.5 Les mécanismes de composition

Cette partie présente les mécanismes de composition qui mettent en oeuvre les interactions de type structurel (section 3.5.1) et comportemental (section 3.5.2). Ces mécanismes permettent de créer une configuration de composants qui est le résultat de la composition des services.

3.5.1 Le mécanisme de composition structurelle

Cette partie détaille le mécanisme de composition structurelle appliqué au modèle de composant et au modèle de comportement.

La définition d'une interaction de type structurel se traduit par la création d'une liaison entre deux interfaces de composants (interface serveur et interface cliente). Une liaison entre les composants implique que les automates des composants vont communiquer entre eux. Cette communication se traduit par des échanges d'événements entre les automates des composants. La création de cette liaison se traduit par la création d'un canal d'événements qui permet l'échange d'évenements entre les automates des composants.

Exemple La figure 3.4 illustre la création d'une liaison entre les interfaces *i1* du composant *C1* et *i2* du composant *C2*. Cette création de liaison se traduit par la création d'un canal d'événements où s'échangent les événements qui correspondent aux opérations définies sur les interfaces. L'opération *op1* se traduit en événements *op1* (qui correspond à l'exécution des actions relatives à l'opération *op1* et qui est émis par l'automate A1 du composant C1 vers l'automate A2 du composant C2) et *end_ op1* (qui indique la fin d'exécution des actions suite à la réception de l'événement *op1* et qui est émis par l'automate A2 vers l'automate A1). Le raisonnement est le même pour l'opération *op2*.

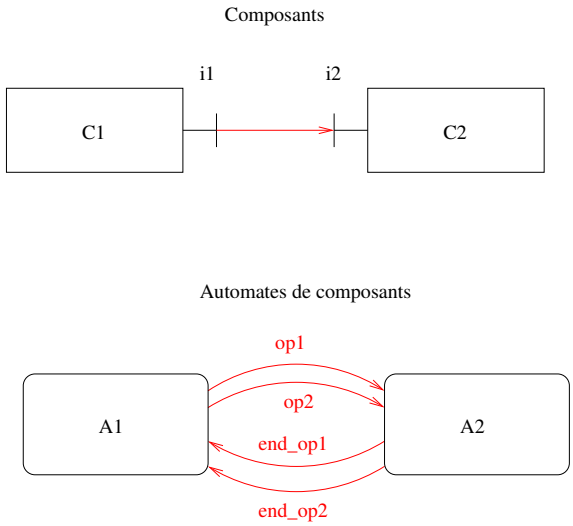


FIG. 3.4 – Création d'une liaison et d'un canal d'événements

3.5.2 Le mécanisme de composition comportementale

Cette partie détaille le mécanisme de composition comportementale appliqué au modèle de composant et au modèle de comportement. L'application d'une relation d'ordre implique une communication entre automates qui se traduit par un échange d'événements qui permettent de synchroniser l'exécution des automates. La définition d'une interaction de type comportemental se traduit par la création de canaux d'événements entre l'automate de la contrainte d'ordonnancement et les automates des composants qui interviennent dans la relation d'ordre.

Exemple La figure 3.5 illustre la création d'un canal d'événements entre les automates *A1* et *A2* et l'automate de la contrainte d'ordonnancement dont l'automate est donné dans la figure 3.3. L'automate *A1* émet les événements *begin_op1* et *end_op1* à destination de l'automate de la contrainte. L'automate *A2* émet l'événement *end_op2* à destination de l'automate de la contrainte. L'automate de la contrainte émet l'événement *op2_synchronization_point* à destination de l'automate *A2* qui correspond au point de synchronisation utilisé pour ordonnancer l'exécution de l'opération *op2*. Ces événements sont générés lors de l'ajout de la contrainte dans la configuration de composants.

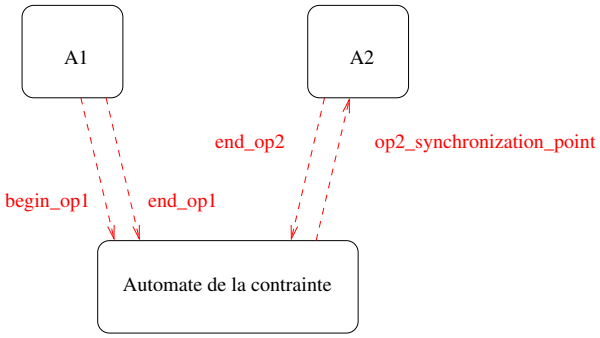


FIG. 3.5 – Création de canaux d'événements

La définition de points de synchronisation modifie les automates des composants dont l'exécution doit être ordonnancée. Ainsi, l'insertion d'un point de synchronisation correspondant à un ordonnancement d'une action *act* exécutée par un automate se traduit

par la création d'une action séquentielle *act2* composée de l'action correspondant à la synchronisation et de l'action *act* puis du remplacement de l'action *act* par l'action *act2* au sein de l'automate.

Lorsque plusieurs points de synchronisation sont définis sur une même action, alors les points de synchronisation se traduisent par une action parallèle d'actions de synchronisation.

3.6 Exemple

Cette partie décline entièrement l'exemple donné dans la partie 1.3.1 dans notre approche de composition comportementale. Elle applique les mécanismes de composition (partie 3.5.1 et partie 3.5.2).

La première partie s'intéresse à la manière dont les services sont conçus sous forme de composants. La deuxième partie présente la création de la configuration de composants qui est le résultat de la composition des composants et des contraintes d'ordonnancement.

3.6.1 Conception des composants

Cette partie est constituée de quatre sous-parties :

1. les deux premières concernent la description des composants dans le modèle Fractal (partie 3.2),
2. les deux dernières concernent la description des automates dans le modèle d'automates (partie 3.3).

3.6.1.1 Description des composants des services

L'exemple donné dans la partie 1.3.1 définit trois gestionnaires qui implantent deux propriétés non fonctionnelles :

1. Le gestionnaire de stockage est implanté dans notre approche sous la forme de deux composants :
   (a) un composant *StorageComponent*, qui charge les données compressées (l'interface serveur est constituée de la méthode *load*), conçu avec deux interfaces : une interface serveur *server* de type *StorageItf* (constituée de la méthode *load*) et une interface cliente *compress* de type *CompressItf*,
   (b) un composant *CompressComponent*, qui décompresse les données chargées par le composant *StorageComponent,* conçu avec une interface serveur *server* de type *CompressItf* (constituée de la méthode *decompress*).
   – le fait de définir deux composants permet de changer l'algorithme de compression sans changer l'algorithme de chargement des données,
2. Le gestionnaire de sécurité est implanté dans notre approche sous la forme de deux composants :
   (a) un composant *SecurityComponent,* qui vérifie si l'utilisateur authentifié peut exécuter les méthodes applicatives, conçu avec deux interfaces : une interface serveur *server* de type *SecurityItf* et une interface cliente *log* de type *LogItf*
   (b) un composant *LogComponent*, qui notifie l'ensemble des opérations effectuées par l'utilisateur, conçu avec une interface serveur *server* de type *LogItf,*

3. le gestionnaire de cryptage est implanté dans notre approche sous la forme d'un composant *EncryptionComponent,* conçu avec une interface serveur *server* de type *EncryptionItf.*

Le tableau 3.2 illustre la modélisation de ces composants. La formalisation graphique est identique à celle introduite dans le modèle Fractal.

Gestionnaires	Composants
Gestionnaire de stockage	server StorageComponent compress server CompressComponent
Gestionnaire de sécurité	server SecurityComponent log server LogComponent
Gestionnaire de cryptage	server EncryptionComponent

TAB. 3.2 – Modélisation des composants

3.6.1.2 Description des composants des contraintes d'ordonnancement

Lors de la composition des services, il apparait deux relations d'ordre qui représentent des dépendances comportementales :

1. la première relation concerne la vérification des permissions effectuée par le gestionnaire de sécurité et le chargement des données effectué par le gestionnaire de stockage. Les données ne peuvent être chargées qu'à partir du moment où la vérification a été réalisée.
2. la deuxième relation concerne la décompression des données effectuée par le gestionnaire de stockage et l'enregistrement des opérations effectuées par l'utilisateur réalisé par le gestionnaire de sécurité. Les opérations enregistrées contiennent des informations relatives aux données chargées. Elles sont lisibles qu'à partir du moment où les données sont décompressées par le gestionnaire de stockage.

Ces relations d'ordre vont être chacune définie comme des contraintes d'ordonnancement.

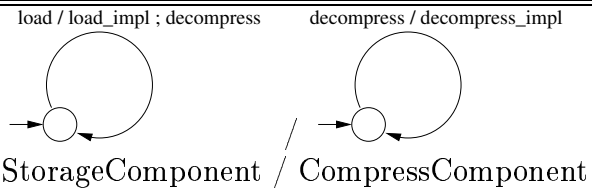
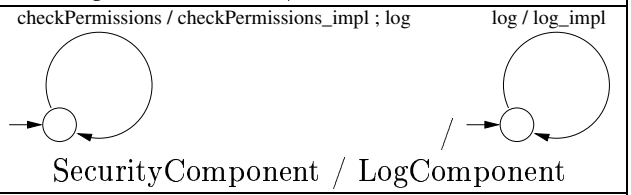
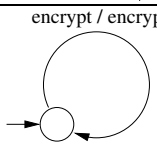
Le tableau 3.3présente la représentation graphique de ces contraintes sous forme de composants.

Contraintes d'ordonnancement	Composants
Première relation d'ordre	First scheduling constraint
Deuxième relation d'ordre	Second scheduling constraint

TAB. 3.3 – Contraintes d'ordonnancement

3.6.1.3 Description des automates des composants

Le tableau 3.4 présente les automates des composants décrits dans la partie 3.6.1.1.

Gestionnaires	Automates
Gestionnaire de stockage	
Gestionnaire de sécurité	
Gestionnaire de cryptage	

TAB. 3.4 – Modélisation des automates

Concernant le gestionnaire de stockage, l'automate du composant *StorageComponent* est constitué d'un seul état et d'une seule transition. A la réception de l'événement *load* défini sur l'interface serveur du composant, l'action associée à la transition est une séquence de deux actions : l'exécution de la méthode *load_impl* fournie par l'implantation du composant et l'émission de l'événement *decompress* sur l'interface cliente du composant. L'automate du composant *CompressComponent* est constitué d'un seul état et d'une seule transition. A la réception de l'événement *decompress* défini sur l'interface serveur du composant, la méthode *decompress_impl* fournie par l'implantation du composant sera exécutée.

Concernant le gestionnaire de sécurité, l'automate du composant *SecurityComponent* est constitué d'un seul état et d'une seule transition. A la réception de l'événement *checkPermissions* défini sur l'interface serveur du composant, l'action associée à la transition est une séquence de deux actions : l'exécution de la méthode *checkPermissions_impl* fournie par l'implantation du composant et l'émission de l'événement *log* sur l'interface cliente du composant. L'automate du composant *LogComponent* est constitué d'un seul état et d'une seule transition. A la réception de l'événement *log* défini sur l'interface serveur du composant, la méthode *log_impl* fournie par l'implantation du composant sera exécutée.

Concernant le gestionnaire de cryptage, l'automate du composant *EncryptionComponent* est constitué d'un seul état et d'une seule transition. A la réception de l'événement *encrypt* défini sur l'interface serveur du composant, la méthode *encrypt_impl* fournie par l'implantation du composant sera exécutée.

3.6.1.4 Description des automates des contraintes d'ordonnancement

La première contrainte d'ordonnancement définit une relation d'ordre entre les actions *checkPermissions_impl* de l'automate du composant *SecurityComponent* et *load_impl* de l'automate du composant *StorageComponent.* Elle est représentée par un automate avec trois états et deux transitions illustré dans la figure 3.6. L'état initial signifie que l'action *checkPermissions_impl* peut être exécutée. L'automate de la contrainte quitte l'état initial et entre dans le deuxième état dès que l'exécution de l'action *checkPermissions_impl* est terminée (il réagit à l'événement *end checkPermissions_impl action*, événement défini sur la transition entre l'état initial et le deuxième état). Le deuxième état signifie que

l'action *load_impl* peut être exécutée. L'automate de la contrainte quitte le deuxième état et entre dans le troisième état dès que l'exécution de l'action *load_impl* est terminée (il réagit à l'événement *end load_impl action*, événement défini entre le deuxième et le troisième état).

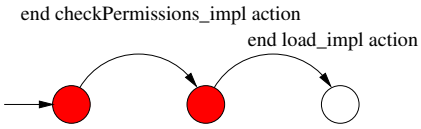


FIG. 3.6 – Contrainte d'ordonnancement *checkPermissions_impl / load_impl*

La deuxième contrainte d'ordonnancement définit une relation d'ordre entre les actions *log* de l'automate du composant *SecurityComponent* et *decompress* de l'automate du composant *StorageComponent*. Elle est représentée par un automate avec trois états et deux transitions illustré dans la figure 3.7. L'état initial signifie que l'action *decompress* peut être exécutée. L'automate de la contrainte quitte l'état initial et entre dans le deuxième état dès que l'exécution de l'action *decompress* est terminée (il réagit à l'événement *end decompress action*, événement défini sur la transition entre l'état initial et le deuxième état). Le deuxième état signifie que l'action *log* peut être exécutée. L'automate de la contrainte quitte le deuxième état et entre dans le troisième état dès que l'exécution de l'action *log* est terminée (il réagit à l'événement *end log action*, événement défini sur la transition entre le deuxième et le troisième état).

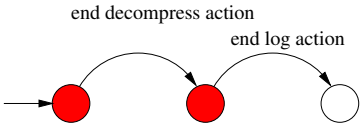


FIG. 3.7 – Contrainte d'ordonnancement *decompress / log*

Les automates sont hiérarchiques : un automate peut être inclus dans un état. Lorsque l'automate de la contrainte entre dans cet état, l'ordonnancement est géré par l'automate contenu dans cet état.

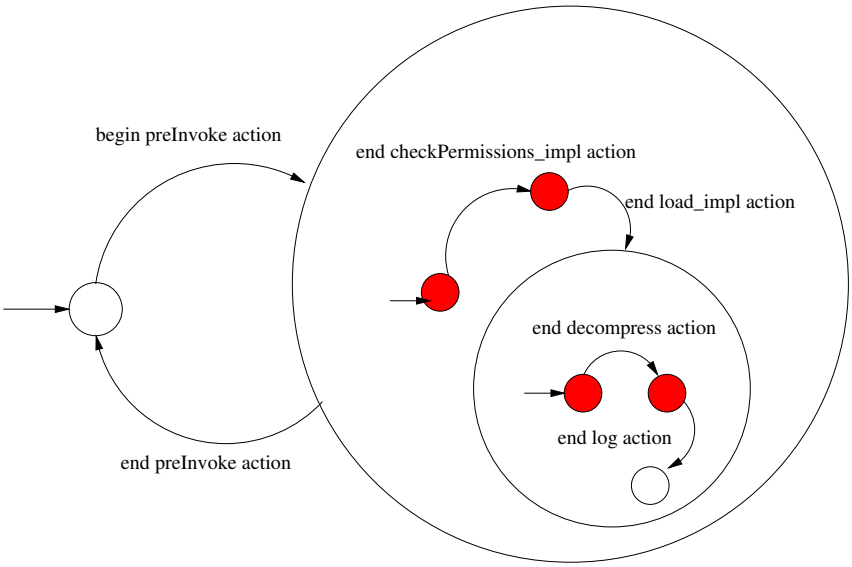


FIG. 3.8 – Contrainte d'ordonnancement

La figure 3.8 présente un automate où les deux contraintes sont prises en compte. La sémantique de cet automate n'est pas la même que la composition des contraintes illustrées dans les figures 3.6 et 3.7 : les différences sont la réinitialisation des contraintes à chaque action *preInvoke* et le fait que les actions *decompress* et *log* sont ordonnées que lorsque l'exécution de l'action *load_impl* est terminée.

Dans cet exemple, à chaque action *preInvoke* exécutée par l'automate *DispatchAutomaton*, la contrainte ordonne les actions *checkPermissions_impl* et *load_impl*, puis, lorsque l'exécution de l'action *load_impl* est terminée, la contrainte ordonne les actions *decompress* et *log*. Par rapport aux exemples précédents, cet exemple nous montre une contrainte d'ordonnancement complexe qui ordonne quatre actions de deux automates, et qui est conditionnée par les actions d'un troisième automate.

3.6.2 Conception de la configuration des composants

3.6.2.1 Composition structurelle

Le composant *StorageComponent* a besoin des fonctionnalités fournies par le composant *CompressComponent* pour décompresser les données. Cette dépendance structurelle se résoud par une interaction entre les composants, qui se traduit par la création d'une liaison entre l'interface cliente du composant *StorageComponent* et l'interface serveur du composant *CompressComponent*. La figure 3.9 présente cette liaison.

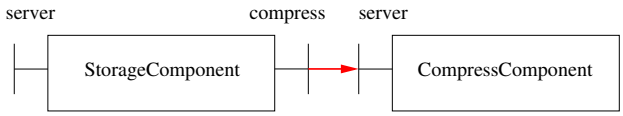


FIG. 3.9 – Gestionnaire de stockage

Le modèle de composants est hiérarchique. Le gestionnaire de stockage peut également être défini sous la forme d'un composite qui est constitué des composants *StorageComponent* et *CompressComponent*. La figure 3.10 illustre ce composite. Une liaison entre l'interface serveur interne du composite et l'interface serveur du composant *StorageComponent* est créée.

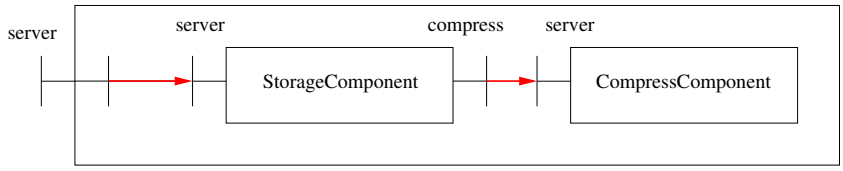


FIG. 3.10 – Gestionnaire de stockage composite

La création de la liaison entre l'interface *compress* du composant *StorageComponent* et l'interface *server* du composant *CompressComponent* se traduit par la création d'un canal d'événements où s'échangent les événements *decompress* (défini sur l'interface *compress* du composant *StorageComponent* et l'interface *server* du composant *CompressComponent*), et *end_decompress* correspondant à la fin de l'exécution de l'action initiée par l'événement *decompress* (défini au niveau de l'automate du composant *StorageComponent* et de l'automate du composant *CompressComponent*). Ainsi, toute émission de l'événement *decompress* provenant de l'action de l'automate du composant *StorageComponent* correspondra à la réception de l'événement *decompress* au sein de l'automate du composant

CompressComponent. La figure 3.11 montre la représentation graphique de la création du canal d'événements entre l'automate du composant *StorageComponent* et l'automate du composant *CompressComponent*.

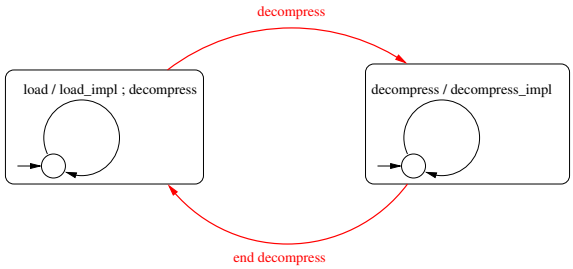


FIG. 3.11 – Automates du gestionnaire de stockage

La figure 3.12 présente la représentation graphique de la création du canal d'événements entre les automates des composants du gestionnaire de stockage lorsque ce dernier est implanté sous forme de composite (voir figure 3.10). Les automates du composant composite communiquent à l'extérieur par émission / réception d'événements.

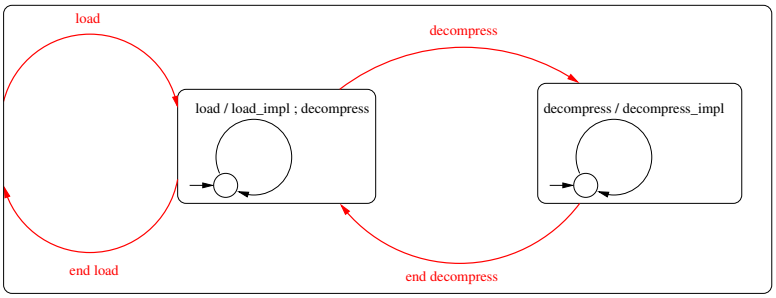


FIG. 3.12 – Automates du gestionnaire de stockage (composite)

3.6.2.2 Composition comportementale

La figure 3.13 illustre la modification des automates qui prennent en compte les points de synchronisation. L'application de la contrainte d'ordonnancement 3.6 se traduit par la modification de l'automate du composant *StorageComponent* qui définit l'action *load_impl* et de l'automate du composant *SecurityComponent* qui définit l'action *checkPermissions_impl*.

Un point de synchronisation qui gère le moment où l'action *load_impl* peut être exécutée est inséré au sein de l'automate du composant *StorageComponent*. Il est identifié par l'événement *load_impl synchronization point*. Une action d'émission de l'événement de fin d'exécution de l'action *load_impl* est également insérée pour indiquer quand l'action *load_impl* est terminée. Ces événements sont échangés entre l'automate du composant *StorageComponent* et l'automate de la contrainte.

Un point de synchronisation qui gère le moment où l'action *checkPermissions_impl* peut être exécutée est inséré au sein de l'automate du composant *SecurityComponent*. Il est identifié par l'événement *checkPermissions_impl synchronization point*. Une action d'émission de l'événement de fin d'exécution de l'action *checkPermissions_impl* (nommé *end checkPermissions_impl action*) est également insérée pour indiquer quand l'action *checkPermissions_impl* est terminée. Ces événements sont échangés entre l'automate du composant *SecurityComponent* et l'automate de la contrainte.

Les éléments en rouge sur la figure correspondent aux événements créés et insérés dans les automates qui communiquent entre eux avec ces événements. Les identifiants des événements de début et de fin d'action et des points de synchronisation sont générés dans l'implantation de cette approche et ne sont pas ceux indiqués dans les figures. Ils ne sont donnés dans les figures que pour améliorer la compréhension des mécanismes mis en oeuvre.

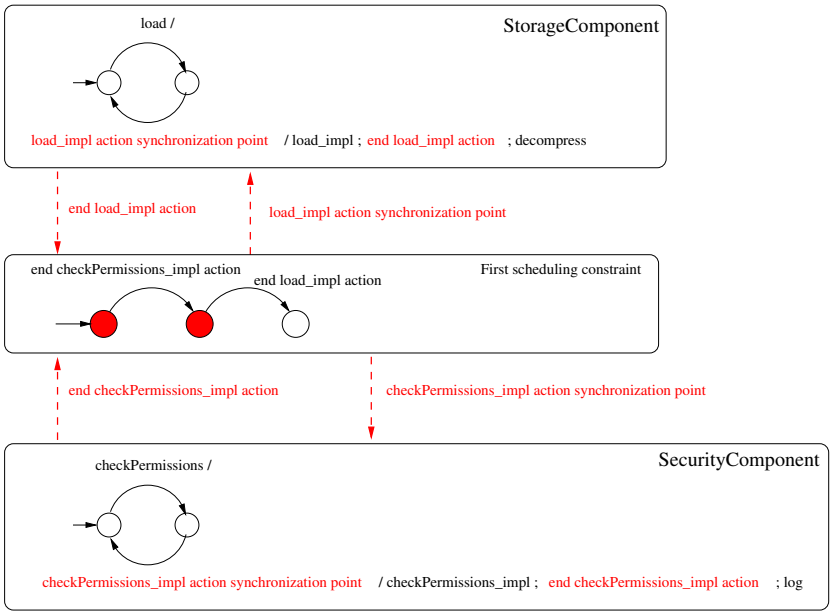


FIG. 3.13 – Application de la première contrainte d'ordonnancement

La figure 3.14 montre les échanges des événements entre les automates de composants et l'automate de la première contrainte d'ordonnancement lorsque les gestionnaires sont représentés sous forme de composite. L'automate du gestionnaire de stockage est le résultat de la composition de l'automate du composant *StorageComponent* et de l'automate du composant *CompressComponent* (non représenté dans la figure). L'automate de la contrainte émet les événements à destination de l'automate du gestionnaire de stockage. La remarque est la même pour le gestionnaire de sécurité.

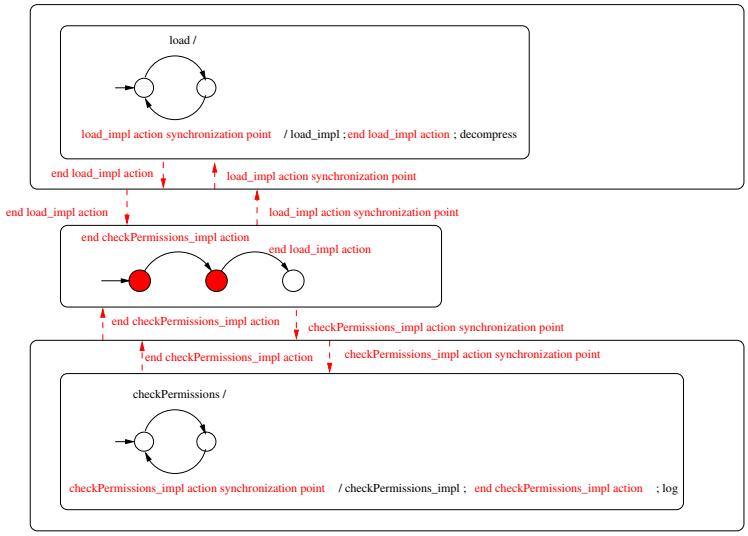


FIG. 3.14 – Application de la première contrainte d'ordonnancement (gestionnaires composites)

3.6.2.3 Architecture de la configuration

La composition des services se traduit par l'introduction du composant *DispatchComponent*. Il utilise les fonctionnalités des composants *StorageComponent*, *SecurityComponent* et *EncryptionComponent*. Dans notre exemple, l'interface serveur du composite qui représente la composition de l'ensemble des services correspond aux opérations réifiées de l'application (l'exemple est présenté avec la méthode *preInvoke)*.

La composition des gestionnaires s'effectue en composant les composants de ces derniers. L'automate du composant *DispatchComponent* exécute en parallèle les actions qui correspondent à l'exécution des composants *StorageComponent*, *SecurityComponent* et *EncryptionComponent* car les actions de leurs automates doivent être entrelacées.

La figure 3.15 illustre la composition des composants qui implantent les services techniques avant l'application des relations d'ordre au sein d'un composite qui correspond à la configuration des composants. Ce composite est un composant qui a une interface serveur qui est utilisée pour exécuter les services qui sont composés à l'intérieur du composite.

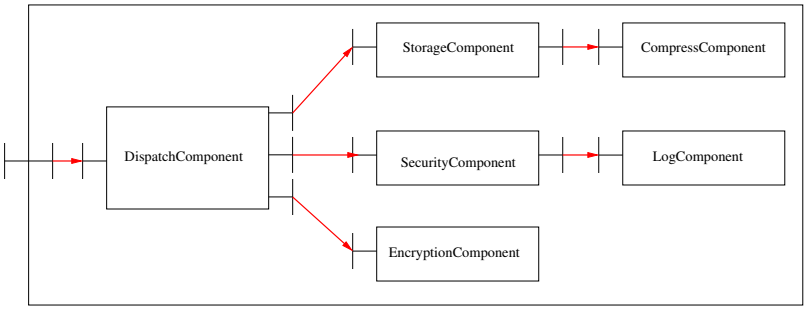


FIG. 3.15 – Composition de composants (avant application des relations d'ordre)

La figure 3.16 illustre la composition des automates des composants de la figure 3.15.

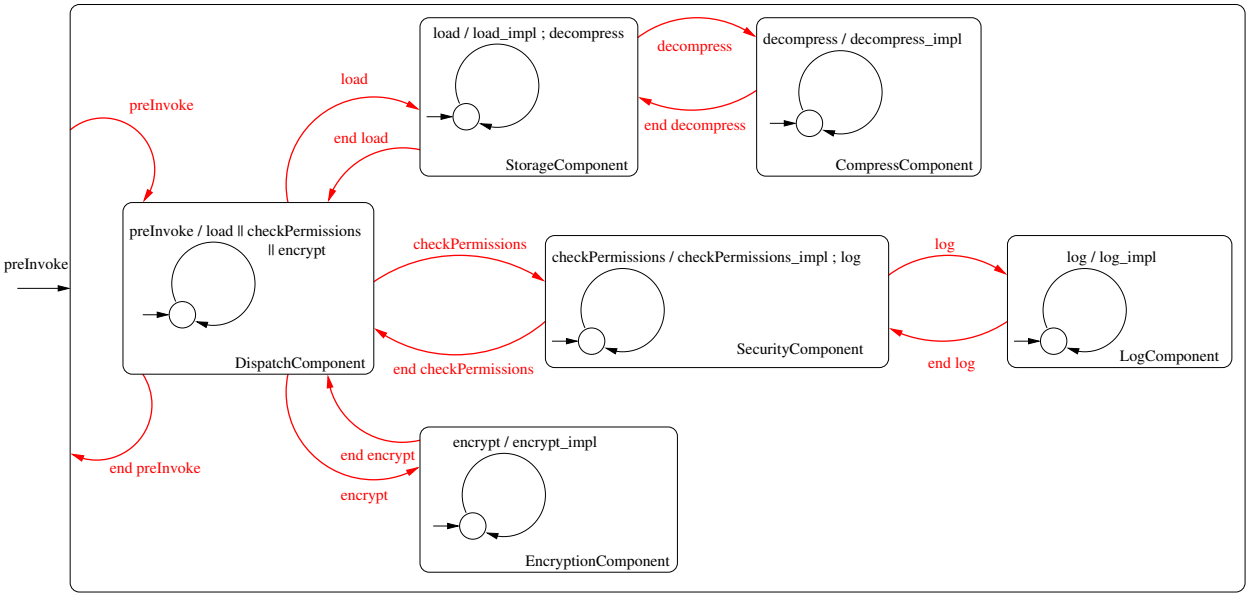


FIG. 3.16 – Composition d'automates (avant application des relations d'ordre)

Pour appliquer les relations d'ordre, les contraintes d'ordonnancement représentées par les figures 3.6 et 3.7 sont ajoutées au composite. La figure 3.17 présente la composition des composants. La figure 3.18 présente la composition des automates.

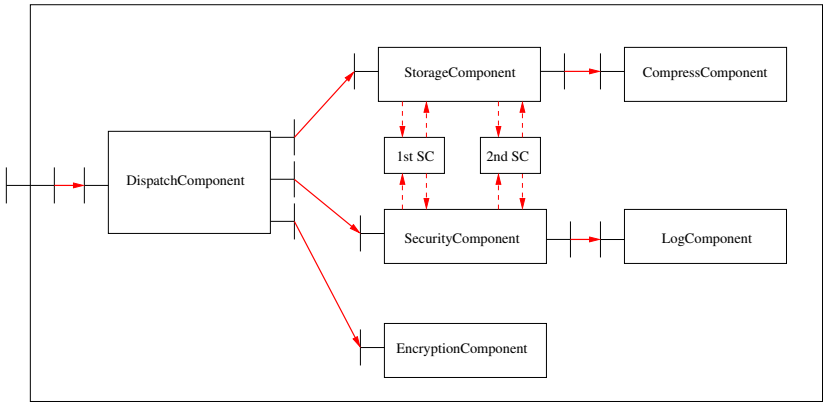


FIG. 3.17 – Composition de composants (après application des relations d'ordre)

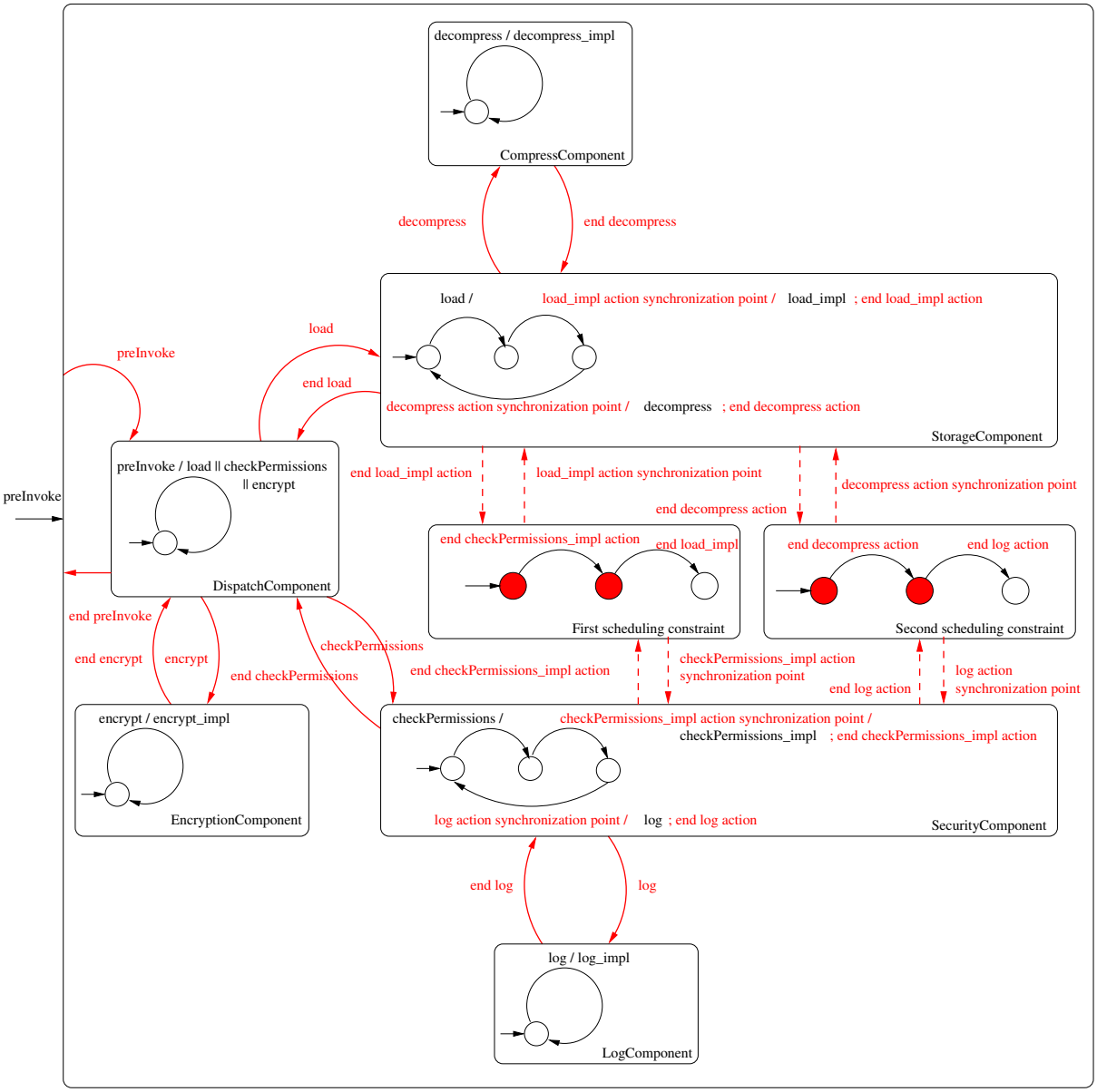


FIG. 3.18 – Composition d'automates (après application des relations d'ordre)

Le détail de l'exécution de l'exemple est donné dans la partie 5.7.

3.7 Conclusion

Ce chapitre a détaillé une nouvelle approche de composition appelée composition comportementale. Après avoir exposé les limites des approches étudiées dans le chapitre 2, l'approche a été définie pour supprimer ces limites. Pour cela, un modèle de composants, un modèle de comportement et un modèle d'expression de relations d'ordre ont été conçus. Les mécanismes de composition ont été définis à partir de ces modèles.

Notre approche permet de créer les interactions de type structurel et de type comportemental. La définition de ces interactions permettent d'avoir un haut degré de composabilité. L'évaluation de cette approche est effectuée dans la partie 6.2.

La composition des services correspond dans notre approche à la conception d'une configuration de composants. Lorsqu'il existe une dépendance structurelle entre deux composants, une liaison est créée entre les interfaces. Lorsqu'il existe une dépendance comportementale entre deux composants, une relation d'ordre peut être appliquée et la synchronisation s'effectue sur les automates qui décrivent le comportement des composants afin d'obtenir un entrelacement de l'exécution des opérations des composants conforme à la relation d'ordre. Afin d'avoir des modèles homogènes, cette relation d'ordre s'exprime avec un modèle d'automate et est conçue sous la forme d'un composant. La synchronisation s'appuie sur des échanges d'événements entre les automates des composants.

La composition des composants se traduit par une composition parallèle d'automates qui interagissent entre eux par échanges d'événements. Le comportement de la configuration est le résultat de la composition des automates des composants qui constituent cette configuration.

Le chapitre 4 décrit une manière d'implanter ces modèles.

Chapitre 4

Canevas de composition comportementale

4.1 Introduction

Ce chapitre décrit la mise en oeuvre de l'approche de composition comportementale exposée dans le chapitre 3. Cette mise en oeuvre correspond à la réalisation d'un canevas logiciel appelé *Brenda*. Le canevas logiciel ou *framework* fournit un cadre architectural qui permet d'implanter les concepts introduits dans notre approche décrite dans le chapitre 3. La définition d'un framework est : " A framework is a set of classes that embodies an abstract design for solutions to a family of related problems" [84].

Le canevas de composition comportementale est constitué :
- d'un canevas de composants (partie 4.2),
- d'un canevas de description d'automates (partie 4.3).

L'objectif de ce chapitre est de détailler l'architecture de ces deux canevas qui sont implantés en langage Java [19].

4.2 Canevas de composants

Cette partie présente le canevas Brenda qui implante l'approche de composition comportementale.

4.2.1 Le canevas Fractal

Le canevas Fractal [42] est l'implantation du modèle Fractal. Ce canevas est utilisé par le canevas de composants de l'approche de composition comportementale. Il est décrit en détail dans [41, 43]. La conception des composants repose sur des *templates*. Un composant s'instantie à partir d'un template. Le template se construit à partir du type du composant qui définit les interfaces du composant. Un template est un composant Fractal. Une fabrique de templates *TemplateFactory* permet de créer les templates.

Les composants disposent d'interfaces de contrôle qui leur permettent de gérer les composants. Ces interfaces sont implantées par des contrôleurs qui sont :
- le *BindingController*, qui gère les liaisons entre les interfaces des composants,
- le *ContentController*, qui gère le contenu du composant,
- le *LifeCycleController*, qui gère le cycle de vie du composant.

4.2.2 Le canevas Brenda

Le modèle de composants que nous avons utilisé dans l'approche de composition comportementale est le modèle Fractal. Le canevas Brenda est une extension du canevas Fractal.

Afin d'introduire les automates qui décrivent le comportement des composants, le canevas Brenda fournit deux interfaces *ReactiveTemplate* et *SchedulingConstraintTemplate* qui permettent d'instancier respectivement les composants et les contraintes d'ordonnancement.

L'interface *ReactiveTemplate* L'interface *ReactiveTemplate* permet d'instancier le template du composant. L'interface *ReactiveTemplate* étend l'interface *Template* de Fractal.

Algorithm 7 Interface *ReactiveTemplate*

```
interface ReactiveTemplate extends Template {
  String getBehaviourDesc() ;
  void setBehaviourDesc (String behaviourDesc) ;
}
```

La description du comportement du composant est fournie au template avec la méthode *setBehaviourDesc*. La description du comportement est étudiée dans la partie 4.3.

L'interface *SchedulingConstraintTemplate* L'interface *SchedulingConstraintTemplate* permet d'instancier le template d'une contrainte d'ordonnancement. Elle étend l'interface *ReactiveTemplate* : une contrainte d'ordonnancement est elle-aussi un composant.

Algorithm 8 Interface *SchedulingReactiveTemplate*

```
interface SchedulingConstraintTemplate extends ReactiveTemplate {
  void setImpactedTemplates(ReactiveTemplate[] tmpl) ;
}
```

Les contraintes d'ordonnancement s'appliquent sur le comportement des composants. La contrainte se définit donc à partir des composants qui sont impliqués dans l'ordonnancement : la méthode *setImpactedTemplates* permet de préciser quels sont les templates des composants qui sont impliqués.

4.3 Canevas de description d'automates

Cette partie présente le canevas de description d'automates qui permet de décrire le comportement des composants. La description des automates s'effectue de manière programmatique. Elle repose sur des fabriques (design pattern *Factory*) qui constituent le canevas et qui permettent de générer les éléments des automates. Ces fabriques sont :

1. *PrimitiveComponentBehaviourDescriptionFactory* : construction des automates des composants.
2. *SchedulingConstraintBehaviourDescriptionFactory* : construction des automates liés aux contraintes d'ordonnancement,
3. *StandardSchedulingConstraintBehaviourDescriptionFactory* : construction de contraintes d'ordonnancement simples. L'implantation de cette fabrique repose sur la fabrique

SchedulingConstraintBehaviourDescriptionFactory. Le détail de cette fabrique est donné en annexe.

Une quatrième fabrique *GenericBehaviourDescriptionFactory* est destinée à la génération des éléments communs aux automates. Les trois autres fabriques utilisent cette dernière.

4.3.1 La fabrique générique

La fabrique *GenericBehaviourDescriptionFactory* permet de créer des éléments des automates. Ces éléments sont communs aux automates des composants et des contraintes d'ordonnancement. Cette partie détaille la conception des automates dont le modèle est défini dans la partie 3.3.2.

4.3.1.1 L'automate

Les fabriques *PrimitiveComponentBehaviourDescriptionFactory* et *SchedulingConstraintBehaviourDescriptionFactory* créent respectivement les automates des composants (*PrimitiveComponentAutomaton*) et des contraintes d'ordonnancement (*SchedulingConstraintAutomaton*). Ces automates sont issus du même modèle. Un automate est défini par un ensemble d'états et un ensemble de transitions. Une transition se définit par un événement, une garde et une action.

L'automate *GenericAutomaton* implante le modèle d'automates. Il crée ses propres éléments :

1. les états,
2. les transitions.

Les états La méthode *createState* de l'interface de l'automate *GenericAutomaton* permet de créer les états de l'automate. L'état initial est défini avec la méthode *setInitialState*.

Le modèle est hiérarchique, donc le nouvel état peut contenir des automates. Cette affectation est effectuée avec la méthode *setInternalAutomata*. Des actions peuvent également être effectuées avec les méthodes *setAction*, *setPreAction* (action exécutée avant d'entrer dans cet état) et *setPostAction* (action exécutée après la sortie de cet état).

Les transitions Les transitions sont créées avec les méthodes de l'interface de l'automate *GenericAutomaton* :

1. *createEventGuardTransition,* pour les transitions constituées d'un événement et d'une garde,
2. *createEventTransition*, pour les transitions constituées d'un événement,
3. *createGuardTransition*, pour les transitions constituées d'une garde,

à partir d'un état source et d'un état cible.

4.3.1.2 Les templates

Dans le modèle d'automates, les transitions se définissent à partir d'événements, d'une garde et d'actions. La fabrique *GenericBehaviourDescriptionFactory* donne la possibilité de créer des *templates* qui permettent de concevoir ces éléments. Ces templates seront instanciés lors de la création des transitions. Un même template peut être instancié sur différentes transitions. L'instantiation de ces templates créent des occurences d'événement (dans le cas de templates d'événement) ou des occurences d'actions (dans le cas de templates d'action). Un ensemble d'occurences d'événements ou d'actions définies sur un automate est appelé *set* d'événements ou *set* d'actions.

Les templates d'événement Dans le modèle d'automates, un événement peut être :

1. une conjonction d'événements (la transition sera effectuée si les deux événements se produisent), dont le template sera créé à partir de la méthode *createAndEventTemplate*,
2. une disjonction d'événements (la transition sera effectuée si l'un des deux événements se produit), dont le template sera créé à partir de la méthode *createOrEventTemplate*.

Les templates de garde Une garde est une expression booléenne et est conçue avec la méthode *createGuardTemplate*.

Les templates d'action Dans le modèle d'automates, une action peut être :

1. une exécution séquentielle de deux actions, dont le template sera créé à partir de la méthode *createSequenceActionTemplate*,
2. une exécution parallèle de deux actions, dont le template sera créé à partir de la méthode *createParallelActionTemplate*.

4.3.2 La fabrique de description d'automates de composants

La fabrique *PrimitiveComponentBehaviourDescriptionFactory* construit des automates de composants. Elle hérite de la fabrique *GenericBehaviourDescriptionFactory*. Elle donne la possibilité de concevoir les éléments spécifiques aux automates des composants.

4.3.2.1 L'automate

La fabrique crée des automates de composant avec la méthode *createPrimitiveComponentAutomaton*. L'automate ainsi créé hérite de l'automate *GenericAutomaton*.

4.3.2.2 Les templates

Les templates d'événement Selon le modèle d'automates, un événement correspond à l'appel d'une opération définie sur une interface du composant.

Le template d'événement est défini à partir de la signature de l'événement avec la méthode *createEventTemplate* de la fabrique *PrimitiveComponentBehaviourFactory*. Cette signature est générée grâce à la fabrique *ReactiveTypeFactory* qui crée les signatures des événements à partir des signatures des méthodes définies sur les interfaces du composant.

Les templates d'action Selon le modèle d'automates, une action peut être :

1. une émission d'un événement qui correspond à une demande d'exécution d'une opération définie sur une interface cliente du composant. Le template est défini à partir de la signature de l'événement avec la méthode *createEventEmitActionTemplate*.
2. une exécution d'une méthode appartenant à l'implantation du composant. Le template est défini avec la méthode *createMethodInvocationActionTemplate*.

4.3.3 La fabrique de description d'automates de contraintes d'ordonnancement

La fabrique *SchedulingConstraintBehaviourDescriptionFactory* construit des automates de contrainte d'ordonnancement. Elle hérite de la fabrique *GenericBehaviourDescriptionFactory*. Elle donne la possibilité de concevoir les éléments spécifiques aux automates des contraintes.

4.3.3.1 L'automate

La fabrique crée des automates de composant avec la méthode *createSchedulingConstraintAutomaton*. L'automate ainsi créé hérite de l'automate *GenericAutomaton*.

4.3.3.2 Les templates

Les templates spécifiques aux contraintes d'ordonnancement sont les templates d'événements qui correspondent aux opérations effectuées par les automates et les templates d'action qui correspondent à la gestion des points de synchronisation.

Les templates d'événements Les événements auxquels l'automate de la contrainte réagira correspondent aux opérations effectuées par les automates.

Ces opérations sont définies dans la partie 3.4.2.1 :

1. le changement d'état :
   (a) l'entrée dans un nouvel état : l'événement associé est défini avec le template *BeginStateEventTemplate* créé par la méthode *createBeginStateEventTemplate*,
   (b) la sortie d'un état : l'événement associé est défini avec le template *EndStateEventTemplate* créé par la méthode *createEndStateEventTemplate*,
2. le début de l'exécution d'une action. Le début de cette exécution est définie :
   (a) par l'action elle-même. L'événement associé au début de l'exécution de cette action est défini avec le template *BeginActionEventTemplate* créé par la méthode *createBeginActionEventTemplate*. Trois cas sont à considérer :
      i. l'événement est généré pour le début de l'exécution d'une occurrence particulière d'action,
      ii. l'événement est généré pour le début de l'exécution de n'importe quelle action issue du même template,
      iii. l'événement est généré pour le début de l'exécution de n'importe quelle action issue du même template mais pour un automate particulier.
   (b) par l'événement qui provoquera l'exécution d'une action. Ainsi, l'événement généré correspond au début de l'exécution de n'importe quelle action provoquée par cet événement. L'événement généré est défini avec le template *BeginActionEventEventTemplate* créé par la méthode *createBeginActionEventEventTemplate*. Trois cas sont à considérer :
      i. l'événement est généré pour le début de l'exécution des actions provoquées par une occurence particulière d'événement,
      ii. l'événement est généré pour le début de l'exécution des actions provoquées par n'importe quel événement issu du même template,
      iii. l'événement est généré pour le début de l'exécution des actions provoquées par n'importe quel événement du même template mais pour un automate particulier.
3. la fin de l'exécution d'une action. La fin de cette exécution est définie :
   (a) par l'action elle-même. l'événement associé à la fin de l'exécution de cette action est défini avec le template *EndActionEventTemplate* créé par la méthode *createEndActionEventTemplate*. Trois cas sont à considérer :

i. l'événement est généré pour la fin de l'exécution dune occurrence particulière d'action,

ii. l'événement est généré pour la fin de l'exécution de n'importe quelle action issue du même template,

iii. l'événement est généré pour la fin de l'exécution de n'importe quelle action issue du même template mais pour un automate particulier.

(b) par l'événement qui provoquera l'exécution d'une action. Ainsi, l'événement généré correspond à la fin de l'exécution de n'importe quelle action provoquée par cet événement. L'événement généré est défini avec le template *EndActionEventEventTemplate* créé par la méthode *createEndActionEventEventTemplate*. Trois cas sont à considérer :

i. l'événement est généré pour la fin de l'exécution des actions provoquées par une occurence particulière d'événement,

ii. l'événement est généré pour la fin de l'exécution des actions provoquées par n'importe quel événement issu du même template,

iii. l'événement est généré pour la fin de l'exécution des actions provoquées par n'importe quel événement du même template mais pour un automate particulier.

Les points de synchronisation L'ordonnancement repose sur la définition de points de synchronisation (partie 3.4.3). Les points de synchronisation sont gérés par des actions définies sur les automates des contraintes d'ordonnancement. Ces actions sont créées par la fabrique de points de synchronisation *SynchronizationFactory* décrite dans la partie 5.3.

La méthode *getSynchronizationEventTemplate* définie sur les templates d'événement détaillés ci-dessus permet de mettre en oeuvre les points de synchronisation sur les opérations définies par les templates d'événement. Les points de synchronisation sont alors créés et insérés au sein des automates dont l'exécution est ordonnancée.

4.3.4 Utilisation de ces fabriques

Le canevas de composants utilise un descripteur qui référence la description des automates du composant. Les automates sont décrits de manière programmatique. Les automates des composants sont construits dans une classe Java.

4.3.4.1 Les automates de composant

La classe qui construit les automates d'un composant étend une des interfaces suivantes :

1. *ComponentBehaviourType* : elle définit la méthode *makeComponentBehaviour* que doit implanter la classe. Les paramètres de cette méthode sont l'ensemble des événements correspondant aux opérations des interfaces (serveur et client) du composant et la fabrique *PrimitiveComponentBehaviourDescriptionFactory*.
2. *ComponentImplBehaviourType* : cette interface diffère de la précédente par le fait que la méthode *makeComponentBehaviour* fournit l'implantation du composant qui est utilisée pour la création d'actions qui correspondent à l'exécution d'opérations définies au sein de l'implantation.

La méthode *makeComponentBehaviour* définie par ces interfaces est utilisée par l'implantation du canevas (partie 5.2). La fabrique crée un automate global à partir de l'ensemble des automates définis pour le composant avec la méthode *createComponentBehaviour*. Cette méthode est utilisée pour définir la valeur de retour de la méthode *makeComponentBehaviour*.

4.3.4.2 Les automates de contrainte d'ordonnancement

La classe qui construit les automates d'une contrainte d'ordonnancement étend une des interfaces suivantes :

1. *SchedulingConstraintBehaviourType* : elle définit la méthode *makeSchedulingConstraintBehaviour* que doit implanter la classe. Les paramètres de cette méthode sont les automates des composants qui sont impliqués dans l'ordonnancement et la fabrique *SchedulingConstraintBehaviourDescriptionFactory*.
2. *SchedulingConstraintImplBehaviourType* : cette interface diffère de la précédente par le fait que la méthode *makeSchedulingConstraintBehaviour* fournit l'implantation de la contrainte qui est utilisée pour la création de gardes qui utilisent des opérations définies au sein de l'implantation afin d'effectuer des évaluations.
3. *StandardSchedulingConstraintBehaviourType* : elle définit la méthode *makeSchedulingConstraintBehaviour* que doit implanter la classe. Les paramètres de cette méthode sont les automates des composants qui sont impliqués dans l'ordonnancement et la fabrique *StandardSchedulingConstraintBehaviourDescriptionFactory*.
4. *StandardSchedulingConstraintImplBehaviourType* : cette interface diffère de la précédente par le fait que la méthode *makeSchedulingConstraintBehaviour* fournit l'implantation de la contrainte.

Les automates impliqués dans l'ordonnancement sont fournis en paramètre de la méthode *makeSchedulingConstraintBehaviour*. Plus précisément, il s'agit d'interfaces Java qui correspondent aux éléments de ces automates (créés par la fabrique détaillée dans la partie 4.3.2). Ces interfaces permettent d'accéder à l'ensemble des éléments d'un automate.

4.4 Exemple

Cette partie illustre l'utilisation du canevas à partir de l'exemple détaillé dans la partie 3.6. Elle présente comment concevoir les composants à l'aide du canevas Brenda. La démarche est identique à celle du canevas de composants Fractal [41, 43]. Aussi, il est naturel que les interfaces du canevas Brenda soient les mêmes que celles du canevas Fractal. L'exemple est constitué de différentes phases :

1. conception de composants avec les templates et la description des automates (partie 4.4.1),
2. conception de la configuration pour assembler les composants (partie 4.4.2),
3. instantiation de la configuration des composants (partie 4.4.3).

4.4.1 Conception des composants

Cette partie présente l'utilisation des canevas de composant et de description d'automates pour concevoir les composants. Un composant est conçu à partir du *template* et de la description de son comportement.

4.4.1.1 Conception des templates

Le template du composant *StorageComponent* (dont la représentation est donnée dans le tableau 3.2) est créé de la manière suivante :

Algorithm 9 Création du template du composant *StorageComponent*

```
 1 Brenda.init(...);
 2 ComponentIdentity boot = Brenda.getBootstrapComponent();
 3 TypeFactory tf = Brenda.getTypeFactory(boot);
 4 ComponentType storageType = tf.createFcType(new InterfaceType{
 5   tf.createFcItfType("server", "apis.StorageItf", false, ...),
 6   tf.createFcItfType("compress", "apis.CompressItf", true, ...)});
 7 TemplateFactory rtf = Brenda.getTemplateFactory(boot);
 8 ComponentIdentity storageTmpl = rtf.createFcTemplate(
 9   storageType, "reactiveTemplate",
10   "staticReactiveContainer", "libs.StorageImpl");
11 Brenda.getReactiveTemplateController(storageTmpl).setBehaviourDesc(
12   "behaviours.StorageBehaviour");
```

Le composant *StorageComponent* dispose de deux interfaces :

1. l'interface *server* est une interface serveur de type *StorageItf* (ligne 5),
2. l'interface *compress* est une interface cliente de type *CompressItf* (ligne 6).

Le canevas Brenda fournit une fabrique *TemplateFactory* (ligne 7) qui permet de créer des templates de composants avec la méthode *createFcTemplate* (ligne 8, 9 et 10). La description du comportement du composant *StorageComponent* est fournie au template *storageTmpl* à partir de la méthode *setBehaviourDesc* via l'interface *ReactiveTemplate* (lignes 11 et 12). Les templates des autres composants sont conçus de la même manière.

Algorithm 10 Création du template du composant composite

```
1 ComponentType compositeType = ...
2 ComponentIdentity compositeTmpl = rtf.createFcTemplate(compositeType);
```

Le template du composant composite (dont la représentation est donnée dans la figure 3.15) est également créé avec la fabrique *TemplateFactory* (ligne 2).

Algorithm 11 Création du template de la première contrainte d'ordonnancement

```
1 ComponentIdentity scTmpl = rtf.createFcTemplate(null,
2    "schedulingConstraintTemplate", "staticReactiveContainer", ...);
3 Brenda.getSchedulingConstraintTemplate(scTmpl).setBehaviourDesc(
4                            "behaviours.scBehaviour");
5 Brenda.getSchedulingConstraintTemplate(scTmpl).setImpactedTemplates(
6                            new ReactiveTemplate[]{storageTmpl, securityTmpl});
```

Le template de la première contrainte d'ordonnancement (dont la représentation est donnée dans le tableau 3.3) est créé avec la fabrique *TemplateFactory*, comme tous les autres composants (lignes 1 et 2).

La sémantique de la contrainte est définie par la description de son comportement. Cette description est fournie au template par la méthode *setBehaviourDesc* (lignes 3 et 4). Les composants qui sont impliqués dans la contrainte sont les composants *StorageComponent* et *SecurityComponent* : la méthode *setImpactedTemplates* fournit au template de la contrainte d'ordonnancement les templates de ces composants (lignes 5 et 6).

4.4.1.2 Conception des automates

L'exemple se focalisera sur la conception

1. de l'automate du composant *StorageComponent* (dont la représentation est donnée dans le tableau 3.4*)*,
2. des automates des contraintes d'ordonnancement (dont la représentation est donnée dans les figures 3.6 et 3.7).

Automate *storageAutomaton* du composant *StorageComponent* Le code suivant construit l'automate *storageAutomaton* qui décrit le comportement du composant *StorageComponent*.

Algorithm 12 Création de l'automate *storageAutomaton* (1ère partie)

```
public class StorageBehaviour implements ComponentImplBehaviourType {
```

La classe *StorageBehaviour* construit l'automate décrivant le comportement du composant *StorageComponent*. Elle implante l'interface Java *ComponentImplBehaviourType*.

```
public ComponentBehaviour makeComponentBehaviour (
                              ReactionComponentType ct,
                              Class implantation,
                              PrimitiveComponentBehaviourDescriptionFactory bf) {
```

La méthode *makeComponentBehaviour* fournit en paramètre le type du composant, son implantation et la fabrique de description d'automates.

```
InterfaceType rit = ct.getInterfaceType("server");
```

L'automate réagit aux événements reçus par l'interface serveur nommée *server*.

```
InterfaceEventType[] types = rit.getEventTypes();
InterfaceEventType load = null;
for (int i = 0; i < types.length; i++) {
   if (types[i].getMethod().getName().equals("load")) {
      load = types[i];
      break;
   }
}
```

A partir de l'ensemble des événements de l'interface *server*, l'événement *load* est sélectionné à partir de sa signature.

```
EventTemplate loadEventTmpl = bf.createEventTemplate(load);
```

Le template d'événement *loadEventTmpl* est créé avec la fabrique *PrimitiveComponentBehaviourDescriptionFactory* à partir de la signature de l'événement *load*.

```
Method loadMethod = implantation.getMethod("load_impl", null);
MethodInvocationActionTemplate loadActTmpl =
         bf.createMethodInvocationActionTemplate(loadMethod);
```

A partir de l'implantations de la méthode *load_impl,* le template d'action *loadActTmpl* est créé.

```
InterfaceType ritComp = ct.getInterfaceType("compress");
InterfaceEventType decomp = ...
```

L'automate peut émettre des événements sur l'interface cliente *compress* afin d'interagir avec l'automate du composant *CompressComponent*, A partir de l'ensemble des événements définis sur l'interface cliente *compress*, l'événement *decompress* est sélectionné à partir de sa signature.

Algorithm 13 Création de l'automate *storageAutomaton* (2ème partie)

```
EventEmitActionTemplate decActTmpl = bf.createEventEmitActionTemplate(decomp);
```

Le template d'action *decActTmpl* correspondant à l'émission de l'événement *decompress* sur l'interface *compress* est créé.

```
PrimitiveComponentAutomaton ca =
            bf.createPrimitiveComponentAutomaton("storageAutomaton");
```

L'automate nommé *storageAutomaton* est créé.

```
State st = ca.createState("st");
```

L'état nommé *st* est créé (figure 4.1).

```
ca.createEventTransition(st, st, loadEventTmpl,
                         bf.createSequenceActionTemplate(loadActTmpl, decActTmpl));
```

Lorsque le signal *load* est reçu, la transition s'effectue. L'automate quitte l'état *st*, exécute la méthode *load_ impl*, émet l'événement *decompress* et retourne dans l'état *st* (figure 4.2).

```
ca.setInitialState(st);
```

L'état initial est l'état *st* (figure4.3).

```
return bf.createPrimitiveComponentBehaviour(ca);
```

Le comportement du composant est décrit par l'automate *storageAutomaton*.

```
}
}
```

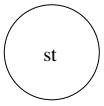


FIG. 4.1 – Création de l'état *st*

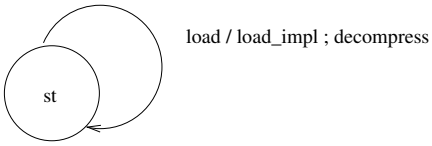


FIG. 4.2 – Création de la transition

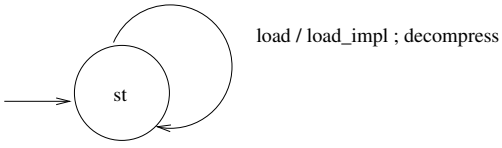


FIG. 4.3 – Création de l'automate et de son état initial

Automate de la deuxième contrainte d'ordonnancement L'exemple qui suit est le code Java d'une contrainte d'ordonnancement qui ordonne l'action *decompress* et l'action *log*.

Algorithm 14 Création de l'automate de la deuxième contrainte d'ordonnancement

```
public class CompressLogBehaviour implements SchedulingConstraintBehaviourType {
```

La classe *CompressLogBehaviour* construit l'automate qui implante la sémantique de la deuxième contrainte d'ordonnancement. Elle implante l'interface Java *SchedulingConstraintBehaviourType.*

```
public ComponentBehaviour makeSchedulingConstraintBehaviour (
                              AccessBehaviour[] impactedBehaviours,
                              SchedulingConstraintBehaviourDescriptionFactory scdf) {
```

La méthode *makeSchedulingConstraintBehaviour* fournit en paramètre les automates impliqués dans la conception de la contrainte et la fabrique de description d'automates de contraintes.

```
AccessAction log = searchLog(impactedBehaviours,
        "securityAutomaton", "logNotify");
AccessAction decompress = searchDecompress(impactedBehaviours,
        "storageAutomaton", "decompress");
```

Les actions *decompress* et *logNotify* sont recherchées dans les automates impliqués dans la contrainte et fournis en paramètre. Les méthodes *searchLog* et *searchDecompress* ont été implantées dans cet exemple afin de rendre la lecture plus lisible. Elles sont détaillées dans l'algorithme 15.

```
SchedulingConstraintAutomaton automaton =
        scdf.createSchedulingConstraintAutomaton("compressLogConstraint");
```

L'automate nommé *compressLogConstraint* est créé à partir de la méthode *createSchedulingConstraintAutomaton* de la fabrique *SchedulingConstraintBehaviourDescriptionFactory*.

```
State st1 = automaton.createState();
State st2 = automaton.createState();
State st3 = automaton.createState();
```

Les états *st1*, *st2* et *st3* sont créés (figure 4.4).

```
SynchronizationEventTemplate decompressTmpl =
  scdf.createEndActionEventTemplate(decompress).getSynchronizationEventTemplate();
```

Le template *decompressTmpl* est créé pour concevoir l'événement correspondant à la fin de l'exécution de l'action *decompress* définie sur l'automate *storageAutomaton.* La méthode *getSynchronizationEventTemplate* indique que l'action *decompress* doit être ordonnée et donc qu'un point de synchronisation sera associé à l'exécution de cette action.

```
SynchronizationEventTemplate logTmpl =
      scdf.createEndActionEventTemplate(log)
              .getSynchronizationEventTemplate();
```

Le template *logTmpl* est créé.

```
automaton.createEventTransition(st1, st2, decompressTmpl);
```

La transition entre l'état *st1* et *st2* s'effectue lorsque l'action *decompress* est terminée (figure 4.5). La méthode *createEventTransition* permet de concevoir cette transition. Le template *decompressTmpl* est instancié. L'automate passera de l'état *st1* à l'état *st2* lorsque l'événement correspondant à la fin de l'exécution de l'action *decompress* sera émis.

```
automaton.createEventTransition(st2, st3, logTmpl);
```

La transition entre l'état st2 et st3 s'effectue lorsque l'action *logNotify* est terminée (figure 4.6).

```
automaton.setInitialState(st1);
```

L'état initial de l'automate est l'état *st1* (figure 4.7).

```
return scdf.createPrimitiveSchedulingConstraintBehaviour(automaton);
```

Le comportement de la contrainte d'ordonnancement (et donc sa sémantique) est donné par l'automate qui vient d'être construit.

```
    }
}
```

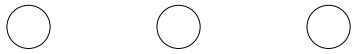


FIG. 4.4 – Création des états

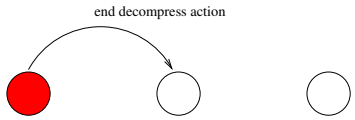


FIG. 4.5 – Création de la première transition

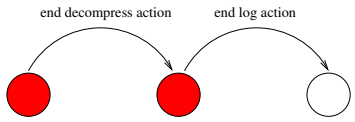


FIG. 4.6 – Création de la deuxième transition

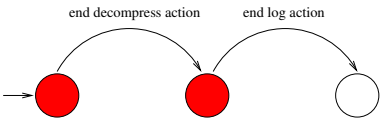


FIG. 4.7 – Création de l'automate de la contrainte et de l'état initial

Algorithm 15 Accès aux actions *log* et *decompress*

La méthode *searchDecompress* récupère la première occurrence de l'action *decompress* trouvée sur le comportement donné. Cette implantation n'est qu'un exemple : elle illustre un parcours pouvant être effectué dans les automates pour rechercher l'élément désiré. La méhode *searchLog* reprend les mêmes idées.

```
public AccessAction searchDecompress  (AccessBehaviour[] behaviours,
                                String automatonName,
                                String actionName) {
for (int i = 0 ; i < behaviours.length; i++) {
   AccessComponentAutomaton[] automata = b[i].getComponentAutomata(automatonName);
```

Sur l'ensemble des automates, on sélectionne ceux qui portent le nom donné en paramètre de la méthode *searchDecompress*.

```
      if (automata.length > 0) {
         AccessPrimitiveComponentAutomaton automaton =
                        (AccessPrimitiveComponentAutomaton) automata[0];
         AccessAutomatonEventEmitActionSet[] sets =
                        automaton.getEventEmitActionsSets();
```

La méthode *getEventEmitActionsSets* permet d'obtenir l'ensemble des actions qui correspondent à l'émission d'événements, actions étant définies au sein de l'automate.

```
         for (int j = 0 ; j < sets.length; j++) {
            AccessAutomatonEventEmitActionsSet set = sets[j];
               if (set.getTemplate().getEventType().getMethod().getName().equals(
                        actionName)) {
                  return set.getInstances()[0];
```

La méthode *searchDecompress* retourne l'action correspondant à l'émission de l'événement *decompress*.

```
               }
         }
      }
   }
   return null;
   }
```

Automate de la première contrainte d'ordonnancement L'exemple qui suit est le code Java d'une contrainte d'ordonnancement qui ordonne l'action *checkPermissions_ impl* et l'action *load_ impl*.

Algorithm 18 Instantiation du composite

```
ComponentIdentity resultComp =
        Brenda.getTemplate(compositeTmpl).instantiateFc() ;
```

Algorithm 16 Création de l'automate de la première contrainte d'ordonnancement

```
public class SecurityStorageBehaviour
       implements StandardSchedulingConstraintBehaviourType {
```

Pour illustrer l'utilisation de la fabrique *StandardSchedulingConstraintBehaviourDescriptionFactory*, la classe *SecurityStorageBehaviour* implante l'interface Java *StandardSchedulingConstraintBehaviourType.*

```
public ComponentBehaviour makeSchedulingConstraintBehaviour (
               AccessBehaviour[] impactedBehaviours,
               StandardSchedulingConstraintBehaviourDescriptionFactory scdf) {
AccessAction securityCheck = searchCheckPermissionsImpl(impactedBehaviours,
      "securityAutomaton", "checkPermissions_impl") ;
AccessAction storageLoad = searchLoadImpl(impactedBehaviours,
      "storageAutomaton", "load_impl") ;
return scdf.createActionsSchedulingConstraint("checkLoadConstraint",
      securityCheck, storageLoad) ;
```

La méthode *createActionsSchedulingConstraint* construit l'automate associé à la sémantique de la relation d'ordre.

```
}
...
}
```

4.4.2 Conception de la configuration des composants

La configuration est le résultat de la composition des composants dont la représentation est donnée dans la figure 3.17). Cette composition s'effectue avec les interfaces de contrôle *ContentController* et *BindingController.*

Algorithm 17 Conception de la configuration des composants

```
Brenda.getContentController(compositeTmpl).addFcSubComponent(storageTmpl) ;
Brenda.getContentController(compositeTmpl).addFcSubComponent(compressTmpl) ;
Brenda.getContentController(compositeTmpl).addFcSubComponent(scTmpl) ;
...
```

L'ajout de la contrainte d'ordonnancement met en oeuvre les interactions de type comportemental.

```
Brenda.getBindingController(storageTmpl).bindFc("compress",
   compressTmpl.getFcInterface("server")) ;
...
```

La création d'une liaison entre les interfaces des composants *StorageComponent* et *CompressComponent* permet de définir les interactions de type structurel entre les composants.

4.4.3 Instantiation de la configuration

L'instantiation du template du composite instantie le composant composite. Cette instantiation s'effectue avec la méthode *instantiateFc* définie sur l'interface *Template* du composant.

La configuration est un composant qui fournit un contrôleur de cycle de vie. Ce contrôleur active le composant.

Algorithm 19 Utilisation de la configuration

```
Brenda.getLifeCycleController(resultComp).startFc() ;
((apis.CompositeItf) resultComp.getFcInterface("server")).preInvoke() ;
```

4.5 Intégration de l'approche de composition comportementale

Un des objectifs de l'approche est de pouvoir être utilisée par les autres approches présentées dans le chapitre 2. Cet objecif rejoint les travaux présentés dans la partie 2.6.

Cette partie s'intéresse à l'intégration de l'approche de composition comportementale afin de mettre en oeuvre les mécanismes de composition de l'approche au sein d'approches de conception. La première section montre comment la configuration de composants peut utiliser des composants Fractal. La deuxième section présente comment le canevas Brenda peut être utilisé dans les autres approches étudiées dans le chapitre 2.

Ce travail rejoint les différents travaux de la partie 2.6. Il s'inscrit dans la direction des travaux de recherche actuels dans le domaine de la conception d'infrastructures adaptables.

4.5.1 Les composants

Cette section présente comment les composants conçus avec le canevas Brenda peuvent utiliser des composants externes à la configuration de composants.

4.5.1.1 Architecture de la configuration

Un composite conçu avec le canevas Brenda ne peut être constitué que de composants conçus avec ce canevas. Les interactions de type comportemental ne peuvent être définies qu'à l'intérieur d'un composite.

Les interactions de type comportemental ne s'appliquent pas sur les composants *CompressComponent*, *LogComponent* et *EncryptionComponent* : ces derniers peuvent donc être définis comme des composants externes réutilisés et composés avec les interfaces du composite comme illustré dans la figure 4.8.

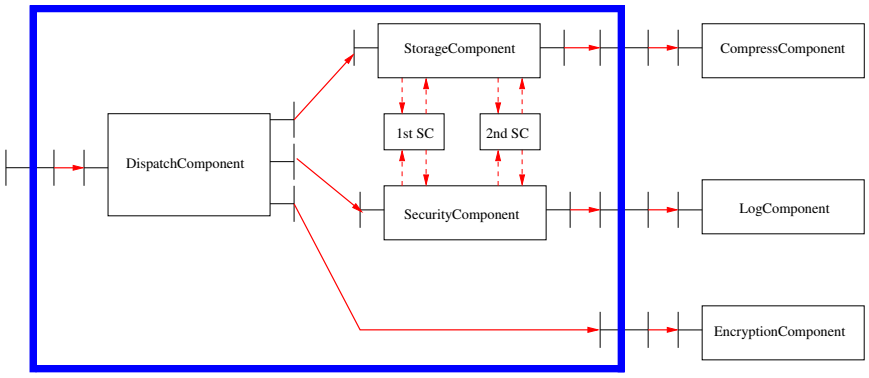


FIG. 4.8 – Configuration de composants

La figure 4.9 illustre la composition des automates des composants composés dans la figure 4.8. Elle montre que les interactions de type structurel définies entre les composants du composite et les composants externes au composite se traduisent par des échanges d'événements entre l'automate global du composite et son environnement.

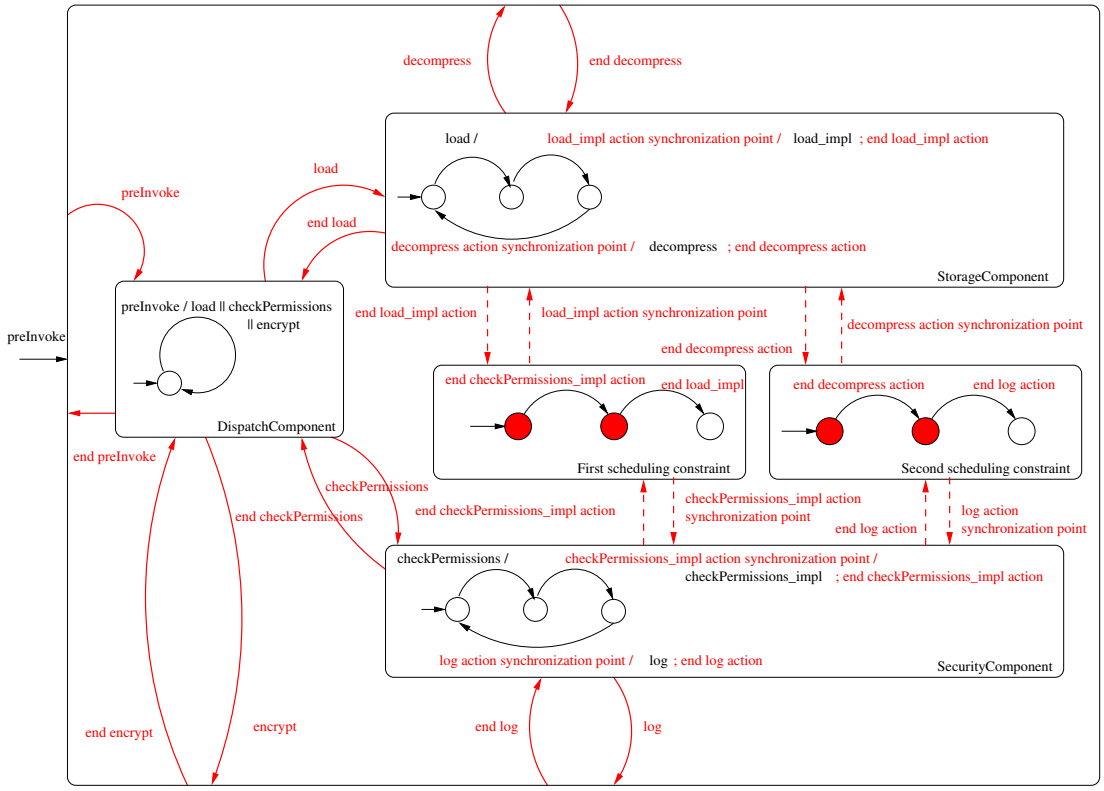


FIG. 4.9 – Composition hiérarchique des automates

4.5.1.2 Implantation de la configuration

Le code suivant est l'implantation de l'exemple donné dans la partie 4.5.1.1.

Algorithm 20 Implantation de la configuration

```
...
compressTmpl = tf.createFcTemplate(compressType, ...);
ComponentIdentity compress = Julia.getTemplate(compressTmpl).instantiateFc();
```

Le composant *compress* est instantié par l'implantation Julia du canevas Fractal.

```
...
Brenda.getBindingController(resultComp).bindFc(
        "compress", compress.getFcInterface("server");
```

Les liaisons entre le composite et le composant *compress* sont créées avec le contrôleur *BindingController* du composite.

```
...
```

4.5.2 Intégration dans les autres approches

Cette section montre un exemple d'intégration de l'approche de composition comportementale au sein d'une autre approche. Le but de cette intégration est d'appliquer les mécanismes de composition définis dans l'approche de composition comportementale au sein d'une autre approche afin d'améliorer le degré de composabilité qu'elle offre, ce qui respecte la philosophie des travaux exposés dans la partie 2.6. Par exemple, cette approche pourrait être utilisée pour implanter un conteneur EJB afin que ce dernier puisse plus facilement mettre en oeuvre de nouveaux services techniques.

Composants et AspectJ Le code suivant est un exemple d'intégration des composants au sein d'une approche orientée aspect étudiée dans le chapitre 2. L'implantation choisie est AspectJ.

Algorithm 21 Composants et AspectJ

```
aspect CompositionAspect {
  private CompositeItf getConcernsCompositionComposite() {...}
  pointcut businessmethodcalls (BusinessObject bo) :
                        target(bo) && call(public * * (..)) ;
  before(BusinessObject bo) : businessmethodcalls (bo) {
    ...
    composite = getConcernsCompositionComposite() ;
    composite.preInvoke(bo) ;}
...
}
```

L'aspect *CompositionAspect* est implante l'ensemble des services qui sont composés entre eux. Cette composition est réalisée au sein de l'implantation de la méthode *getConcernsCompositionComposite* qui construit la configuration des composants détaillée précédemment.

Cet algorithme montre clairement que la conception du canevas Brenda permet à celui-ci d'être utilisé dans les autres approches de conception, telles que les approches orientées conteneurs afin que ceux-ci puissent intégrer de nouveaux services.

4.6 Conclusion

Ce chapitre a présenté la mise en oeuvre de l'approche de composition comportementale décrite dans le chapitre 3. L'approche a été implantée sous la forme de canevas logiciels qui permettent de matérialiser les différents concepts de l'approche :

1. le premier canevas permet de concevoir les composants à partir du modèle de composants de l'approche, ce canevas se base sur le canevas Fractal,
2. le deuxième permet de concevoir les automates de ces composants à partir du modèle d'automates de l'approche.

Pour illustrer l'utilisation de ces canevas, l'exemple donné dans le chapitre 3 a été implanté à partir de ces canevas. Le résultat généré par l'approche peut être intégré dans les approches étudiées dans le chapitre 2. Le chapitre 5 détaille l'implantation des canevas.

Chapitre 5

Implantation

5.1 Introduction

Ce chapitre présente l'implantation des canevas de composants et de description d'automates. L'implantation a été conçue à partir de l'identification des différentes phases citées dans la partie 4.4 :

1. Lors de la conception des composants, les automates sont générés à partir de leur description. Cette phase met en oeuvre les éléments suivants :
   (a) le générateur d'automates (présenté dans la partie 5.2),
   (b) les fabriques de description d'automates : *GenericBehaviourDescriptionFactory*, *PrimitiveComponentBehaviourDescriptionFactory*, *SchedulingConstraintBehaviourDescriptionFactory*, *StandardSchedulingConstraintBehaviourDescriptionFactory*
2. Lors de la conception de la configuration, les composants sont assemblés en définissant des liaisons entre eux. Cette phase met en oeuvre les éléments suivants :
   (a) le générateur de points de synchronisation (présenté dans la partie 5.3),
   (b) le générateur de canaux d'événements (expliqué dans la partie 5.4),
3. Lors de l'instantiation de la configuration, les composants de la configuration sont instantiés pour être exécutés. Cette phase met en oeuvre les éléments suivants :
   (a) le générateur de machine réactive (présenté dans la partie 5.5).

Après avoir détaillé l'ensemble de ces éléments, la partie 5.6 donnera le fonctionnement de l'architecture de l'implantation des canevas. La partie 5.7 détaillera l'exécution de l'exemple.

5.2 Le générateur d'automates

Le générateur d'automates construit les automates à partir de leur description. Il est constitué de deux fabriques : *ComponentBehaviourFactory* et *SchedulingConstraintBehaviourFactory*.

5.2.1 La fabrique *ComponentBehaviourFactory*

Cette fabrique dont l'interface est donnée ci-dessous permet de générer les automates des composants :

- A partir de la description donnée par le paramètre *behaviourDesc*, l'automate est généré avec la méthode *createComponentBehaviour*.
- Le comportement d'un composite est le résultat de la composition des comportements de ses sous-composants : l'automate du composite est généré à partir des automates des sous-composants avec la méthode *addSubComponentBehaviour*. La méthode *createCompositeBehaviour* permet de générer l'automate du composite.

Algorithm 22 Interface de la fabrique *ComponentBehaviourFactory*

```
interface ComponentBehaviourFactory {
    ComponentBehaviour createComponentBehaviour (ReactiveComponentType ct, String behaviourDesc) ;
    ComponentBehaviour createComponentBehaviour (ReactiveComponentType ct, String implementationDesc,
                                                 String behaviourDesc) ;
    ComponentBehaviour createCompositeBehaviour (ReactiveComponentType ct) ;
    ComponentBehaviour addSubComponentBehaviour
                      (ComponentBehaviour compositeBehaviour,
                       ComponentBehaviour subComponentBehaviour) ;
}
```

5.2.2 La fabrique *SchedulingConstraintBehaviourFactory*

Cette fabrique dont l'interface est donnée ci-dessous permet de générer les automates des contraintes d'ordonnancement.

- A partir de la description donnée par le paramètre *behaviourDesc* et des automates impliqués par la contrainte, l'automate de la contrainte d'ordonnancement est généré avec la méthode *createSchedulingConstraintBehaviour*,
- La méthode *createStandardSchedulingConstraintBehaviour* permet de générer un automate décrit à partir de la fabrique *StandardSchedulingConstraintBehaviourDescriptionFactory*.

Algorithm 23 Interface de la fabrique *SchedulingConstraintBehaviourFactory*

```
interface SchedulingConstraintBehaviourFactory {
  ComponentBehaviour createSchedulingConstraintBehaviour
                        (String behaviourDesc, ComponentBehaviour[] impactedBehaviours) ;
  ComponentBehaviour createSchedulingConstraintBehaviour
                        (String implementationDesc,
                         String behaviourDesc, ComponentBehaviour[] impactedBehaviours) ;
  ComponentBehaviour createStandardSchedulingConstraintBehaviour
                        (String behaviourDesc, ComponentBehaviour[] impactedBehaviours) ;
  ComponentBehaviour createStandardSchedulingConstraintBehaviour
                        (String implementationDesc,
                         String behaviourDesc, ComponentBehaviour[] impactedBehaviours) ;
}
```

5.2.3 Implantation des fabriques

L'implantation de ces fabriques prennent en compte le fait que la description des automates est réalisée à partir des fabriques *PrimitiveComponentBehaviourDescriptionFactory, SchedulingConstraintBehaviourDescriptionFactory* et *StandardSchedulingConstraintBehaviourDescriptionFactory*. Cette implantation appelle donc les méthodes *makeComponentBehaviour* ou *makeSchedulingConstraintBehaviour* (pour les contraintes d'ordonnancement) qui implantent la description des automates en leur fournissant les fabriques correspondantes.

5.3 Le générateur de points de synchronisation

La synchronisation est le mécanisme utilisé pour ordonnancer l'exécution des composants. Elle se concrétise dans notre approche par l'insertion de points de synchronisation

au sein des automates.

Dans l'implantation des canevas, les points de synchronisation sont définis comme des actions ajoutées aux automates. Ces points de synchronisation sont gérés par les automates des contraintes d'ordonnancement. Les automates des contraintes d'ordonnancement utilisent des actions qui gèrent ces points de synchronisation. Ces actions sont instantiées à partir de templates appelés *SynchronizationPointControlActionTemplate*. Ces templates sont conçus avec la méthode *createSynchronizationPointControlActionTemplate* de la fabrique *SynchronizationFactory*.

Algorithm 24 Interface de la fabrique *SynchronizationFactory*

```
interface SynchronizationFactory {
  SynchronizationPointControlActionTemplate createSynchronizationPointControlActionTemplate() ;
}
```

Ces templates sont créés par les éléments qui entrent en jeu dans la synchronisation.

Le template *SynchronizationPointControlActionTemplate* créer avec la méthode *createSynchronizationPointActionTemplate* le template d'action correspondant aux points de synchronisation qui seront intégrés au sein des automates des composants impliqués dans l'ordonnancement. Le template *SynchronizationPointActionTemplate* ainsi généré sera automatiquement instantié dans l'automate du composant lors de l'ajout des points de synchronisation. Par exemple, si la contrainte porte sur le début de toutes les actions provoquées par un événement (ce même événement peut se retrouver dans différentes transitions d'un même automate), alors les points de synchronisation représentés par des actions *SynchronizationPointAction* seront insérés dans l'automate, c'est-à-dire qu'une séquence d'action sera créée avec l'action du point de synchronisation précédant l'action à ordonnancer, et ce pour toutes les actions provoquées par l'événement. Si un point de synchronisation existe déjà pour une action donnée, alors le nouveau point de synchronisation sera mis en parallèle avec l'autre point de synchronisation.

5.4 Le générateur de canaux d'événements

Les mécanismes de création de canaux d'événements détaillés dans les parties 3.5.1 et 3.5.2 sont implantés par le générateur de canaux. Le générateur de canaux est constitué d'une fabrique de liaisons *BindingFactory* dont l'interface est donnée ci-dessous.

Algorithm 25 Interface de la fabrique *BindingFactory*

```
interface BindingFactory {
  Binding createBinding (InternalInterfaceEventType evtOut,
                         InternalInterfaceEventType evtIn) ;
  Binding createBinding (InternalSynchronizationPointSignalType syncOut,
                         InternalSynchronizationPointSignalType syncIn) ;
}
```

Les interactions entre les automates sont des échanges de signaux. Elles sont matérialisées par les canaux d'événements. Les canaux sont créés avec les méthodes *createBinding* de la fabrique *BindingFactory* :

1. la première méthode crée les canaux qui matérialisent les interactions de type structurel. Cette méthode est utilisée par le canevas de composants lors de la conception de configurations de composants. Les paramètres sont :
   (a) *evtOut* (événement qui sera émis par l'automate) dont le type est le type d'événement défini sur une interface cliente,

(b) *evtIn* (événement qui sera reçu par l'automate) dont le type est le type d'événement défini sur une interface serveur.

2. la deuxième méthode crée les canaux qui matérialisent les interactions de type comportemental. Cette méthode est utilisée par le canevas de composants lors de l'instantiation des configurations. Les paramètres sont :

   (a) *syncOut* (événement qui sera émis par l'automate de la contrainte d'ordonnancement) dont le type est le type d'événement correspondant au point de synchronisation,

   (b) *syncIn* (événement qui sera reçu par l'automate du composant) dont le type est le type d'événement correspondant au point de synchronisation.

5.5 Le générateur de machine réactive

Le générateur de machine réactive permet de créer l'environnement d'exécution qui implante le modèle d'exécution qui respecte le modèle de concurrence défini dans la partie 3.4.3.

5.5.1 Le modèle d'exécution

L'implantation du modèle de concurrence défini dans la partie 3.4.3 est réalisée par un langage de programmation. Cette section s'intéresse au langage Esterel (et à ses propriétés) qui implante le modèle de concurrence et à son environnement qui a été utilisé dans l'implantation du générateur de machine réactive. Elle explique les raisons du choix de ce langage pour exécuter les automates des composants.

5.5.1.1 Le langage Esterel

Le langage Esterel [31, 29, 16, 32, 27, 30, 54, 70, 28] est un langage synchrone [72], qui signifie que les processus concurrents s'exécutent et échangent des informations dans un temps conceptuel nul, et réactif, qui signifie que le programme réagit aux stimuli provenant de son environnement. Le langage utilise un style impératif.

Le modèle d'automates défini dans la partie 3.3.2 est traduit dans le langage Esterel pour que les automates puissent être exécutés. La traduction du modèle est détaillée dans la partie 5.5.3.

Le langage Esterel introduit deux classes de primitives :

1. les primitives impératives standard, telles que l'émission de signal, la séquence, la boucle, etc (ces primitives ont un temps nul),
2. les primitives temporelles comme les triggers, les boucles temporelles, etc.

Le langage s'appuie sur une sémantique mathématique donnée dans [30, 54, 28].

5.5.1.2 L'environnement logiciel

La machine réactive La compilation d'un programme écrit dans le langage Esterel produit une machine réactive qui réagit aux stimuli de l'environnement. Dans notre approche de composition comportementale, la machine réactive correspond aux automates Esterel conçus à partir des automates des composants. Elle réagit aux appels de méthodes définies sur les interfaces serveur du composite. Le composite définit les limites de la bulle synchrone.

La compilation Le compilateur a été développé par une équipe du laboratoire DTL/ASR de France Télécom R&D. La technique de compilation de ce compilateur s'appuie sur l'identification des points d'arrêt dans le programme Esterel. Le code généré s'appuie sur ces points d'arrêt. Cette technique est détaillée dans [135]. Une extension a été développée pour générer le code de la machine réactive dans le langage Java.

5.5.2 Les éléments architecturaux du générateur de machine réactive

Le générateur construit, à partir de l'automate global issu de la composition des composants, la machine réactive générée à partir du code Esterel. Le générateur de machine réactive est responsable de la conversion du modèle d'automates présenté dans la partie 3.3.2 vers le modèle défini par Esterel.

Le générateur est constitué :

1. d'un convertisseur (partie 5.5.3), qui construit un arbre syntaxique de code Esterel à partir de l'automate global,
2. d'un générateur de code (partie 5.5.4), qui génère le code Esterel correspondant à l'arbre syntaxique,
3. d'un compilateur (partie 5.5.5), qui compile le code Esterel pour générer le code Java qui correspond à la machine réactive,
4. d'un générateur de code de la membrane (partie 5.5.6), qui génère le code Java qui permet d'exécuter la machine réactive,
5. d'un chargeur de code (partie 5.5.7), qui compile et instantie le code Java généré.

5.5.3 Le convertisseur

Le convertisseur traduit le modèle d'automates défini dans la partie 3.3.2 vers le modèle défini par le langage Esterel. Il est constitué de différentes fabriques.

5.5.3.1 L'arbre syntaxique

L'arbre syntaxique correspondant au code Esterel est construit par les convertisseurs présentés dans la partie 5.5.3.2.

La grammaire du langage Esterel [27] a été implantée avec des classes Java qui correspondent aux noeuds de l'arbre. La correspondance entre la grammaire du langage Esterel et les classes Java sont indiquées dans :

- le tableau 7.1 pour les modules Esterel,
- le tableau 7.2 pour l'interface d'un module Esterel,
- le tableau 7.3 pour le corps d'un module Esterel,
- le tableau 7.4 pour les expressions qui mettent en oeuvre des données,
- le tableau 7.5 pour les expressions qui mettent en oeuvre les signaux,
- le tableau 7.6 pour les expressions qui mettent en oeuvre les délais,
- le tableau 7.7 pour les primitives.

Ces tableaux sont donnés en annexe.

La figure 5.1 représente une partie de l'arbre syntaxique de l'automate *storageAutomaton* (présenté dans la figure 3.4). Chaque noeud de l'arbre a un type (donné entre parenthèse). Chaque élément des automates possède un identificateur. Le code Esterel est donné en annexe.

FIG. 5.1 – Arbre syntaxique de l'automate *storageAutomaton* (traduit en module *component_5_component_3_storageAutomaton*)

5.5.3.2 Les fabriques

La fabrique *EsterelConverterFactory* crée un convertisseur *EsterelConverter* à partir de la méthode *createConverter*. Un *EsterelConverter* est le convertisseur qui traduit les automates qui définissent le comportement d'un composant en automates Esterel.

Il dispose d'une méthode *convert* qui fournit une *ModulesList* qui correspond à la racine de l'arbre syntaxique.

Algorithm 26 Interface de la fabrique *EsterelConverterFactory*

```
interface EsterelConverterFactory {
  public EsterelConverter createConverter(InternalBehaviour behaviour);
}
```

Algorithm 27 Interface du convertisseur *EsterelConverter*

```
interface EsterelConverter {
  public ModulesList convert();
}
```

De même qu'il existe trois fabriques de description d'automates (décrites dans la partie4.3), il existe deux fabriques de convertisseurs :

1. *EsterelComponentAutomatonConverterFactory* : cette fabrique génère un convertisseur *EsterelComponentAutomatonConverter* (qui étend l'interface *EsterelConverter*) pour convertir un automate d'un composant,
2. *EsterelSchedulingConstraintAutomatonConverterFactory* : cette fabrique génère un convertisseur *EsterelSchedulingConstraintAutomatonConverter* (qui étend l'interface *EsterelConverter*) pour convretir un automate d'une contrainte d'ordonnancement.

Algorithm 28 Interface de la fabrique *EsterelComponentAutomatonConverterFactory*

```
interface EsterelComponentAutomatonConverterFactory {
  public EsterelComponentAutomatonConverter createConverter
                              (InternalPrimitiveComponentAutomaton automaton);
}
```

Algorithm 29 Interface de la fabrique *EsterelSchedulingConstraintAutomatonConverterFactory*

```
interface EsterelSchedulingConstraintAutomatonConverterFactory {
  public EsterelSchedulingConstraintAutomatonConverter createConverter
                              (InternalSchedulingConstraintAutomaton automaton);
}
```

Pour chaque automate, un convertisseur dédié est créé par la fabrique correspondante. Cette création est à l'initiative du convertisseur *EsterelConverter*. Ainsi, chaque automate peut avoir un convertisseur dédié qui sera généré en fonction de ses caractéristiques.

L'implantation actuelle des convertisseurs d'automates utilise l'algorithme de conversion décrit dans la partie 5.5.3.3. Elle est constituée de trois convertisseurs :

1. le convertisseur *StateBasedEsterelGenericAutomatonConverter* concerne la partie générique des convertisseurs, elle convertit les états et les transitions des automates,

2. le convertisseur *StateBasedEsterelComponentAutomatonConverter* étend l'implantation *StateBasedEsterelGenericAutomatonConverter*, et convertit les éléments spécifiques des automates des composants (émission de signaux, etc),
3. le convertisseur *StateBasedEsterelSchedulingConstraintAutomatonConverter* étend l'implantation *StateBasedEsterelGenericAutomatonConverter*, et convertit les éléments spécifiques des automates des contraintes d'ordonnancement (émission de signaux correspondant aux points de synchronisation, etc).

Ainsi, à tout niveau, de nouvelles implantations de convertisseur peuvent être réalisées.

5.5.3.3 Algorithme de conversion

L'algorithme de conversion réalisé dans l'implantation actuelle sera décomposé en trois parties :

1. la première partie concerne la conversion des éléments communs des automates,
2. la deuxième partie concerne la conversion des éléments des automates des composants,
3. la troisième partie concerne la conversion des éléments des automates des contraintes.

L'annexe explicite le code Esterel généré à partir de l'exemple.

Pour toutes ces parties, chaque élément de l'automate est associé à un identificateur unique.

5.5.3.4 Algorithme de conversion du convertisseur générique

Un automate d'un composant se traduit par un module Esterel. Un automate est constitué :

1. d'**états** : un état d'un automate se traduit par un *TrapStatement*. Dès que la transition est effectuée, un signal qui correspond à la sortie de l'instruction *trap* est émis (*ExitStatement*). L'état d'un automate peut être constitué d'autres automates : ces automates sont traduits chacun en module Esterel et l'état est traduit par une exécution parallèle de ces modules dans l'action du trap.
2. de **transitions**
   (a) Les éléments d'une transition sont l'événément, la garde et l'action. Tous les événement des automates sont traduits en signaux. Une conjonction d'événements est traduite en *AndSignalsExpression*. Une disjonction d'événements est traduite en *OrSignalsExpression*. Les gardes sont des expressions booléennes qui sont traduites en *DataExpression*. Une action se traduit par un *Statement*. Une exécution parallèle d'actions est traduite en *ParallelStatement*. Une exécution séquentielle d'actions est traduite en *SequenceStatement*.
   (b) pour chaque transition correspond à un signal qui est émis si la transition doit être effectuée. Trois cas se présentent :
      i. la transition se définit par un événement, donc si l'événement se produit, alors la transition doit s'effectuer : la transition est traduite par un *EveryStatement* qui, si le signal correspondant à l'événement est présent dans l'instant synchrone, émet le signal correspondant à l'exécution de la transition,
      ii. la transition se définit par une garde : la transition est traduite par un *EveryStatement* qui, si l'expression booléenne correspondant à la garde est évaluée à vrai, émet le signal correspondant à l'exécution de la transition,

iii. la transition se définit par un événement et une garde : la transition est traduite par un *EveryStatement* qui, si le signal correspondant à l'événement est présent dans l'instant synchrone et si l'expression booléenne correspondant à la garde est évaluée à vrai, émet le signal correspondant à l'exécution de la transition.

5.5.3.5 Algorithme de conversion du convertisseur d'automates de composants

Cette partie s'intéresse à la conversion des éléments spécifiques des automates des composants. La fabrique de description d'automates de composants décrite dans la partie 4.3.2 construit des templates d'événements qui sont définis sur les interfaces des composants et des templates d'actions. Seules les instances issues de ces templates sont utilisées dans l'automate, donc l'algorithme de conversion ne traduit que les événements et actions déclarés avec ces templates.

Les événements définis sur les interfaces sont traduits en signaux. Ces signaux sont déclarés dans l'interface du module (*SignalDeclaration*) soit en *input* pour les événements définis sur l'interface serveur du composant, soit en *output* pour les événements définis sur les interfaces clientes du composant.

Une action est :

- soit l'émission d'un événement défini sur une interface cliente du composant : cette action se traduit par un *EmitStatement* sur un signal qui correspond à la traduction de l'événement,
- soit l'exécution d'une opération de l'implantation du composant : cette action se traduit soit par un *ProcedureCallStatement* sur une procédure déclarée (*ProcedureDeclaration*) qui correspond à une opération de l'implantation, soit par un *FunctionCallExpression* sur une fonction déclarée (*FunctionDeclaration*) qui correspond à une opération de l'implantation qui retourne une valeur. Les fonctions et les procédures sont prises en charge par le générateur du code de la membrane qui génère le code qui exécutera les opérations respectives de l'implantation,
- soit une action qui correspond à un point de synchronisation : cette action se traduit par un *AwaitStatement* sur un signal qui correspond à la traduction de l'événement émis par la contrainte qui gère ce point de synchronisation.

Les paramètres des opérations sont sérialisés. Le résultat de cette sérialisation devient la valeur du signal correspondant à cette opération. Le code de la sérialisation / désérialisation est généré par les convertisseurs.

Les composants Un composant se traduit par un module Esterel qui exécute en parallèle les modules qui correspondent à la traduction de tous les automates de ce composant. Un composite se traduit également par un module Esterel qui exécute en parallèle les modules correspondant aux sous-composants (y compris les contraintes).

Les échanges d'événements qui matérialisent la liaison entre une interface serveur d'un composant et l'interface cliente d'un autre se traduisent par des émissions et des réceptions de signaux : à la réception de chaque signal correspondant à un événement défini sur l'interface cliente, un signal, correspondant à l'événement défini sur l'interface serveur qui a la même signature que l'événement de l'interface cliente, est émis. Cette traduction est un couple *EveryStatement* / *EmitStatement*. Ces signaux sont déclarés au sein de l'interface du module qui correspond à la traduction du composite.

Les échanges d'événements entre les composants et les contraintes d'ordonnancement suivent la même logique.

5.5.3.6 Algorithme de conversion du convertisseur d'automates de contraintes

Cette partie s'intéresse à la conversion des éléments spécifiques des automates des contraintes. La fabrique de description d'automates de contraintes décrite dans la partie 4.3.3 construit des templates d'événements et des templates d'actions. Seules les instances issues de ces templates sont utilisées dans l'automate, donc l'algorithme de conversion ne traduit que les événements et actions déclarés avec ces templates.

Les événements sont traduits en signaux. Ces signaux sont déclarés dans l'interface du module (*SignalDeclaration*) soit en *input* pour les événements correspondant aux opérations effectuées par les automates des composants , soit en *output* pout les événements correspondant aux points de synchronisation.

Les actions qui gèrent les points de synchronisation définies à partir des templates *SynchronizationPointControlActionTemplate* créés par le générateur de points de synchronisation (partie 5.3) sont traduits en *SustainStatement* sur un signal correspondant au point de synchronisation.

Les contraintes Une contrainte, tout comme un composant, se traduit par un module Esterel qui exécute en parallèle les modules qui correspondent à la traduction de tous les automates de cette contrainte.

5.5.4 Le générateur de code Esterel

La génération du code Esterel consiste à générer le code Esterel dans un fichier à partir de l'arbre syntaxique construit par le convertisseur.

La fabrique *EsterelGeneratorFactory* permet de créer un générateur de code *EsterelGenerator* avec la méthode *createGenerator*.

Algorithm 30 Interface du générateur *EsterelGenerator*

```
interface EsterelGenerator {
  File generate (ModulesList modules) ;
}
```

La méthode *generate* crée un fichier contenant le code Esterel correspondant à l'arbre syntaxique donné en paramètre.

5.5.5 Le compilateur

Le code Esterel doit être compilé pour être exécuté. La fabrique *EsterelCompilerFactory* crée un compilateur *EsterelCompiler* dont la méhode *compile* génère un fichier contenant le code source écrit dans le langage hôte (Java dans notre cas) à partir du fichier *strlFile* qui contient le code Esterel dont le nom du module principal est donné par le paramètre *moduleName*.

Algorithm 31 Interface du compilateur *EsterelCompiler*

```
interface EsterelCompiler {
  File compile (String moduleName, File strlFile) ;
}
```

La compilation utilise le compilateur de France Télécom R&D [135]. L'implantation de l'interface *EsterelCompiler* utilise ce compilateur.

5.5.6 Le générateur du code de la membrane

Le code de la membrane correspond au code qui exécute la machine réactive générée par les éléments précédents. La membrane est celle du composite instantié. La génération de ce code s'effectue par la méthode *generate* du générateur *EsterelMembraneCodeGenerator* créé à partir de la fabrique *EsterelMembraneCodeGeneratorFactory*.

Algorithm 32 Interface du générateur *EsterelMembraneCodeGenerator*

```
interface EsterelMembraneCodeGenerator {
  File generate (InternalComponentBehaviour behaviour, ReactiveComponentType cType) ;
}
```

Le code de la membrane est constitué de deux parties :

1. une partie qui exécute la machine réactive. Ce code contient :
   (a) l'émission des signaux qui correspond à l'appel des méthodes définies sur les interfaces serveur du composite,
   (b) l'émission des signaux qui correspond à la fin d'exécution des méthodes définies sur les interfaces clientes du composite.
2. une partie qui est appelé par la machine réactive. Ce code contient :
   (a) la réception des signaux émis par la machine réactive, et qui correspond à la fin d'exécution des méthodes définies sur les interfaces serveur du composite,
   (b) la réception des signaux émis par la machine réactive, et qui correspond à l'appel des méthodes définies sur les interfaces clientes du composite,
   (c) l'appel des méthodes implantées dans les implantations des sous-composants.

La classe *BasicEsterelMembraneCodeGenerator* est une implantation de base d'un générateur *EsterelMembraneCodeGenerator*. Elle prend en compte ces deux parties :

1. elle génère une classe Java pour la première partie. Elle est donnée en paramètre à la machine réactive lors de son instantiation.
2. elle génère une classe Java pour la deuxième partie. Elle implante l'interface Java *MembraneCode* dont la méthode *MC_start* qui initialise la machine réactive. Cette méthode est appelée par le contrôleur *LifeCycleController*. Elle étend la classe précédente et gère le cycle de vie de la machine réactive. A chaque appel d'une méthode d'une interface serveur du composant, elle émet un signal correspondant à cette méthode et fait réagir la machine réactive.

Le code de la membrane généré pour l'exemple est donné en annexe.

5.5.7 Le chargeur de code

Le code Java généré (code de la membrane et code de la machine réactive) doit être compilé et chargé en mémoire pour être exécuté. La fabrique *EsterelLoaderFactory* crée un chargeur *EsterelLoader* qui, à partir de la méthode *load*, charge les classes Java générées.

Algorithm 33 Interface du chargeur *EsterelLoader*

```
interface EsterelLoader {
  Object load(File javaStrlFile, String mcClassName, File javaMCFile) ;
}
```

5.5.8 Architecture du générateur de machine réactive

Cette partie s'intéresse à l'architecture du générateur de machine réactive. Dans l'implantation actuelle, le générateur s'appuie sur l'environnement Esterel et est constitué de convertisseurs d'automates, d'un générateur de code, d'un compilateur Esterel, d'un générateur de code de membrane, et d'un chargeur de code. Le fonctionnement de cette architecture est matérialisé par un processus d'instantiation *BehaviourInstantiationProcess*. Il est alors possible de définir de nouveaux processus d'instantiation si l'environnement logiciel change.

5.5.8.1 La conception d'un processus d'instantiation

Un processus d'instantiation *BehaviourInstantiationProcess* définit le fonctionnement du générateur de machine réactive.

Algorithm 34 Interface du processus d'instantiation

```
interface BehaviourInstantiationProcess {
  public Object build() ;
}
```

La méthode *build* est appelée par une *BehaviourInstanceFactory*.
Ce processus est créé à partir de la fabrique *BehaviourInstantiationProcessFactory*.

Algorithm 35 Interface de la fabrique *BehaviourInstantiationProcessFactory*

```
interface BehaviourInstantiationProcessFactory {
  public BehaviourInstantiationProcess createBehaviourInstantiationProcess
            (InternalComponentBehaviour behaviour, ReactiveComponentType cType) ;
}
```

L'utilisation d'un processus d'instantiation a lieu lors de l'instantiation des composants. Cette instantiation repose sur une fabrique *BehaviourInstanceFactory* qui crée des instances de composants en utilisant le générateur de machine réactive avec la méthode *createComponentBehaviourInstance*.

Algorithm 36 Interface de la fabrique *BehaviourInstanceFactory*

```
interface BehaviourInstanceFactory {
  Object createComponentBehaviourInstance
            (ComponentBehaviour behaviour, ReactiveComponentType cType) ;
}
```

5.5.8.2 L'implantation du processus d'instantiation Esterel

Dans l'implantation actuelle, le générateur de machine réactive s'appuie sur l'environnement Esterel. Le processus d'instantiation implanté est donc dédié à l'environnement Esterel.

Le code Java de l'implantation d'un processus d'instantiation pour la génération en Esterel (*EsterelBehaviourInstantiationProcess*) est le suivant :

Algorithm 37 Code Java d'un processus d'instantiation

```
class EsterelBehaviourInstantiationProcess
             implements BehaviourInstantiationProcess {
  public Object build() {
    EsterelConverter converter = converterFactory.createConverter (behaviour);
    ModulesList modules =
               converter.convert();
    File strlFile =
               gen.generate(modules);
    File javaStrlFile =
               compiler.compile (modules.getMainModule().getName(), strlFile);
    File javaMCFile =
               mcGen.generate (behaviour, cType);
    return loader.load (javaStrlFile,
                        namingFact.createMembraneCodeGeneratedClassName(behaviour),
                        javaMCFile);
  }
}
```

1. la première ligne correspond à la création des convertisseurs des automates,
2. la deuxième ligne correspond à la conversion des automates sous forme d'arbre syntaxique du langage Esterel, cette conversion est réalisée par un *EsterelConverter*,
3. la troisième ligne correspond à la génération du fichier Esterel *strlFile* à partir de l'arbre syntaxique, cette génération est réalisée par un *EsterelGenerator*,
4. la quatrième ligne compile le fichier Esterel *strlFile* et crée le fichier Java *javaStrlFile* correspondant à l'implantation de la machine réactive, cette compilation est réalisée par un *EsterelCompiler*,
5. la cinquième ligne correspond à la génération du code de la membrane du composant, cette génération est réalisée par un *EsterelMembraneCodeGenerator*,
6. la dernière ligne compile les fichiers Java (la machine réactive et la membrane du composant) et génère le bytecode correspondant, puis le charge et instantie les classes, cette instantiation est réalisée par un *EsterelLoader*.

La figure 5.2 détaille ce processus d'instantiation. Le template dispose de la description des automates, du type et de l'implantation du composant. Le processus d'instantiation génère la machine réactive qui prend en charge l'entrelacement des séquences d'exécution du composant.

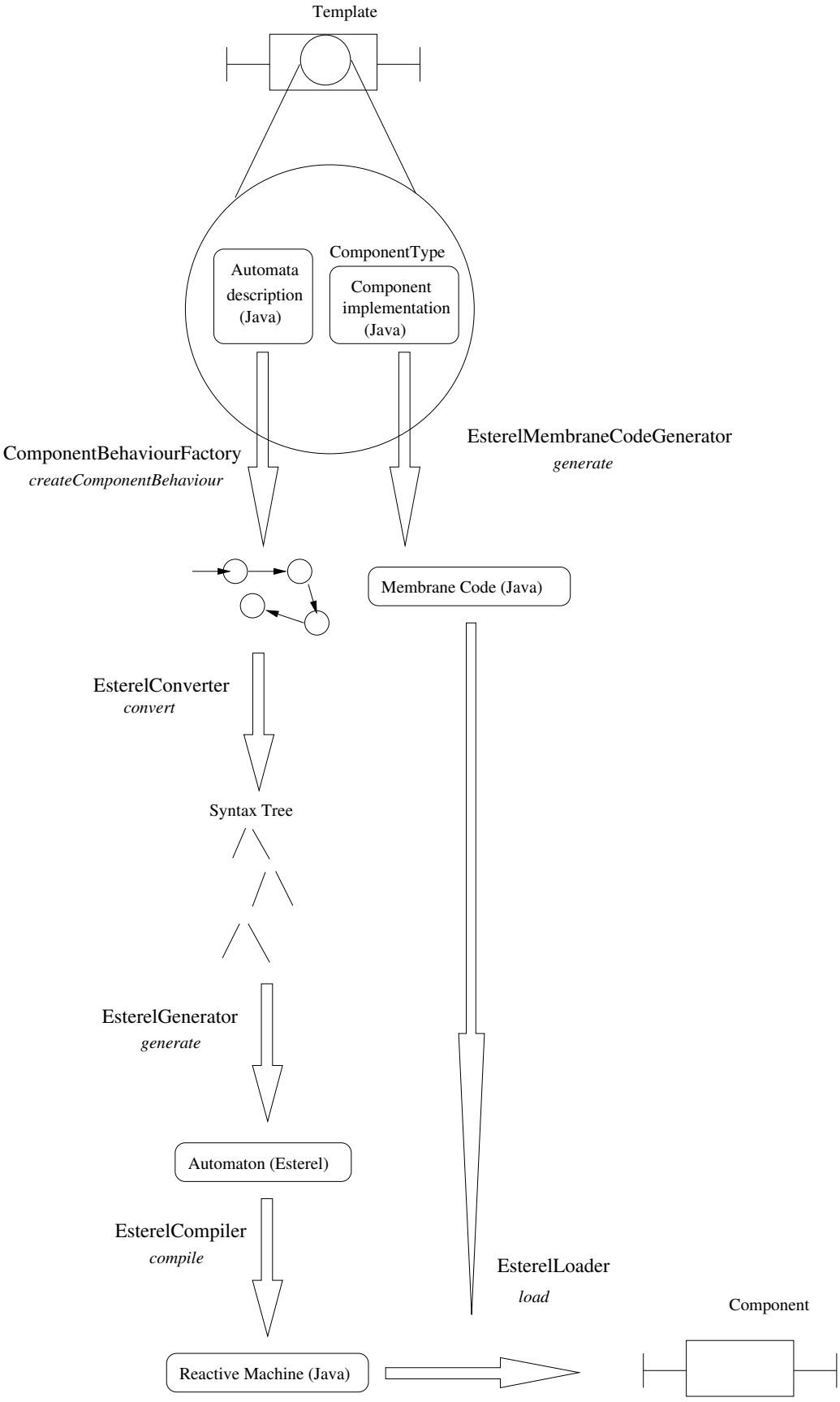


FIG. 5.2 – Processus d'instantiation

Algorithm 38 Interface de la fabrique *BootstrapFactory*

```
interface BootstrapFactory {
  ComponentBehaviourFactory createComponentBehaviourFactory();
  SchedulingConstraintBehaviourFactory createSchedulingConstraintBehaviourFactory();
  BindingFactory createBindingFactory();
  SynchronizationFactory createSynchronizationFactory();
  BehaviourInstanceFactory createBehaviourInstanceFactory();
}
```

5.6 Implantation des canevas

5.6.1 Implantation du canevas de composants

L'implantation du canevas de composants est constituée des implantations des templates *ReactiveTemplate* et des contrôleurs *BindingController*, *ContentController*, et *LifeCycleController*. Ces implantations utilisent les différents générateurs exposés précédemment.

L'ensemble de ces générateurs sont regroupés dans la fabrique *BootstrapFactory* qui permet de créer les fabriques de ces générateurs.

Les méthodes *createComponentBehaviourFactory* et *createSchedulingConstraintBehaviourFactory* créent respectivement les fabriques pour générer les automates des composants et des contraintes d'ordonnancement. La méthode *createBindingFactory* crée la fabrique de liaisons. La méthode *createBehaviourInstanceFactory* crée la fabrique qui permet d'instancier un composant.

5.6.1.1 Les templates

Un template de composant permet de concevoir et d'instancier un composant. De ce fait, l'implantation du template utilise le générateur d'automates pour la conception et le générateur de machine réactive pour l'instantiation.

Le template *ReactiveTemplate* La figure 5.3 donne le diagramme de séquence illustrant la génération des automates d'un composant. Le template crée la fabrique *ComponentBehaviourFactory* à partir de la fabrique *BootstrapFactory* afin de générer les automates à partir de leur description.

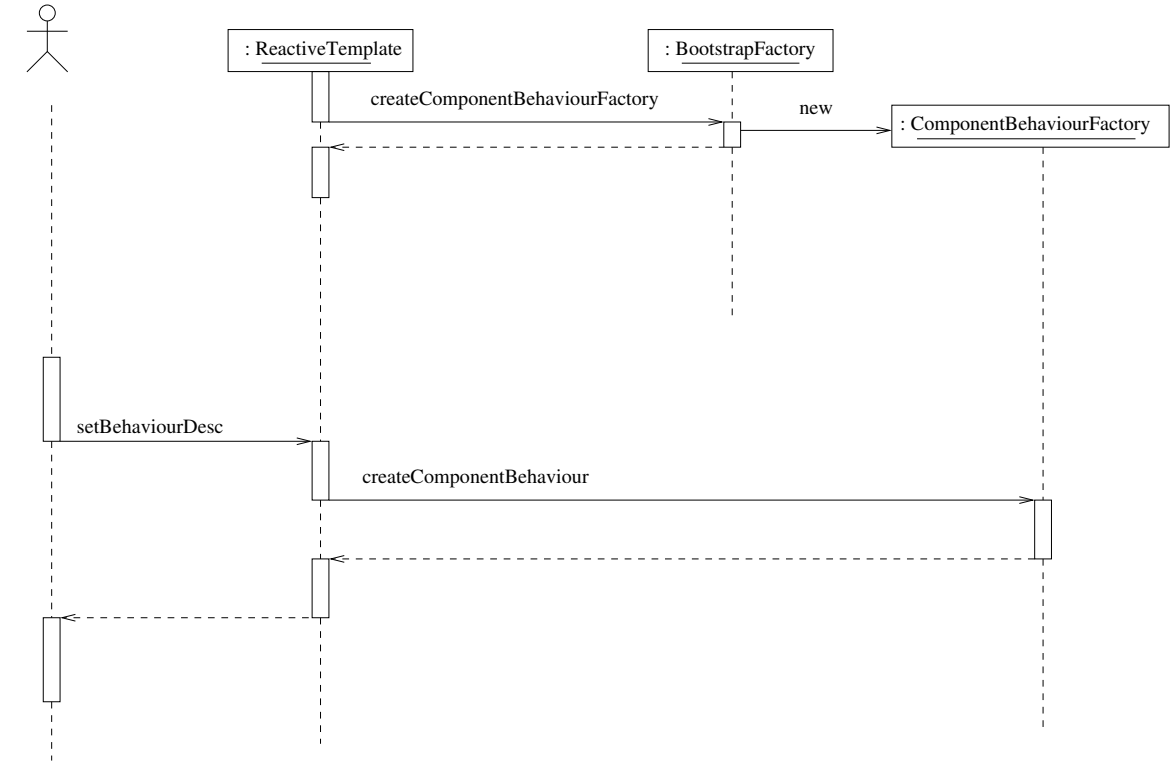


FIG. 5.3 – Génération des automates d'un composant

La figure 5.4 donne le diagramme de séquence présentant l'instantiation du composant. Le template crée un process d'instantiation à partir de la fabrique *BehaviourInstantiationProcessFactory* créée par la *BootstrapFactory*. Le process d'instantiation permet alors de concevoir la machine réactive qui prend en charge l'exécution du composant.

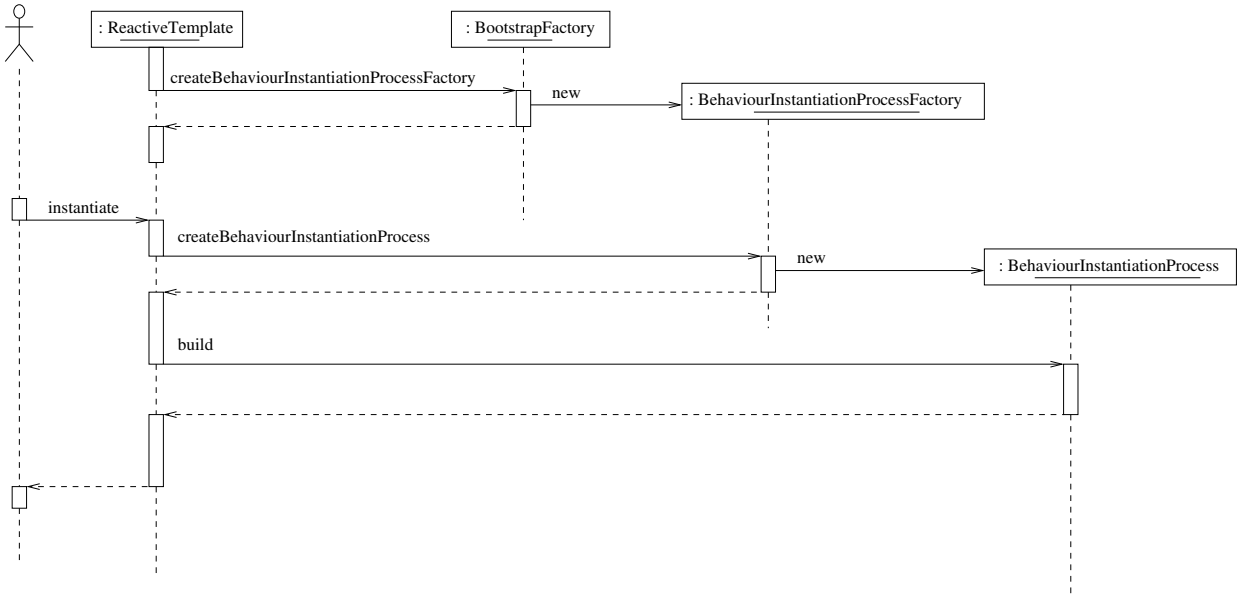


FIG. 5.4 – Instantiation du composant

La figure 5.5 montre l'instantiation du template du composant *StorageComponent*. La membrane du composant contient la machine réactive. Le contenu du composant contient l'implantation du composant.

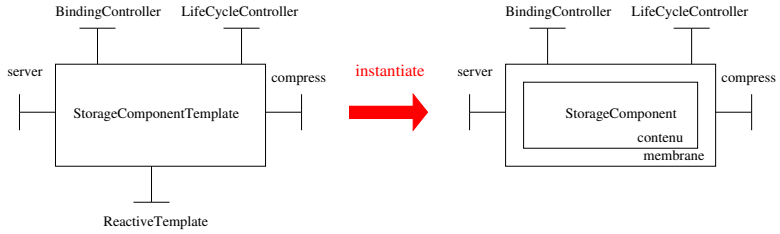


FIG. 5.5 – Template du composant *StorageComponent*

Le template *SchedulingConstraintTemplate* La figure 5.6 donne le diagramme de séquence illustrant la génération des automates d'une contrainte d'ordonnancement. Le template génère les automates de la contrainte avec la fabrique *SchedulingConstraintBehaviourFactory* qui est créée à partir de la fabrique *BootstrapFactory*.

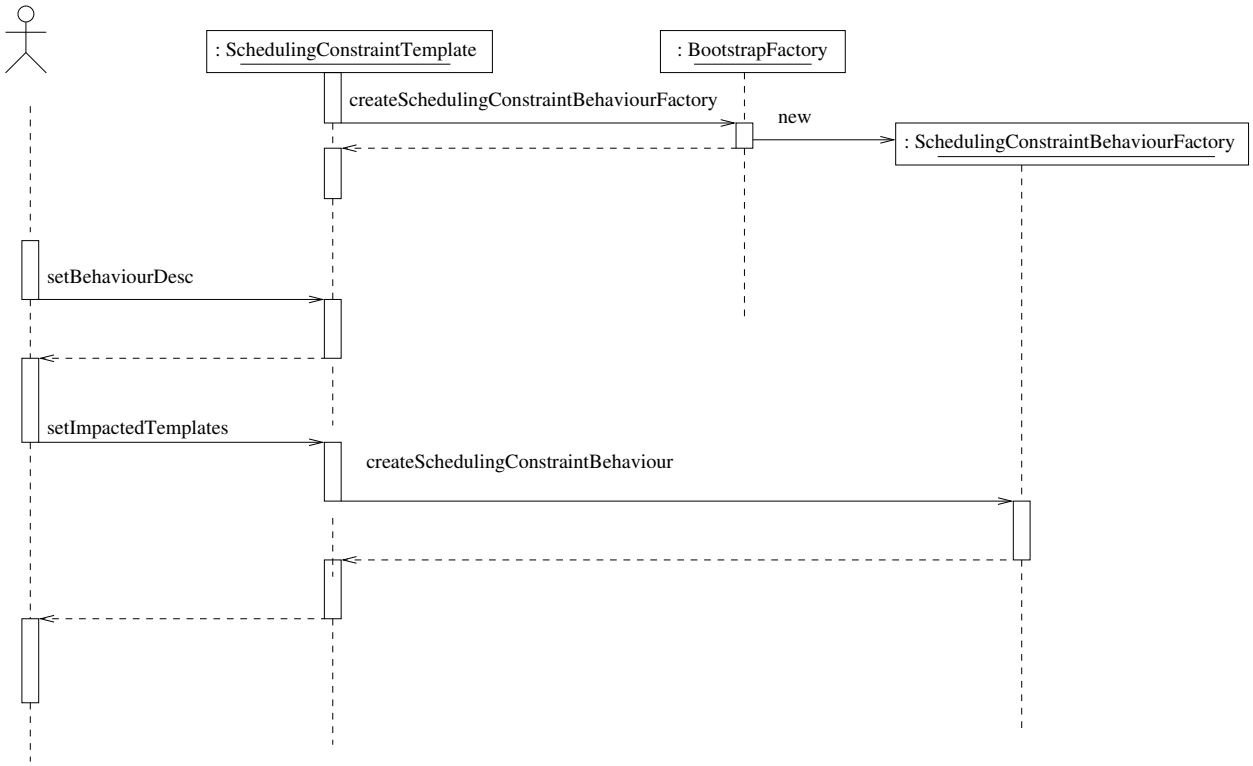


FIG. 5.6 – Génération des automates d'une contrainte d'ordonnancement

5.6.1.2 Les contrôleurs

Le contrôleur BindingController Ce contrôleur utilise la fabrique *BindingFactory* pour définir les échanges de signaux entre les interfaces des composants. Pour chaque signal défini sur les interfaces, une liaison est créée.

La figure 5.7 donne le diagramme de séquence présentant une création d'un binding entre interfaces.

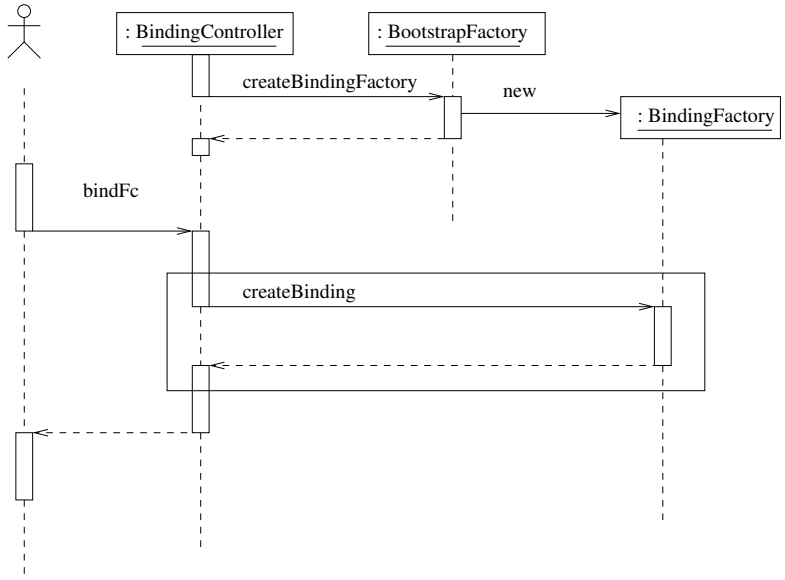


FIG. 5.7 – Création d'une liaison entre interfaces

Le contrôleur ContentController Ce contrôleur permet de contrôler le contenu du composite. L'implantation de la méthode *addFcSubComponent* de l'interface *ContentController* se sert donc de la méthode *addSubComponentBehaviour* de la fabrique *ComponentBehaviourFactory* pour ajouter le comportement des sous-composants au comportement du composite.

La figure 5.8 donne le diagramme de séquence présentant l'ajout d'un sous-composant au composite.

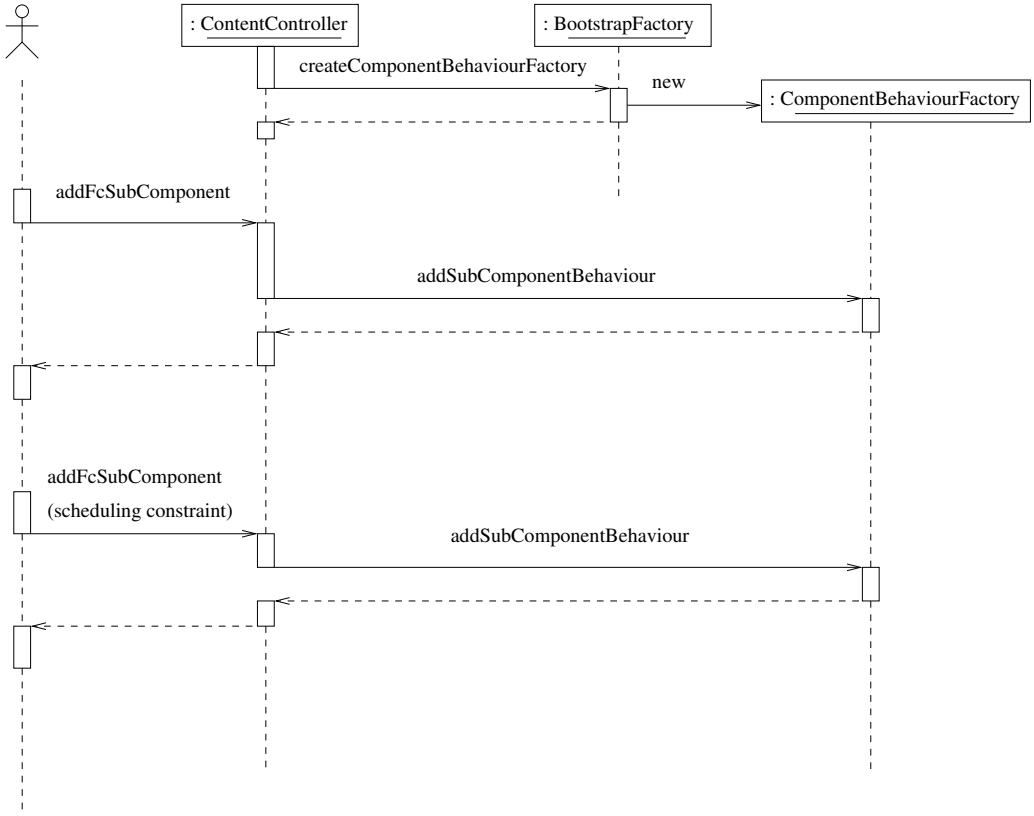


FIG. 5.8 – Ajout de composants

Le contrôleur LifeCycleController Ce contrôleur utilise la machine réactive générée. Le contrôleur utilise l'interface *MembraneCode* qui fournit la méthode *MC_start* qui permet d'initialiser la machine réactive. Le code de la membrane généré implante cette interface.

5.6.2 Implantation du canevas de description des automates

L'implantation du caneva de description des automates est constituée des implantations des fabriques *GenericBehaviourDescriptionFactory, PrimitiveComponentBehaviourDescriptionFactory, SchedulingConstraintBehaviourDescriptionFactory* et *StandardSchedulingConstraintBehaviourDescriptionFactory.*

5.7 Exemple

Le chapitre précédent 4 a montré les mécanismes de composition des composants (services techniques et contraintes d'ordonnancement). A l'exécution, l'ordonnancement est déterminé par la machine réactive générée par le framework. Le code de la machine réactive est donné dans la partie 7.3.

A partir de l'exemple présenté dans le chapitre précédent 4, les cinq composants sont composés au sein d'un composite qui définit la limite de la bulle réactive. La figure 5.9 donne la configuration des composants qui jouent un rôle dans l'entrelacement des actions des automates.

La figure 5.10 donne les automates des composants correspondant à la configuration de la figure 5.9. Tous les automates sont dans leur état initial (représentés par un cercle rouge).

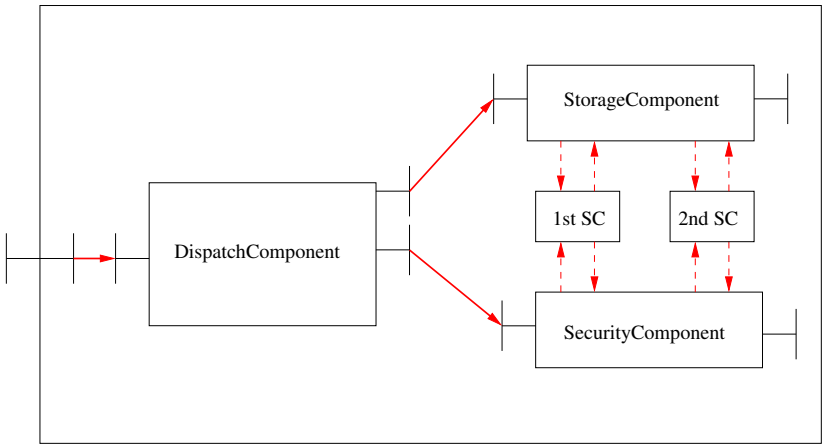


FIG. 5.9 – Configuration des composants

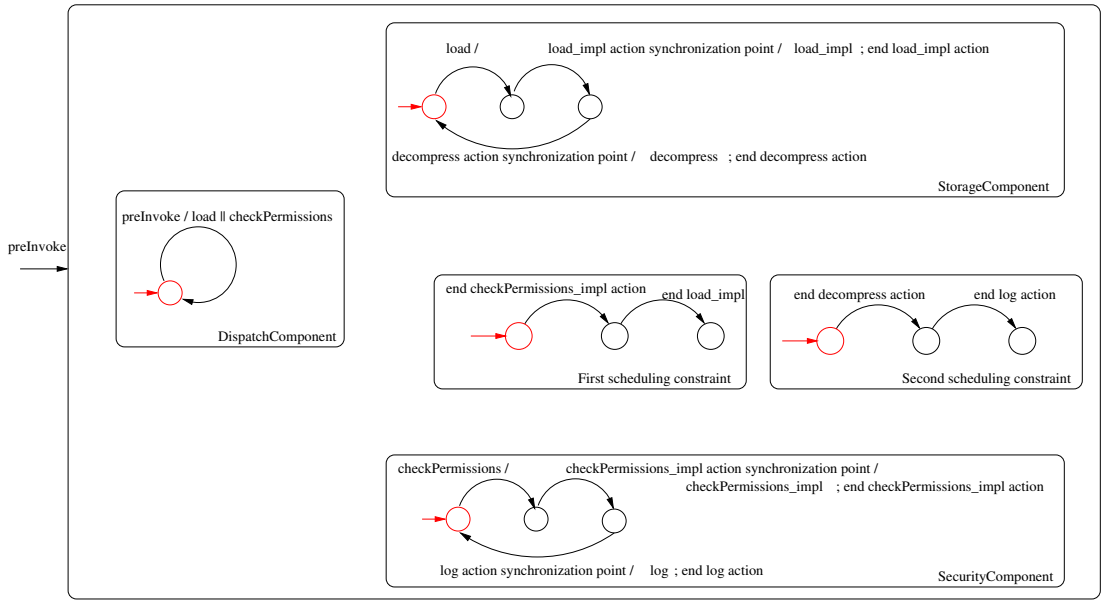


FIG. 5.10 – Initialisation des automates des composants

La figure 5.11 montre les interactions entre les composants réactifs qui implantent les services et les contraintes. Les interactions entre les composants qui implantent les services sont de type structurel. Les interactions entre les composants qui implantent les services et les contraintes d'ordonnancement sont de type comportemental.

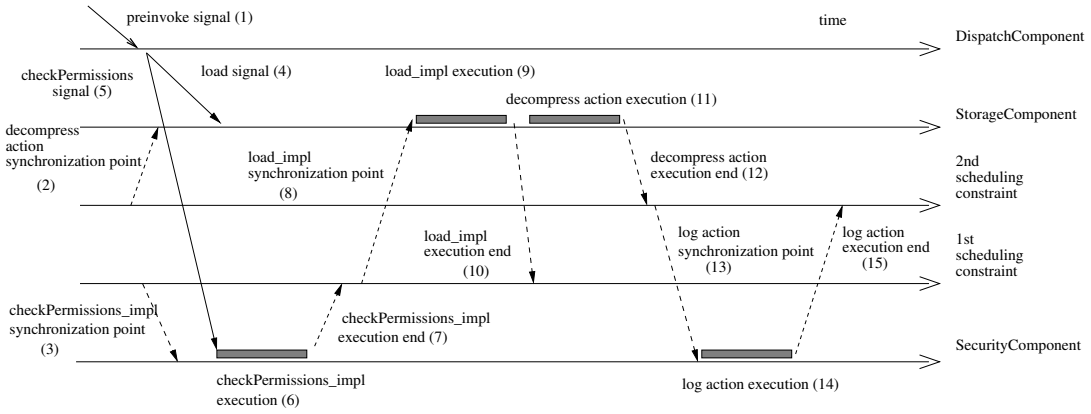


FIG. 5.11 – Chronogramme de l'exécution de l'exemple

L'initialisation du composite initialise la machine réactive. Cette initialisation prend en compte les automates des contraintes d'ordonnancement :

- l'automate de la première contrainte (l'opération *load_impl* ne peut pas être exécutée avant l'opération *checkPermissions_impl*) entre dans son état initial (la signification de cet état est que l'opération *checkPermissions_impl* peut s'exécuter).
- l'automate de la deuxième contrainte (l'action *log* ne peut pas être exécutée avant l'action *decompress*) entre dans son état initial (la signification de cet état est que l'opération *decompress* peut s'exécuter),

Après cette initialisation, l'automate de la première contrainte émet un signal (indiqué par le numéro 3 dans le chronogramme) correspondant à un point de synchronisation à l'automate du composant *SecurityComponent* pour lui indiquer que les conditions sont remplies en terme d'ordonnancement pour qu'il puisse exécuter l'opération *checkPermissions_impl*. L'automate de la deuxième contrainte émet un signal (indiqué par le numéro

2 dans le chronogramme) correspondant à un point de synchronisation à l'automate du composant *StorageComponent* pour lui indiquer que les conditions sont remplies en terme d'ordonnancement pour qu'il puisse exécuter l'action *decompress*. Ces émissions sont représentées dans la figure 5.12.

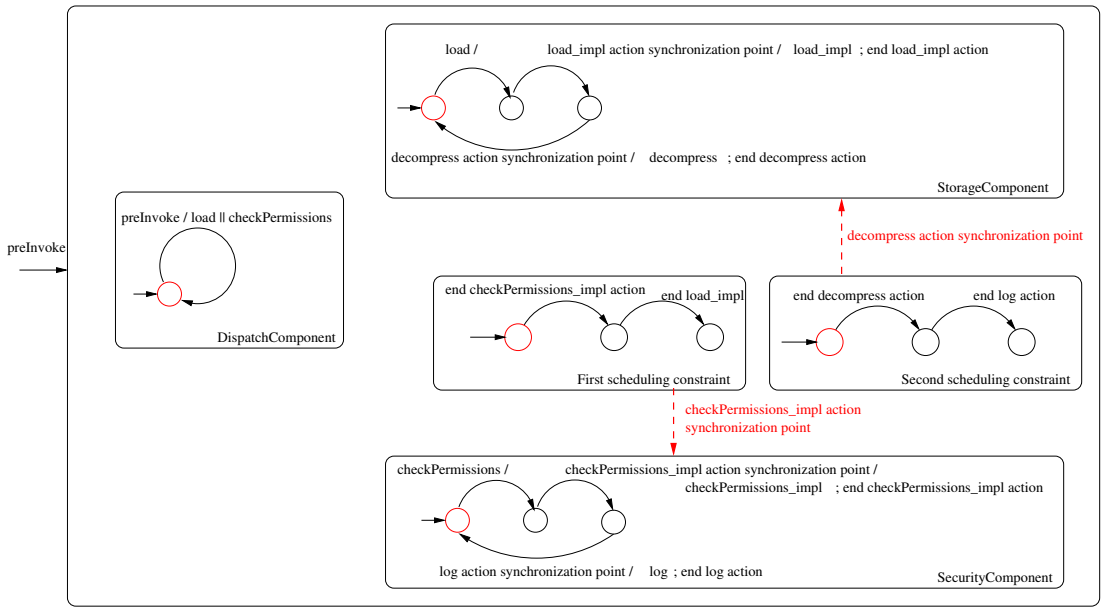


FIG. 5.12 – Emission des points de synchronisation

La méthode *preInvoke* est définie sur l'interface du composite. L'invocation de cette méthode émet un signal *preInvoke* sur la machine réactive. A la réception de ce signal (indiqué par le numéro 1 dans le chronogramme), l'automate du composant *DispatchComponent* émet en parallèle les signaux *load* (indiqué numéro 4 dans le chronogramme) et *checkPermissions* (indiqué par le numéro 5 dans le chronogramme). Cette mise en parallèle correspond au fait que les services sont indépendants (ils n'ont pas besoin des fonctionnalités de l'autre pour remplir leurs fonctions). La figure 5.13 représente cette invocation.

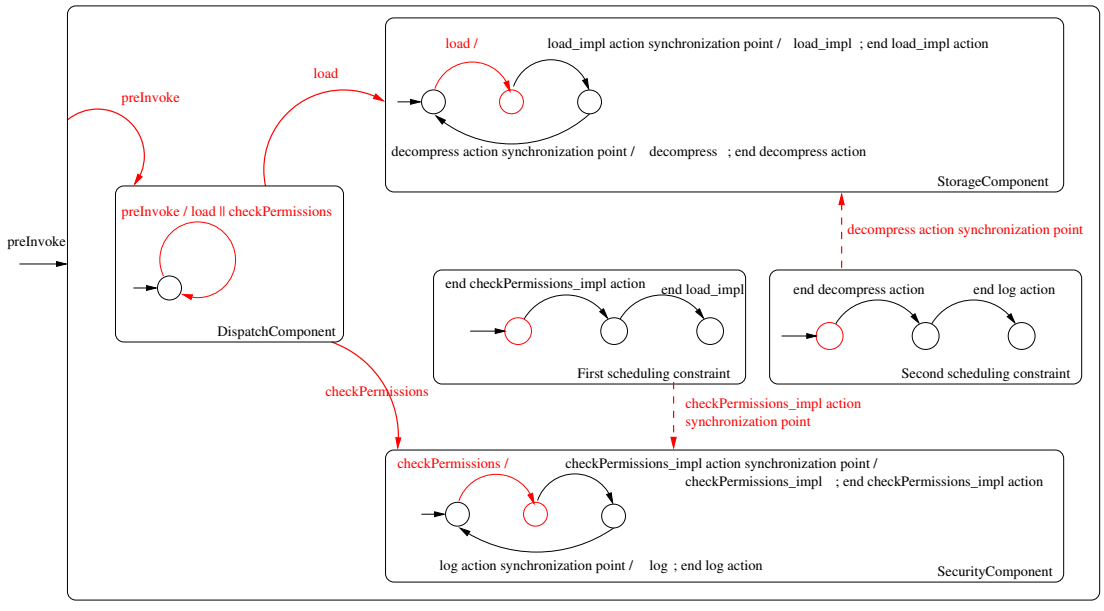


FIG. 5.13 – Invocation de la méthode *preInvoke*

A la réception du signal *checkPermissions*, l'automate du composant *SecurityComponent* exécute l'opération *checkPermissions_impl* (indiquée par le numéro 6 dans le chronogramme). A la réception du signal *load*, l'automate du composant *StorageComponent* attend le signal de synchronisation (indiqué par le numéro 8 dans le chronogramme) qui lui permettra d'exécuter l'opération *load_impl*. La figure 5.14 présente ces transitions.

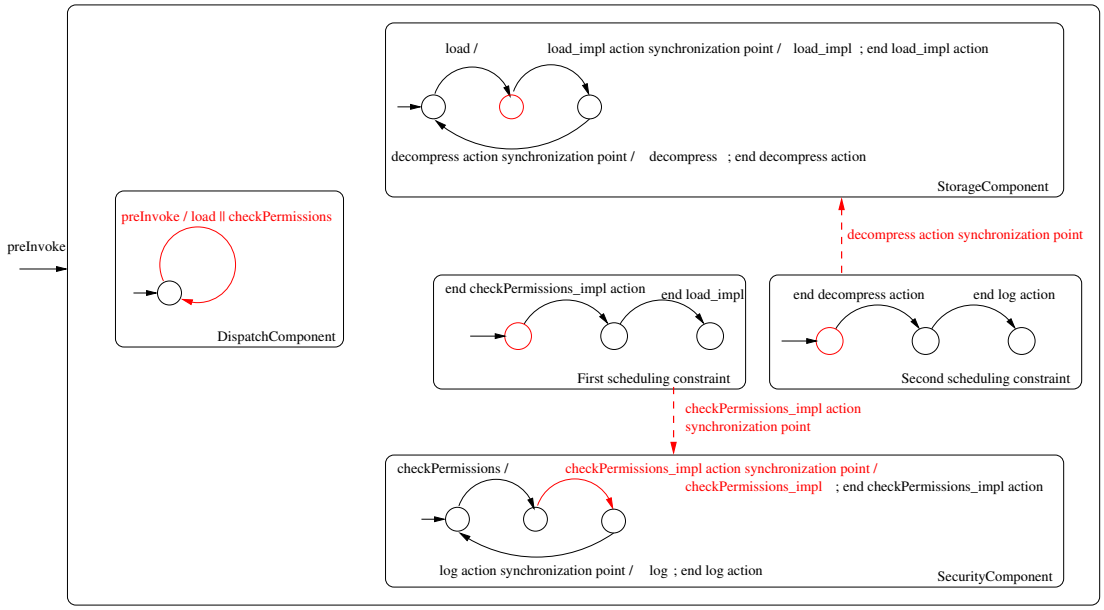


FIG. 5.14 – Exécution de l'opération *checkPermissions_impl*

L'automate du composant *SecurityComponent* émet un signal (indiqué par le numéro 7 dans le chronogramme) correspondant à la fin de l'exécution de l'opération *checkPermissions_impl* à l'automate de la première contrainte (cette dernière s'intéresse à cette exécution). A la réception du signal (indiqué par le numéro 7 dans le chronogramme) correspondant à la fin de l'exécution de l'opération *checkPermissions_impl*, l'automate de la première contrainte envoie un signal de synchronisation (indiqué par le numéro 8 dans le chronogramme) à l'automate du composant *StorageComponent* lui indiquant qu'il peut exécuter l'opération *load_impl* (l'opération *checkPermissions_impl* ayant été exécutée, l'ordre est respecté). A la réception de ce signal (indiqué par le numéro 8 dans le chronogramme), l'automate exécute l'opération *load_impl* (indiquée par le numéro 9 dans le chronogramme). La figure 5.15 présente ces transitions.

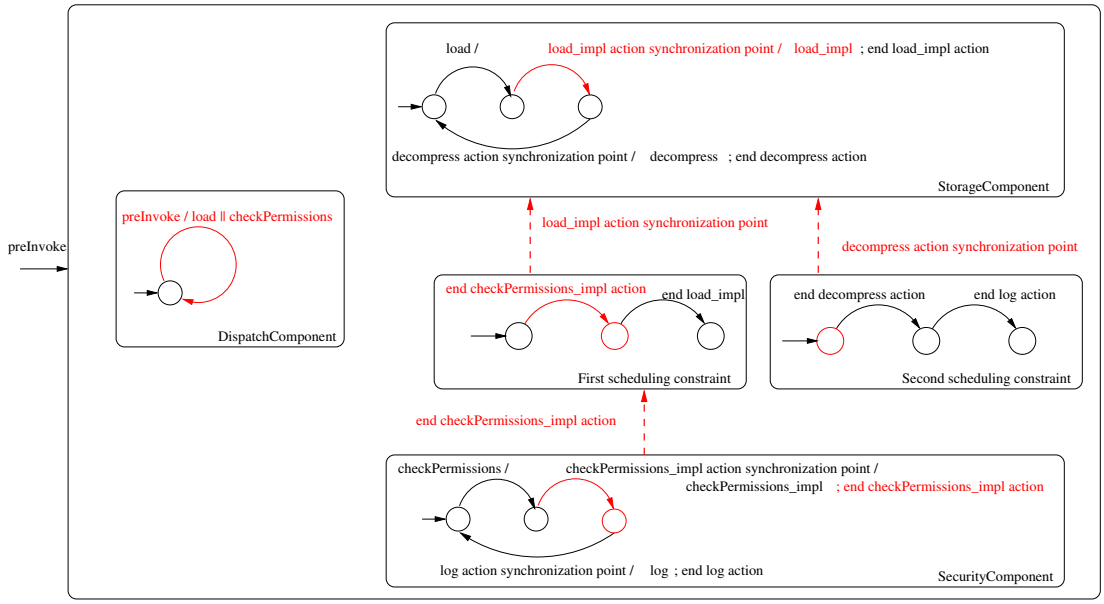


FIG. 5.15 – Exécution de l'opération *load_ impl*

A la fin de l'exécution de l'opération *load_ impl*, l'automate du composant *StorageComponent* émet le signal de fin d'exécution de cette opération (indiqué par le numéro 10 dans le chronogramme) à l'automate de la première contrainte. Il émet le signal *decompress* (indiqué par le numéro 11 dans le chronogramme) à l'automate du composant *CompressComponent*. L'automate du composant *CompressComponent* émet un signal correspondant à la fin de l'exécution des opérations liées à la réception du signal *decompress*. La figure 5.16 présente ces transitions.

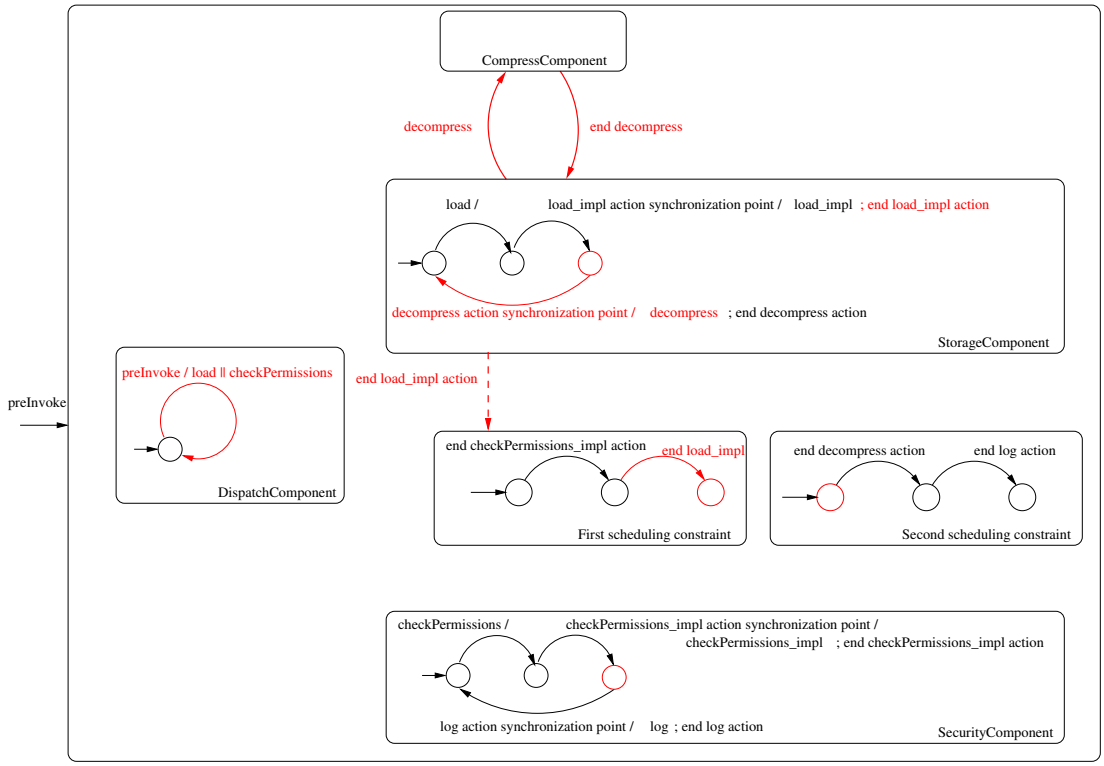


FIG. 5.16 – Exécution de l'opération *decompress*

L'automate du composant *StorageComponent* émet à l'automate de la deuxième contrainte

un signal (indiqué par le numéro 12 dans le chronogramme) correspondant à la fin de l'exécution de l'opération correspondant à l'émission du signal *decompress*. Il émet un signal de fin d'exécution des opérations liées à l'émission du signal *load* à destination de l'automate *DispatchComponent*.

A la réception de ce signal, l'automate de la deuxième contrainte émet un signal de synchronisation (indiqué par le numéro 13 dans le chronogramme) à l'automate du composant *SecurityComponent* lui indiquant qu'il peut exécuter l'opération correspondant à l'émission du signal *log*. La deuxième opération que doit exécuter l'automate du composant *SecurityComponent* est l'émission d'un signal *log* (indiqué par le numéro 14 dans le chronogramme) à l'automate du composant *LogComponent*. La première contrainte impose que cette opération ne peut pas s'exécuter avant l'opération *decompress* (d'où le point de synchronisation). L'automate du composant *SecurityComponent* peut exécuter l'opération d'émission du signal *log* à l'automate du composant *LogComponent*. Ce dernier émet alors un signal à destination de l'automate du composant *SecurityComponent* correspondant à la fin de l'exécution des opérations liées à la réception du signal *log*. L'automate du composant *SecurityComponent* émet un signal de fin d'exécution de l'opération d'émission du signal *log* (indiqué par le numéro 15 dans le chronogramme) à destination de l'automate de la deuxième contrainte. Il émet un signal de fin d'exécution des opérations liées à l'émission du signal *checkPermissions* à destination de l'automate du composant *DispatchComponent*. L'automate du composant *DispatchComponent* émet alors le signal correspondant à la fin d'exécution des opérations liées à l'émission des opérations liées à l'émission du signal *preInvoke*.

La figure 5.17 présente ces transitions.

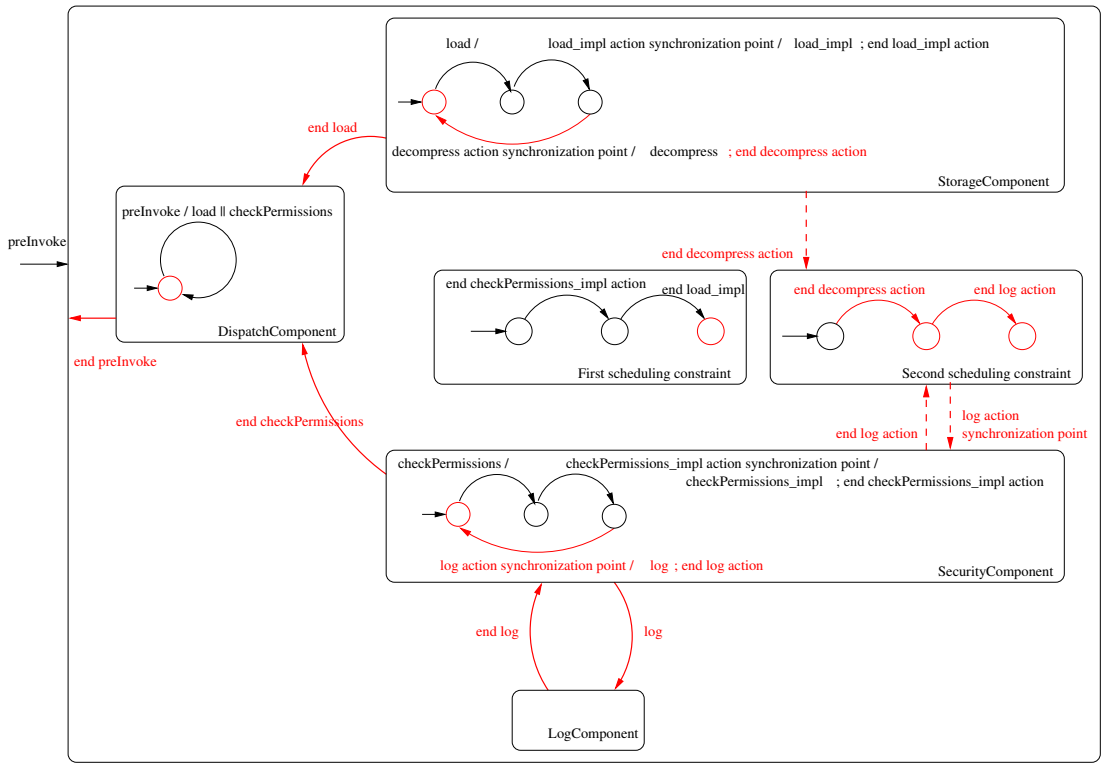


FIG. 5.17 – Exécution de l'opération *log*

5.8 Conclusion

Ce chapitre a détaillé l'implantation des canevas décrits dans le chapitre 4. Cette implantation repose sur différents générateurs qui interviennent lors des différentes phases de la conception et de l'exécution des composants. L'implantation actuelle se base sur la sémantique du langage synchrone réactif Esterel. L'implantation met en oeuvre la génération de machines réactives à partir de ce langage. Ces machines réactives fournissent un environnement d'exécution qui répond aux propriétés des modèles introduits dans l'approche de composition comportementale.

L'exécution de l'exemple a été détaillée et présente l'évolution des automates au cours de l'exécution.

Le chapitre suivant conclue sur l'approche de composition comportementale en évaluant cette dernière par rapport aux autres approches.

Chapitre 6

Evaluations et Conclusion

6.1 Rappel

Le chapitre 2 a présenté différentes approches de conception qui permettent de construire des infrastructures à base de services techniques. Le chapitre 3 a introduit une nouvelle approche de composition appelée approche de composition comportementale qui prend en compte un certain nombre de limites des autres approches. Les chapitre 4 et 5 ont détaillé l'implantation de cette approche.

Ce chapitre est destiné à évaluer l'approche de composition comportementale. Une première partie s'intéresse à évaluer l'approche de composition comportementale selon les critères définis dans la partie 2.5.1. La deuxième partie vise les points de flexibilité du canevas Brenda mis en oeuvre grâce à l'architecture de ce dernier. La troisième partie concerne les limites du travail effectué. La dernière partie s'oriente sur les perspectives de ce travail.

6.2 Evaluation

6.2.1 Evaluation de l'approche de composition comportementale

Cette partie vise à comparer l'approche de composition comportementale avec les autres approches selon les critères définis dans la partie 2.5.1.

6.2.1.1 Tableau d'évaluation

Le tableau 6.1 présente cette évaluation selon le modèle de conception, les mécanismes de composition et les techniques d'intégration.

Critères de comparaison	Approche de composition comportementale Canevas Brenda
Modèle de conception	Composants
Composition structurelle	
Nature de l'interaction	Appel de méthodes définies sur une interface de composant
Expression de l'interaction	Définition de liaisons sur les interfaces des composants
Implantation de l'interaction	Canal d'événements entre les automates des composants
Statique / dynamique	Statique et dynamique
Composition comportementale	
Nature de l'interaction	Relation d'ordre entre les opérations des automates
Expression de l'interaction	Définition de la relation d'ordre sous forme d'automates
Implantation de l'interaction	Contrainte d'ordonnancement et canal d'événements liés à la synchronisation
Statique / dynamique	Statique et dynamique
Techniques d'intégration	-

TAB. 6.1 – Evaluation de l'approche de composition comportementale

Selon le tableau 6.1, les services sont conçus comme des composants suivant le modèle Fractal.

Dans le cadre de la composition, les interactions de type structurel correspondent à l'exécution d'opérations définies sur les interfaces des composants. Elles s'expriment par la création d'une liaison entre les interfaces qui correspond à la création d'un canal d'événements entre les automates des composants. Les interactions de type comportemental correspondent à une relation d'ordre entre les opérations effectuées par les automates des composants et s'expriment sous la forme d'automates. Elles sont implantées avec des contraintes d'ordonnancement qui sont des composants et des canaux d'événements liés à la synchronisation entre les automates des contraintes et les automates des composants.

Le comportement défini à partir des relations d'ordre est non déterministe du fait que l'entrelacement des séquences d'exécution effectué par la machine réactive est déterminé à partir du comportement des composants.

En ce qui concerne la dynamicité, le canevas Brenda s'appuie sur un compilateur qui génère le comportement résultant de la composition des services : le canevas permet de générer dynamiquement du code Esterel, mais implique également de réinstantier la configuration à partir d'une nouvelle machine réactive (et donc une réinitialisation des automates). La composition dynamique est limitée par l'environnement d'exécution Esterel et non pas par les modèles de l'approche de composition comportementale.

6.2.1.2 Comparaison avec les autres approches

Les approches réflexives Il apparaît qu'il est difficile de comparer l'approche de composition comportementale avec les approches réflexives. Ces dernières sont axées sur le lien entre la partie fonctionnelle et la partie non fonctionnelle d'une application.

Les approches orientées aspects Le modèle de conception diffère (aspects pour AOP, composants pour l'approche de composition comportementale). En ce qui concerne la composition des services, l'approche de composition comportementale permet de définir les interactions de type structurel entre les services, ce que ne peuvent pas réaliser les approches orientées aspects. Elle permet également de définir les interactions de type comportemental avec un modèle d'expression des relations d'ordre plus étendu que ceux définis dans les implantations des approches orientées aspects. Le grain d'ordonnancement défini dans l'approche de composition comportementale est plus fin : le grain est l'action définie dans le modèle d'automate 3.3.2. L'entrelacement est déterminé lors de l'exécution des services alors que, dans les approches orientées aspects, l'entrelacement est déterminé indépendamment de l'exécution des services. L'approche de composition comportementale ne définit pas la manière dont les services sont intégrés au sein de l'application, contrairement aux approches orientées aspects qui introduisent un tisseur d'aspects responsable de leur intégration. Concernant la dynamicité, il est possible d'ajouter et de supprimer dynamiquement des composants au sein de la configuration, ainsi que les relations d'ordre (car une contrainte d'ordonnancement est également un composant). Seules EAOP et PROSE permettent de modifier dynamiquement la configuration d'aspects et donc les relations d'ordre entre eux.

L'approche interactions logicielles La différence entre l'approche de composition comportementale et l'approche interactions logicielles concerne l'expression de l'entrelacement. La fusion des interactions obéit à un ensemble de règles de fusion. Le résultat de la fusion correspond donc à l'entrelacement des interactions, la génération de cet entrelacement étant déterminé par les règles de fusion. L'approche de composition comportementale permet d'exprimer les relations d'ordre entre les actions des automates des composants, et donc l'entrelacement de l'exécution des composants est déterminé à partir des relations d'ordre et non pas à partir de règles de réécriture qui limitent la capacité d'entrelacer l'exécution des interactions. De plus, l'entrelacement est déterminé dynamiquement à partir de l'exécution des composants alors que l'entrelacement de l'exécution des interactions est statique (même si la fusion est réalisée dynamiquement).

Les approches orientées conteneurs Les approches orientées conteneur ne fournissent aucun mécanisme de composition. La conception des conteneurs ne permettent pas de rendre les conteneurs adaptables. L'approche de composition comportementale définit les mécanismes de composition qui permettent de définir les interactions entre les services. Le principal inconvénient de cette approche concerne les performances à l'exécution car l'ordonnancement est déterminé à l'exécution. Mais le surcoût est négligeable par rapport au coût d'exécution des services.

6.2.2 Les points de flexibilité du canevas

Cette partie présente les points de flexibilité de l'approche de composition comportementale.

L'approche de composition comportementale et du canevas qui implante cette approche ont été conçus afin que les modèles de composants, de comportement et d'expression des relations d'ordre soient orthogonaux pour introduire des points de flexibilité.

Le modèle de composants n'impose pas un modèle de comportement. Par contre, le modèle de comportement s'appuie sur des éléments du modèle de composants (le comportement décrit est celui des composants) : ces éléments sont les interfaces du composant. Le modèle de comportement n'impose pas un modèle d'expression des relations d'ordre.

Par contre, le modèle d'expression des relations d'ordre repose sur des éléments du modèle de comportement (pour les automates, ces éléments sont l'état et les actions).

La figure 6.1 présente le lien entre les modèles. La définition du modèle de comportement se base sur le modèle de composant, comme illustré dans la partie 3.3.2. La définition du modèle d'expression des relations d'ordre se base sur le modèle de comportement, comme illustré dans la partie 3.4.

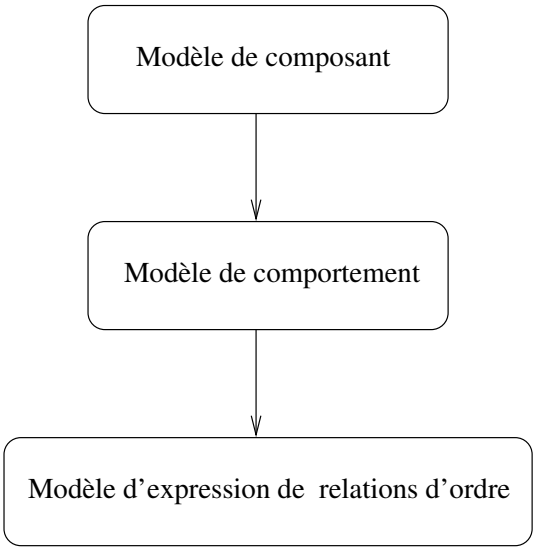


FIG. 6.1 – Lien entre les modèles

L'environnement d'exécution doit respecter la sémantique des différents modèles. Il est envisageable de mettre en oeuvre plusieurs environnements d'exécution pour ces modèles.

La suite détaille les points de flexibilité au sein des différents modèles.

6.2.2.1 Modèle de comportement

Le canevas de composants décrit dans la partie 4.2 utilise une classe qui implante la description du comportement du composant. Dans l'implantation actuelle, le modèle de comportement est un modèle d'automates. Un autre modèle de comportement peut être défini en implantant de nouvelles fabriques de description de comportement. Ainsi, la classe qui implante la description du comportement du composant utilisera ces nouvelles fabriques sans que le canevas de composants qui implante le modèle de composants ne soit modifié.

6.2.2.2 Modèle d'expression des relations d'ordre

Les contraintes d'ordonnancement définissent les relations d'ordre sous la forme de composants. Dans l'implantation actuelle, les relations d'ordre s'expriment à partir d'un modèle d'automates. Un autre modèle d'expression des relations d'ordre peut être défini en implantant une nouvelle fabrique de description. La relation d'ordre sera toujours implantée sous la forme d'un composant, mais la description de celle-ci s'effectuera avec la nouvelle fabrique de description. Ainsi, la classe qui implante la description de la relation d'ordre utilisera cette nouvelle fabrique sans que le canevas de composants qui implante le modèle de composants ne soit modifié.

6.2.2.3 Environnements d'exécution

Dans l'implantation actuelle, l'environnement d'exécution est l'environnement du langage Esterel. Il est possible d'implanter un nouveau processus d'instantiation *BehaviourInstantiationProcess* pour prendre en compte un nouvel environnement d'exécution. Ce processus peut utiliser de nouveaux éléments du générateur de machine réactive. L'implantation de ce nouveau processus ne modifie pas l'implantation du canevas de composants.

6.3 Les contributions et les limites

Cette section illustre les contributions et les limites de nos travaux.

6.3.1 Contributions

Les contributions sont multiples et chacune tend à atteindre les objectifs fixés au départ.

La première contribution concerne l'étude des mécanismes de composition des approches de conception d'infrastructures réparties (chapitre 2). Dans le cadre de cette étude, nous avons pu évaluer les approches et présenter leurs limites dans le domaine de la composition de services. Le résultat de cette étude montre que les interactions entre les services sont peu ou pas exprimées, ce qui limite la composition de ces derniers. Cette étude a également permis d'identifier deux types d'interactions : les interactions de type structurel et les interactions de type comportemental.

La deuxième contribution est la définition d'une approche de composition dont l'objectif est de palier les manques des approches de conception dans le domaine de la composition de services. L'objectif de cette approche est de pouvoir exprimer les interactions entre les services. Cette approche repose sur un ensemble de modèles (composant, comportement, expression de relations d'ordre). Elle met en oeuvre les interactions de type structurel et comportemental. Cette approche permet de concevoir des configurations de composants qui sont le résultat de la composition des services. Les interactions de type structurel permettent aux composants d'utiliser les fonctionnalités des autres composants de la configuration. Elles s'expriment par la création de liaisons entre les interfaces des composants. Les interactions de type comportemental permettent de déterminer l'entrelacement des séquences d'exécution des composants. Elles s'expriment par des relations d'ordre entre les opérations effectuées par les composants. Ces relations d'ordre s'expriment avec des automates, ce qui offre une expressivité supérieure aux autres approches qui mettent en oeuvre ce type d'interaction. Le grain d'ordonnancement est plus fin. Une relation d'ordre est définie sous forme de composant et donc bénéficie des propriétés du modèle. L'évaluation de notre approche (partie 6.2) montre qu'elle supprime un certain nombre de limites identifiées dans les autres approches. Un des apports fondamentaux concerne la prise en charge de la non orthogonalité. Dans les autres approches, la résolution de la non orthogonalité s'effectue en définissant un ordre d'exécution entre les entités. Dans notre approche, cet ordre s'exprime sur les opérations effectuées par les composants, qui détermine l'entrelacement des séquences d'exécution entre les composants. De plus, la relation d'ordre s'applique sur une action effectuée par l'automate d'un composant. Une relation d'ordre est définie sous la forme d'une contrainte d'ordonnancement : l'application d'une relation d'ordre revient à ajouter la contrainte dans la configuration de composants et donc permet de déterminer un nouvel entrelacement des séquences d'exécution des composants.

La troisième contribution est la réalisation d'un canevas logiciel qui implante les modèles définis dans l'approche de composition comportementale. Il peut être utilisé par les

autres approches. Il permet de reconfigurer dynamiquement les configurations de composants. Cette capacité de reconfiguration s'adresse à la fois aux composants qui implantent les services (dans l'exemple, le composant *CompressComponent* peut être remplacé par un autre composant qui implante un autre algorithme de décompression), mais aussi aux contraintes d'ordonnancement, ce qui signifie que l'entrelacement des séquences d'exécution peut ainsi être dynamiquement modifié. Dans l'implantation actuelle du canevas, la reconfiguration implique la génération d'une nouvelle machine réactive. Le canevas dispose d'une certaine flexibilité car il dispose de points de flexibilité qui permettent d'implanter de nouveaux modèles.

6.3.2 Limites

En ce qui concerne la première contribution (l'étude des mécanismes de composition), l'étude effectuée n'est pas exhautive. Elle recence les approches les plus utilisées pour concevoir les infrastructures.

Les limites de la deuxième contribution (la définition de l'approche) sont issues de la complexité inhérente à l'approche. En effet, décrire le comportement des composants sous forme d'automates demande un effort de modélisation. Le manque le plus important provient de la vérification lors de la composition comportementale. L'approche ne définit aucun mécanisme pour vérifier si les relations d'ordre sont cohérentes ou non. La partie 6.4 donne quelques pistes pour implanter ce mécanisme de vérification. L'approche vise à concevoir des configurations de composants. Une configuration est un composant composite. Dans notre approche, tous les composants qui ont des interactions de type comportemental doivent être dans un composant composite. Cela constitue une hypothèse forte quant à la composition des services. En ce qui concerne les modèles, les exceptions telles que définies dans le langage Java ne sont pas prises en compte.

La troisième contribution (la réalisation du canevas logiciel) demande un environnement logiciel conséquent (les canevas et l'environnement de compilation du langage Esterel). La difficulté est inhérente à la richesse des éléments des canevas.

6.4 Perspectives

Cette partie vise à donner quelques perspectives aux travaux présentés dans ce mémoire. Les perspectives portent sur l'approche de composition comportementale et sur le canevas.

6.4.1 Approche de composition comportementale

Formalisation de l'approche [15] propose de définir des algèbres de processus comme fondation pour la programmation orientée aspects. Il définit un aspect comme un processus. Le tissage d'aspects correspond à la transformation de programme en un processus principal construit à partir des processus combinés avec un opérateur de synchronisation.

[60, 58, 59] offre un cadre générique de programmation par aspects. Le langage formel utilisé pour décrire les aspects permet de définir une résolution concernant un conflit d'aspects (lorsque ces derniers sont non orthogonaux). Ces travaux ont permis de définir l'approche EAOP [3].

Le travail effectué au cours de cette thèse peut se rapprocher des travaux d'Andrews [15] car les automates introduits dans notre approche se basent sur les algèbres de processus [39, 78]. Le tissage d'aspects correspond donc dans notre approche à la composition des automates basée sur les contraintes d'ordonnancement. Il serait intéressant d'étendre

les travaux de J. Andrews [15] et de R. Douence [60, 58, 59] à partir de notre approche. Un certain nombre de propriétés pourraient être définies sur la composition.

Autres modèles de description de comportement / ordonnancement La propriété d'encapsulation du modèle Fractal et le point de flexibilité défini dans la partie 6.2.2.1 permettent de définir de nouveaux modèles de description de comportement sans modifier le canevas actuel.

La propriété d'encapsulation et le point de flexibilité défini dans la partie 6.2.2.2 permettent de définir de nouveaux modèles d'expression de relation d'ordre.

Vérification de comportements Un des points intéressants concerne la vérification du comportement, appelée *model-checking* [26, 49, 50]. Elle a pour objectif de prouver que les propriétés de sûreté (*safety property*) et de vivacité (*liveness property*) sont respectées [14].

Une perspective intéressante de notre travail serait de proposer à partir de nos modèles une technique de vérification du comportement de la configuration des composants. Une des techniques de vérification utilisée dans le model-checking est basée sur la notion d'*observer* [57]. Une des possibilités serait d'implanter la propriété (de sûreté ou de vivacité) à vérifier sous la forme d'un composant qui jouerait le rôle de *l'observer*.

Implantation des approches La partie 2.6 a montré que quelques approches servent à implanter les autres. Notre approche est une approche dédiée à la composition. La perspective serait d'utiliser notre approche au sein des autres approches, telles que les approches orientées aspects comme présenté dans la partie 4.5.2 ou les approches orientées conteneur.

6.4.2 Le canevas de composition comportementale

Les perspectives s'intéressent aux extensions données au canevas.

Description graphique des automates Les automates sont actuellement construits de manière programmatique. Une des extensions serait de définir un environnement graphique qui permettrait de concevoir ces automates. Cet environnement pourrait générer le code Java correspondant aux automates.

Autres modèles d'exécution Le point de flexibilité défini dans la partie 6.2.2.3 permet d'utiliser des environnements d'exécution autres qu'Esterel. Une des extensions du canevas serait de convertir les automates en programmes réactifs écrits dans le langage Junior[37]. Junior permet d'ajouter dynamiquement de nouveaux comportements réactifs.

6.5 Conclusion finale

La conception d'infrastructures réparties repose sur la composition des propriétés non fonctionnelles qui la constituent. L'objectif de notre travail a été de définir une approche de composition de services qui palie les manques des différentes approches de conception étudiées dans le chapitre 2. Pour atteindre cet objectif, une démarche a été appliquée. Elle est donnée dans le tableau 6.2.

Lors de la composition, les services interagissent entre eux. La nouvelle approche appelée approche de composition comportementale prend en compte deux types d'interactions : les interactions de type structurel et les interactions de type comportemental.

Cette approche introduit trois modèles (chapitre 3) :

1. un modèle de composants pour concevoir les services, le modèle choisi est le modèle Fractal,
2. un modèle de comportement, pour décrire le comportement des composants, le modèle choisi est un modèle d'automates,
3. un modèle d'expression de relations d'ordre, pour déterminer l'entrelacement des séquences d'exécution des composants. Ce modèle définit les contraintes d'ordonnancement conçues sous forme de composants et dont la sémantique de la relation d'ordre est donnée par l'automate de la contrainte.

Les mécanismes de composition mettent en oeuvre les interactions entre les composants.

Cette approche a permis un certain nombre d'avancées dans le domaine de la composition. Elle a notamment donné un apport siginificatif pour le traitement de la non orthogonalité qui se manifeste dans les approches orientées aspects. Elle a également mis en avant les limites des approches de conception d'infrastructures et donné des solutions pour supprimer ces limites.

Cette approche est implantée sous la forme d'un canevas logiciel appelé Brenda qui permet de concevoir les composants (il s'appuie sur le canevas Fractal), de décrire leur comportement sous forme d'automates (en utilisant des fabriques) et de les instantier (chapitre 4). Le canevas génère une machine réactive qui ordonne l'exécution des composants et permet donc de déterminer l'entrelacement des séquences d'exécution en fonction des relations d'ordre introduites avec les contraintes. Cette machine réactive est générée à partir de l'environnement Esterel qui est un langage synchrone réactif. Les automates sont traduits dans ce langage (chapitre 5).

L'approche et le canevas ont été conçus pour qu'ils puissent être réutilisés dans les autres approches (partie 4.5.2).

La vision générale de l'approche est donnée dans le tableau 6.3. La vision générale des canevas est donnée dans le tableau 6.4.

Démarche	Résultat
Problématique	Composition de services
Identification des limites des autres approches	Pas ou peu d'expressivité des interactions entre les services
Proposition d'une nouvelle approche de composition	Approche de composition comportementale
Implantation de l'approche	Canevas Brenda

TAB. 6.2 – Vision générale de la démarche

Approche de composition comportementale	Modèles	Implantations
Composant	Réutilisation du modèle Fractal	Réutilisation du canevas Fractal
Comportement	Définition d'un modèle d'automates	Canevas de description d'automates
Expression de relations d'ordre	Définition de contraintes d'ordonnancement	Réutilisation du canevas Fractal et du canevas de description d'automates

TAB. 6.3 – Vision générale de l'approche de composition comportementale

Canevas	Implantation des canevas
Canevas de composants	Implantation des contrôleurs (génération de machines réactives)
Canevas de description d'automates	Fabriques de description d'automates

TAB. 6.4 – Vision générale de l'implantation des canevas

L'illustration de cette approche s'appuie sur l'exemple donné dans la partie 1.3.1. Pour finir, un certain nombre de perspectives ont été esquissées.

Bibliographie

[1] Aopsys. See http ://www.aopsys.com/.

[2] AspectJ. See http ://www.aspectj.org/.

[3] Event-based Aspect Oriented Programming. See http ://www.emn.fr/x-info/eaop/.

[4] JBoss. See http ://www.jboss.org/.

[5] JOnAS. See http ://jonas.objectweb.org/.

[6] Le dictionnaire universel francophone en ligne. http ://www.francophonie.hachette-livre.fr.

[7] Microsoft .Net. See http ://www.microsoft.com/net/.

[8] ObjectWeb. The Fractal component model - http ://fractal.objectweb.org/.

[9] OpenJava. See http ://www.csg.is.titech.ac.jp/openjava/.

[10] The Arcad RNTL Project. See http ://arcad.essi.fr.

[11] The ObjectWeb Consortium. See http ://www.objectweb.org/.

[12] M. Aksit and B. Tekinerdogan. Aspect-Oriented Programming Using Composition Filters. In *Aspect Oriented Programming workshop at the European Conference on Object-Oriented Programming (ECOOP'98*, page 435. Springer-Verlag, July 1998.

[13] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition-Filters. In *ECOOP'93 Workshop on Object-Based Distributed Processing*, pages 152–184, 1993.

[14] G. R. Andrews. *Concurrent Programming - Principles and Practice.* The Benjamin Cummings Publishing Company Ltd., 1991.

[15] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of Lecture Notes in Computer Science, pages 187–209, Berlin, 2001.

[16] C. André, F. Boulanger, and A. Girault. Software Implementation of Synchronous Programs. In *IEEE Internationnal Conferennnce on Application of Concurrency to System Design*, 2001.

[17] A. Arnold. Systèmes de transitions finis et sémantique des processus communicants. *Techniques et Sciences de l'Informatique*, 1989.

[18] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* Masson, 1992.

[19] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[20] M. Bartorello, H. Maguin, M. Blay-Fornarino, A-M. Dery, and M. Riveill. Adjonction de services au sein d'un serveur EJB. In *Journées Composants 2001*, october 2001.

[21] L. Berger. Interactions et modèles de programmation : Support des interactions par les modèles à objets et à composants. *Journal l'Objet*, 2001.

[22] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle Micado.* PhD thesis, Université de Nice Sophia-Antipolis, 2001.

[23] L. Berger, A-M. Dery, and M. Fornarino. Interactions Between Objects : an Aspect of Object-Oriented Languages. In *ICSE'98 Workshop on Aspect-Oriented Programming*, April 1998.

[24] L. Bergmans. *The Composition-Filters Object Model.* PhD thesis, University of Twente, 1994.

[25] L. Bergmans and M. Aksit. Composing Multiple Concerns Using Composition Filters. In *Communications of the ACM*, October 2001.

[26] B. Bernard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *System and Software Verification : Model-checking Techniques and Tools.* Springer-Verlag, 2001.

[27] G. Berry. *The Esterel Language Primer.* Draft book , available at http ://www.inria.fr/meije/esterel, 1996.

[28] G. Berry. *The Constructive Semantics of Pure Esterel.* Draft book, available at http ://www.esterel.org, version 3, 1999.

[29] G. Berry. *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 2000.

[30] G. Berry and L. Cosserat. The synchronous programming languages Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[31] G. Berry, P. Couronné, and G. Gonthier. Synchronous Programming of Reactive Systems : an Introduction to Esterel. Technical Report 647, INRIA, 1987.

[32] G. Berry and G. Gonthier. The Esterel synchronous programming language : design , semantics, implementations. *Science of Computer Programming*, 19, 1992.

[33] M. Blay-Fornarino, A. Charif, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 2004.

[34] Mireille Blay-Fornarino, Anne-Marie Pinna-Dery, and Michel Riveill. Towards Dynamic Configuration of Distributed Applications. In *22nd International Conference on Distributed Computing Systems Workshops ICDCSW'02*, 2002.

[35] G. Boudol. Notes on algebraic calculi of process. In *Logic and Models of Concurrent Systems*, pages 261–303. Springer Verlag, 1985.

[36] N. Bouraqadi-Saâdani and T. Ledoux. Le point sur la programmation par aspects. In *Techniques et sciences informatiques, 20(4) :505-528.* 2001.

[37] F. Boussinot, L. Hazard, and J.-F. Susini. Programming with Junior. Draft - July 9, 2000.

[38] G. W. Brams. *Réseaux de Petri : théorie et pratique.* Masson, 1982.

[39] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of ACM 31*, 1984.

[40] A.W. Brown and K. Short. Components and Objects : The Foundations of Component based Development. In *5th International Symp. Assessment of Software Tools and Technologies*, 1997.

[41] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J-B. Stefani. An Open Component Model and Its Support in Java. In Pringer-Verlag, editor, *International*

Symposium on Component-based Software Engineering (ICSE2004-CBSE7, volume 3054 de Lecture Notes in Computer Science, pages 7–22, Edinburgh, Scotland, May 2004.

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

[43] E. Bruneton, T. Coupaye, and J-B. Stefani. The Fractal Component Model Specification 1.0. http ://fractal.objectweb.org/specification/index.html. Technical report, ObjectWeb, February 2004.

[44] B. Burke, M. Fleury, A. Brock, C. Hussenet, K. Khan, A. Oswald, M. Culpepper, and H. Khan. JBoss-Aspect Oriented Programming. http ://www.jboss.org/developers/projects/jboss/aop.

[45] S. Chiba. A metaobject protocol for C++. In *10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299, Austin, Texas, USA, October 1995.

[46] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[47] S. Chiba and M. Tatsubori. Yet Another java.lang.Class. In *ECOOP'98 workshop on Reflective object-oriented Programming and Systems*, July 1998.

[48] R. Chitchyan and I. Sommerville. Comparing Dynamic AO Systems. In *Dynamic Aspects Workshop 2004 (DAW04)*, pages 120–134, 2004.

[49] E. Clarke, J.O. Grumberg, and D.A. Peled. *Model-Checking*. The MIT Press, 1999.

[50] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Trans. Program. Lang. Syst.*, volume 697, 1986.

[51] R. Cleaveland, S. Smolka, and al. Strategic Directions in Concurrency Research. *ACM Computing Surveys*, 28(4), December 1996.

[52] D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel. Integration of Non-Functional Properties in Containers. In *WCOP 2001*, 2001.

[53] C. A. Contantinides and T. Elrad. Composing Concerns with a Framework Approach. In *International Workshop on Distributed Dynamic Multiservice Architectures, 21st International Conference on Distributed Computing Systems (ICDCS-21)*, volume 2, pages 133–140, April 2001.

[54] L. Cosserat. *Sémantique opérationnelle du langage synchrone Esterel.* PhD thesis, Université de Nice, 1985.

[55] T. Coupaye, R. Lenglet, M. Beauvois, E. Bruneton, and P. Déchamboux. Composants et composition dans l'architecture des systèmes répartis. In *ASF (ACM SIGOPS France) Journées Composants 2001 : flexibilité du système au langage (flexibility from systems to languages)*, Besançon, France, October 2001.

[56] L. G. DeMichiel, L. Ümit Yalçinalp, and S. Krishnan. Enterprise Java Beans Specification Version 2.0 Proposed Final Draft 2. Technical report, Sun Microsystems Inc., April 2001.

[57] M. Diaz, G. Juanole, and J.-P. Courtiat. Observer - A Concept for Formal On-Line Validation of Distributed Systems. In *IEEE Transactions on Software Engineering*, December 1994.

[58] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, 2002.

[59] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *4rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, March 2004.

[60] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection)*, volume 2192 of LNCS. Springer Verlag, 2001.

[61] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming : Introduction. In *Communications of the ACM*, volume 44, October 2001.

[62] W. Fokkink. *Introduction to Process Algebra.* Springer-Verlag, 2000.

[63] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[64] D. Garlan and D. Perry. Introduction to the special issue on software architecture. In *IEEE Transactions on Software Engineering 1995*, 1995.

[65] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, 1993.

[66] M. Golm. Design and Implementation of a Meta Architecture for Java. Master's thesis, University of Erlangen Nümberg - Germany, January 1997.

[67] M. Golm and J. Kleinöder. MetaJava : an efficient Run-Time Meta Architecture for Java. In *International Workshop on Object-Orientation in Operating Systems*, June 1996.

[68] M. Golm and J. Kleinöder. MetaJava - A Platform for Adaptable Operating-System Mechanisms. In *ECOOP'97 Workshop on Object-Orientation and Operating Systems (ECOOP'97)*, June 1997.

[69] M. Golm and J. Kleinöder. MetaXa and the Future of Reflection. In *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++*, October 1998.

[70] G. Gonthier. *Sémantique et modèles d'exécution des langages réactifs synchrones ; application à Esterel.* PhD thesis, Université d'Orsay, 1988.

[71] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[72] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[73] D. Harel. Statecharts : A visual formalism for complex systems. In *Science of Computer Programming*, 1987.

[74] D. Harel and E. Gery. Executable Object Modeling With Statecharts. In *IEEE Computer 30*, number 7. 1997.

[75] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure object). In *OOPSLA'93*, 1993.

[76] G. Heineman and W.T. Council. *Component-based Software Engineering.* Addison Wesley, 2001.

[77] M. Hennessy. *Algebraic theory of processes.* The MIT Press, 1988.

[78] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[79] C.A.R Hoare. *Concurrent Programming.* Addison-Wesley, 1990.

[80] W. L. Hürsch and C. V. Lopes. Separation of Concerns. Technical report, College of Computer Science, Northeastern University, Boston, Massachussetts, USA, February 1995.

[81] I. I. Ianov. The logical schemes of algorithms. In *Problems in Cybernetics*, volume 1, pages 75–127, 1960.

[82] ISO. ITU/ISO Reference Model of Open Distributed Processing - Part 2 : Foundations, International Standard ISO/IEC 10746-2, ITU-T Recommendation X.902. Technical report, ISO, 1995.

[83] ISO. ITU/ISO Reference Model of Open Distributed Processing - Part 3 : Foundations, International Standard ISO/IEC 10746-3, ITU-T Recommendation X.903. Technical report, ISO, 1995.

[84] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2) :23–35, June 1988.

[85] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[86] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP 2001*, volume 2072. Springer, 2001.

[87] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. In *Communications of the ACM*, volume 44. ACM Press, 2001.

[88] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Spinger, editor, *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of Lecture Notes in Computer Science, June 1997.

[89] J. Kienzle, Y. Yu, and J. Xiong. On Composition and Reuse of Aspects. In *2nd Foundations of Aspects-Oriented Languages Workshop at AOSD 2003*, Boston, March 2003.

[90] D. H. Lorenz. Visitor Beans ; An Aspect-Oriented Pattern. In *ECOOP'98 workshop on Aspect-Oriented Programming*, 1998.

[91] R. Marvie, P. Merle, J.-M. Geib, and M. Vadet. OpenCCM : une plate-forme ouverte pour composants CORBA. In *Seconde Conférence française sur les Systèmes d'Exploitation (CFSE'2)*, Paris, France, April 2001.

[92] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In *IEEE Transactions on Software Engineering*, volume 26, 2000.

[93] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-Oriented Programming. In *ECOOP'97 Workshop Reader*, pages 483–496. Springer-Verlag, 1997.

[94] Microsoft. The Component Object Model Specification. Technical report, Microsoft Corporation, 1995.

[95] R. Milner. A calculus of communicating systems. In *Lect. Notes Comput. Sci. 92*, 1980.

[96] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[97] A. Nicoara and G. Alonso. Dynamic AOP with Prose. In *International Workshop on Adaptative and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with CAISE'05*, June 2005.

[98] A. Nicoara and G. Alonso. PROSE - A middleware platform for dynamic adaptation. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, March 2005.

[99] N. Noda and T. Kishi. Implementing Design Patterns Using Advanced Separation of Concerns. In *OOPSLA 2001 Workshop on ASoC in OOS, 2001*, 2001.

[100] A. Oliva and L. E. Buzato. Composition of Meta-Objects in Guarana. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[101] A. Oliva and L. E. Buzato. The Design and Implementation of Guarana. In *5th USENIX Conference on Object-Oriented technologies and Systems (COOTS'99)*, 1999.

[102] OMG. The Common Object Request Broker : Architecture and Specification. Technical report, OMG, 1995.

[103] OMG. CORBA Components Version 3.0. Technical report, OMG, June 2002.

[104] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-Oriented Composition Rules. In *OOPSLA'95*, 1995.

[105] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying Subject-Oriented Composition. In *Theory and Practice of Object Systems*, 1996.

[106] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *ECOOP 2000*, 2000.

[107] R. Pawlak. *Interaction-based Aspect oriented Programming for the construction of multiple concern applications (La Programmation par Aspects interactionnelle pour la Construction d'Applications à Préoccupations Multiples).* PhD thesis, CNAM-Cedric, 2002.

[108] R. Pawlak, L. Duchien, and G. Florin. An automatic Aspect Weaver with reflective Programming language. In Springer, editor, *Meta-Level Architectures and Reflection, Reflection'99*, volume 1616, page 250, July 1999.

[109] R. Pawlak, L. Duchien, G. Florin, and L. Seinturier. JAC : Un Framework pour la Programmation Orientée Aspect en Java. *L'Objet*, 8, 2002.

[110] R. Pawlak, L. Duchien, L. Seinturier, G. Florin, F. Legond, and L. Martelli. *JAC : A Framework for Separation of Concerns and Distribution.* Addison-Wesley, 2004.

[111] R. Pawlak, J.P. Rétaille, and L. Seinturier. *La Programmation Orientée Aspect.* Eyrolles, 2004.

[112] R. Pawlak and L. Seinturier. Dynamic and Distributed Aspect-Oriented Programming with JAC. In *Tutorial JAC at AOSD 2003*, March 2003.

[113] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A Flexible Framework for AOP in Java. In *Reflection'01*, volume LNCS 2192, pages 1–24, 2001.

[114] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A Flexible Solution for Aspect-Oriented Programming in Java. In Springer, editor, *3rd International Conference on Metalevel Architectures and Separation of Crosscuting Concerns (Reflection 2001)*, volume 2192 of Lecture Notres in Computer Science, pages 1–24, Kyoto, Japan, September 2001.

[115] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC : an Aspect-Based Distributed Dynamic Framework. *Software : Practice and Experience*, October 2004.

[116] N. Pessemier, L. Seinturier, and L. Duchien. Components, ADL and AOP : Towards a Common Approach. In *Workshop ECOOP Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04)*, June 2004.

[117] N. Pessemier, L. Seinturier, L. Duchien, and O. Barais. Une extension de Fractal pour l'AOP. In *Première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2004)*, September 2004.

[118] J.L. Peterson. Petri Nets. In *ACM Computing Survey*, volume 9. September 1977.

[119] A. Popovi, T. Gross, and G. Alonso. Dynamic Homogenous AOP wih Prose. Technical report, Department of Computer Science, ETH Zürich, Zürich, Switzerland, March 2001.

[120] A. Popovici, G. Alonso, and T. Gross. Just in Time Aspects : Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development*, 2003.

[121] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development*, 2002.

[122] W. Reisig. Petri Nets : an Introduction. In *EACTS Monographs on Theorical Computer Science*, 1985.

[123] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1998.

[124] J. D. Rutledge. On Ianov's program schemes. *J. Assoc. Comput. Mach*, 1964.

[125] J.M. Sobel and D.P. Friedman. An Introduction to Reflection-Oriented Programming. In *Reflection'96*, 1996.

[126] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. In *Communication of the ACM.* October 2001.

[127] BEA Systems and al. CORBA Component Model Joint Revised Submission. Technical report, Object Management Group, July 1999. OMG Document orbos/99-07-01.

[128] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 2nd edition edition, 2002.

[129] C. Szyperski and C. Pfister. Why objects are not enough. In *International Component Users Conference*, 1996.

[130] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual.* http ://www.research.ibm.com/hyperspace, 2000.

[131] P. Tarr and H. Ossher. Hyper/J : multi-dimensional separation of concerns for Java. In *23rd International Conference on Software Engineering*, 2001.

[132] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N Degrees of Separation : Multi-Dimensional Separation of Concerns. In *ICSE 1999*, 1999.

[133] M. Tatsubori. An Extension Mechanism for the Java Language. Master's thesis, University of Tsukuba, 1999.

[134] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Otano. OpenJava : A Class-Based Macro System for Java. In W. Cazzola, R. J. Stroud, and F. Tisato, editors, *Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[135] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient Compilation of ESTEREL for Real-Time Embedded Systems. In *CASES'00*, November 2000.

[136] J. Wichman. ComposeJ : The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Master's thesis, University of Twente, 1999.

Chapitre 7

Annexe

Les annexes sont constituées :

1. des implantations de l'exemple selon les différentes approches étudiées dans le chapitre 2,
2. du détail de la fabrique de description d'automates standards de contraintes d'ordonnancement appelée *StandardSchedulingConstraintBehaviourDescriptionFactory,*
3. de la génération de la machine réactive correspondant à l'exemple (partie 5.7) décomposée en deux parties :
   (a) la première partie illustre la conversion des automates des composants de l'exemple en donnant le code Esterel généré,
   (b) la deuxième partie présente le code généré de la membrane du composite.

7.1 Implantation de l'exemple avec les différentes approches

Cette partie est constituée des implantations de l'exemple donné dans la partie 1.3.1 avec AspectJ, JAC, PROSE, EAOP, ISL, JBoss-AOP.

7.1.1 Implantation en AspectJ

Les algorithmes 39, 40 et 41 donnent l'implantation des gestionnaires avec AspectJ.

Algorithm 39 Gestionnaire de stockage en AspectJ

```
aspect StorageAspect {
  pointcut businessmethodcalls (BusinessObject bo) :
    target (bo) && call(public * * (..));
  before (BusinessObject bo) : businessmethodcalls(bo) {
      if (!bo.isLoaded) load(bo);
  }
  private void load(BusinessObject bo) {
    ...
    decompress(...);
  }
  private void decompress(...) {...}
}
```

Algorithm 40 Gestionnaire de sécurité en AspectJ

```
aspect SecurityAspect {
  before (BusinessObject bo) : businessmethodcalls(bo) {
    ...
    checkPermissions(bo) ;
    ...
  }
  private void checkPermissions (BusinessObject bo)
    throws SecurityException {
    ...
    log.notify("Access to object owned by "+bo.getOwner()) ;
  }
}
```

Algorithm 41 Gestionnaire de cryptage en AspectJ

```
aspect EncryptionAspect {
  before (BusinessObject bo) : businessmethodcalls(bo) {
    cryptAndSend(bo) ;}
  private void cryptAndSend(BusinessObject bo) {...}
}
```

7.1.2 Implantation en JAC

Les algorithmes 42, 43 et 44 donnent l'implantation des gestionnaires avec JAC.

Algorithm 42 Gestionnaire de stockage en JAC

```
public class StorageAC extends AspectComponent
{
  StorageAC() {
    pointcut(".*",".*",".*", new StorageWrapper(this), "businessmethodcalls") ;
  }
  public class StorageWrapper extends Wrapper {
    public void businessmethodcalls (Interaction interaction) {
      if (!((BusinessObject) interaction.wrapper).isLoaded)
         load((BusinessObject) interaction.wrapper) ;
      proceed(interaction) ;
      ...
    }
  }
}
```

Algorithm 43 Gestionnaire de sécurité en JAC

```
public class SecurityAC extends AspectComponent {
  SecurityAC() {
    pointcut(".*",".*",".*", new SecurityWrapper(this), "businessmethodcalls") ;
  }
  public class SecurityWrapper extends Wrapper {
    public void businessmethodcalls (Interaction interaction) {
     ...
      checkPermissions ((BusinessObject) interaction.wrapper) ;
     ...
      proceed(interaction) ;
     ...
    }
    private void checkPermissions (BusinessObject bo)
        throws SecurityException {
      ...
      log.notify("Access to object owned by "+bo.getOwner()) ;
    }
  }
}
```

Algorithm 44 Gestionnaire de cryptage en JAC

```
public class EncryptionAC extends AspectComponent {
  EncryptionAC() {
    pointcut (“.*”,”.*”,”.*”, new EncryptionWrapper(this), “businessmethodcalls”) ;
  }
  public class EncryptionWrapper extends Wrapper {
    public void businessmethodcalls (Interaction interaction) {
      cryptAndSend ((BusinessObject) wrapper) ;
      proceed(interaction) ;
      ...
    }
  }
}
```

7.1.3 Implantation en PROSE

Les algorithmes 45, 46 et 47 donnent l'implantation des gestionnaires avec PROSE.

Algorithm 45 Gestionnaire de stockage en PROSE

```
public class StorageCrosscut extends MethodCut {
  protected PointCutter pointCutter() {
    return Executions.before() ;
  }
  public void METHOD_ARGS(BusinessObject bo, ...) {
    if ( !(bo.isLoaded))
         load(bo) ;
    ...
  }
}
public class StorageAspect extends Aspect {
  Crosscut c1 = new StorageCrosscut() ;
  protected Crosscut[] crosscuts() {
    return new Crosscut [] {c1};
  }
}
```

Algorithm 46 Gestionnaire de sécurité en PROSE

```
public class SecurityCrosscut extends MethodCut {
  protected PointCutter pointCutter() {
    return Executions.before() ;
  }
  public void METHOD_ARGS(BusinessObject bo, ...) {
    ...
    checkPermissions(bo) ;
    ...
  }
  private void checkPermissions (BusinessObject bo)
        throws SecurityException {
    ...
    log.notify(“Access to object owned by “+bo.getOwner()) ;
   }
}
public class SecurityAspect extends Aspect {
  Crosscut c1 = new SecurityCrosscut() ;
  protected Crosscut[] crosscuts() {
    return new Crosscut [] {c1};
  }
}
```

Algorithm 47 Gestionnaire de cryptage en PROSE

```
public class EncryptionCrosscut extends MethodCut {
  protected PointCutter pointCutter() {
    return Executions.before() ;
  }
  public void METHOD_ARGS(BusinessObject bo, ...) {
    cryptAndSend(bo) ;
    ...
  }
}
public class EncryptionAspect extends Aspect {
  Crosscut c1 = new EncryptionCrosscut() ;
  protected Crosscut[] crosscuts() {
    return new Crosscut [] {c1};
  }
}
```

7.1.4 Implantation en EAOP

Les algorithmes 48, 49 et 50 donnent l'implantation des gestionnaires en EAOP.

Algorithm 48 Gestionnaire de stockage en EAOP

```
public class StorageAspect extends Aspect {
  public void definition() {
    ...
    MethodCall mc = nextBusinessObjectMethodCallEvent() ;
    BusinessObject bo = (BusinessObject) (mc.receiver) ;
    if (!bo.isLoaded) load(bo) ;
    ...
  }
  MethodCall nextBusinessObjectMethodCallEvent() {
    boolean ok = false ; MethodCall mc = null ; ...
    while (!ok) {
      e = nextEvent() ;
      ok = (e instanceof MethodCall) ;
      mc = (MethodCall) e ;
    }
    return mc ;
  }
}
```

Algorithm 49 Gestionnaire de sécurité en EAOP

```
public class SecurityAspect extends Aspect {
  public void definition() {
    ...
    MethodCall mc = nextBusinessObjectMethodCallEvent() ;
    BusinessObject bo = (BusinessObject) (mc.receiver) ;
    checkPermissions(bo) ;
    ...
  }
  private void checkPermissions (BusinessObject bo)
        throws SecurityException {
      ...
      log.notify("Access to object owned by "+bo.getOwner()) ;
  }
  MethodCall nextBusinessObjectMethodCallEvent() {
    boolean ok = false ; MethodCall mc = null ; ...
    while (!ok) {
      e = nextEvent() ;
      ok = (e instanceof MethodCall) ;
      mc = (MethodCall) e ;
    }
    return mc ;
  }
}
```

Algorithm 50 Gestionnaire de cryptage en EAOP

```
public class EncryptionAspect extends Aspect {
  public void definition() {
    ...
    MethodCall mc = nextBusinessObjectMethodCallEvent() ;
    BusinessObject bo = (BusinessObject) (mc.receiver) ;
    cryptAndSend(bo) ;
    ...
  }
  MethodCall nextBusinessObjectMethodCallEvent() {
    boolean ok = false ; MethodCall mc = null ; ...
    while (!ok) {
      e = nextEvent() ;
      ok = (e instanceof MethodCall) ;
      mc = (MethodCall) e ;
    }
    return mc ;
  }
```

7.1.5 Implantation en ISL

Les algorithmes 51, 52 et 53 donnent l'implantation des interactions avec les gestionnaires en ISL.

Algorithm 51 Gestionnaire de stockage en ISL

```
interaction StorageInteraction (BusinessObject bo, StorageManager sm) {
  bo.* -> if (!bo.isLoaded) {
            sm.load(bo) ;
          };
          bo._call
}
```

Algorithm 52 Gestionnaire de sécurité en ISL

```
interaction SecurityInteraction (BusinessObject bo, SecurityManager sm) {
  bo.* -> try {
            sm.checkPermissions(bo) ;
          } catch ...
          bo._call
}
```

Algorithm 53 Gestionnaire de cryptage en ISL

```
interaction EncryptionInteraction (BusinessObject bo, EncryptionManager em) {
  bo.* -> em.cryptAndSend(bo) ;
          bo._call
}
```

7.1.6 Implantation en JBoss-AOP

Les algorithmes 54, 55 et 56 donnent l'implantation des gestionnaires en JBoss-AOP.

Algorithm 54 Gestionnaire de stockage en JBoss-AOP

```
public class StorageInterceptor implements Interceptor {
...
  public String getName() {
    return StorageInterceptor ;
  }
  public Object invoke (Invocation invocation)  throws Throwable {
    BusinessObject bo = (BusinessObject) invocation.getTargetObject() ;
    if (!bo.isLoaded) storageManager.load(bo) ;
    return invocation.invokeNext() ;
  }
}
```

Algorithm 55 Gestionnaire de sécurité en JBoss-AOP

```
public class SecurityInterceptor implements Interceptor {
...
  public String getName() {
    return SecurityInterceptor ;
  }
  public Object invoke (Invocation invocation)  throws Throwable {
    BusinessObject bo = (BusinessObject) invocation.getTargetObject() ;
    securityManager.checkPermissions(bo) ;
    ...
    return invocation.invokeNext() ;
  }
}
```

Algorithm 56 Gestionnaire de cryptage en JBoss-AOP

```
public class EncryptionInterceptor implements Interceptor {
...
  public String getName() {
    return EncryptionInterceptor ;
  }
  public Object invoke (Invocation invocation) throws Throwable {
    BusinessObject bo = (BusinessObject) invocation.getTargetObject() ;
    encryptionManager.cryptAndSend(bo) ;
    ...
    return invocation.invokeNext() ;
  }
}
```

7.2 La fabrique *StandardSchedulingConstraintBehaviourDescriptionFactory*

Les méthodes sont :

1. *createEventsSchedulingConstraint* : pour créer les contraintes à partir des événements définis sur les automates. En remarque, les contraintes exprimées sur les événements portent sur les actions effectuées en réaction à ces événements. Les contraintes peuvent s'appliquer sur l'instance, le set et le template : une contrainte appliquée à partir d'un template ou d'un set s'appliquera sur l'ensemble des instances en rapport avec respectivement le template ou le set.
   (a) evt1 $<$ evt2 (evt1, evt2 : instance d'événement) : la signification de cette contrainte est que toute action réagissant à l'instance d'événement evt2 (située sur une transition de l'automate du composant) ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,
   (b) Tevt1 $<$ evt2 (Tevt1 : template d'événement, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1,
   (c) Sevt1 $<$ evt2 (Sevt1 : set d'événement, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,
   (d) evt1 $<$ Tevt2 (evt1 : instance d'événement, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

(e) evt1 < Sevt2 (evt1 : instance d'événement, Sevt2 : set d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

(f) Sevt1 < Sevt2 (Sevt1, Sevt2 : sets d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

(g) Sevt1 < Tevt2 (Sevt1 : set d'événement, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

(h) Tevt1 < Sevt2 (Tevt1 : template d'événement, Sevt2 : set d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1,

(i) Tevt1 < Tevt2 (Tevt1, Tevt2 : templates d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1.

2. *createEventStateSchedulingConstraint*

   (a) evt1 < st2 (evt1 : instance d'événement, st2 : état d'automate) : l'automate ne pourra passer dans l'état st2 qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

3. *createEventsStateSchedulingConstraint*

   (a) Sevt1 < st2 (Sevt1 : set d'événement, st2 : état d'automate) : l'automate ne pourra passer dans l'état st2 qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

   (b) Tevt1 < st2 (Tevt1 : template d'événement, st2 : état d'automate) : l'automate ne pourra passer dans l'état st2 qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1.

4. *createEventActionSchedulingConstraint*

   (a) evt1 < act2 (evt1 : instance d'événement, act2 : instance d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

5. *createEventActionsSchedulingConstraint*

   (a) evt1 < Sact2 (evt1 : instance d'événement, Sact2 : set d'action) : toute action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

   (b) evt1 < Tact2 (evt1 : instance d'événement, Tact2 : template d'action) : toute action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à l'instance d'événement evt1,

6. *createEventsActionSchedulingConstraint*

   (a) Tevt1 < act2 (Tevt1 : template d'événement, act2 : instance d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1,

(b) Sevt1 < act2 (Sevt1 : set d'événement, act2 : instance d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

7. *createEventsActionsSchedulingConstraint*

(a) Tevt1 < Tact2 (Tevt1 : template d'événement, Tact2 : template d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1,

(b) Tevt1 < Sact2 (Tevt1 : template d'événement, Sact2 : set d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du template Tevt1,

(c) Sevt1 < Tact2 (Sevt1 : set d'événement, Tact2 : template d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

(d) Sevt1 < Sact2 (Sevt1 : set d'événement, Sact2 : set d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de l'action réagissant à n'importe quelle instance d'événement du set Sevt1,

8. *createStatesSchedulingConstraint*

(a) st1 < st2 (st1, st2 : états d'automate) : l'automate ne pourra passer à l'état st2 qu'à partir du moment où l'autre automate sera passé à l'état st1,

9. *createStateEventSchedulingConstraint*

(a) st1 < evt2 (st1 : état d'automate, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 ne pourra être exécutée qu'à partir du moment où l'automate sera passé à l'état st1,

10. *createStateEventsSchedulingConstraint*

(a) st1 < Sevt2 (st1 : état d'automate, Sevt2 : instance d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'à partir du moment où l'automate sera passé à l'état st1,

(b) st1 < Tevt2 (st1 : état d'automate, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'à partir du moment où l'automate sera passé à l'état st1,

11. *createStateActionSchedulingConstraint*

(a) st1 < act1 (st1 : état d'automate, act1 : instance d'action) : l'instance d'action act1 ne pourra être exécutée qu'à partir du moment l'automate sera passé à l'état st1,

12. *createStateActionsSchedulingConstraint*

(a) st1 < Tact2 (st1 : état d'automate, Tact2 : template d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'à partir du moment où l'automate sera passé à l'état st1,

(b) st1 < Sact2 (st1 : état d'automate, Sact2 : set d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'à partir du moment où l'automate sera passé à l'état st1,

13. *createActionsSchedulingConstraint*

(a) act1 < act2 (act1, act2 : instances d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

(b) act1 < Sact2 (act1 : instance d'action, Sact2 : set d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

(c) act1 < Tact2 (act1 : instance d'action, Tact2 : template d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

(d) Sact1 < act2 (Sact1 : set d'action, act2 : instance d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(e) Sact1 < Sact2 (Sact1, Sact2 : sets d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(f) Sact1 < Tact2 (Sact1 : set d'action, Tact2 : template d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(g) Tact1 < act2 (Tact1 : template d'action, act2 : instance d'action) : l'instance d'action act2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

(h) Tact1 < Sact2 (Tact1 : template d'action, Sact2 : set d'action) : toute instance d'action du set Sact2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

(i) Tact1 < Tact2 (Tact1, Tact2 : templates d'action) : toute instance d'action du template Tact2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

14. *createActionEventSchedulingConstraint*

(a) act1 < evt2 (act1 : instance d'action, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 (située sur une transition de l'automate du composant) ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

15. *createActionEventsSchedulingConstraint*

(a) act1 < Sevt2 (act1 : instance d'action, Sevt2 : set d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

(b) act1 < Tevt2 (act1 : instance d'action, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de l'instance d'action act1,

16. *createActionsEventSchedulingConstraint*

(a) Sact1 < evt2 (Sact1 : set d'action, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(b) Tact1 < evt2 (Tact1 : template d'action, evt2 : instance d'événement) : toute action réagissant à l'instance d'événement evt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

17. *createActionsEventsSchedulingConstraint*

(a) Sact1 < Sevt2 (Sact1 : set d'action, Sevt2 : set d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(b) Sact1 < Tevt2 (Sact1 : set d'action, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(c) Tact1 < Sevt2 (Tact1 : template d'action, Sevt2 : se d'événement) : toute action réagissant à n'importe quelle instance d'événement du set Sevt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

(d) Tact1 < Tevt2 (Tact1 : template d'action, Tevt2 : template d'événement) : toute action réagissant à n'importe quelle instance d'événement du template Tevt2 ne pourra être exécutée qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1,

18. *createActionStateSchedulingConstraint*

(a) act1 < st2 (act1 : instance d'action, st2 : état d'automate) : l'automate ne pourra passer à l'état st2 qu'après la fin de l'exécution de l'instance d'action act1,

19. *createActionsStateSchedulingConstraint*

(a) Sact1 < st2 (Sact1 : set d'action, st2 : état d'automate) : l'automate ne pourra passer à l'état st2 qu'après la fin de l'exécution de n'importe quelle instance d'action du set Sact1,

(b) Tact1 < st2 (Tact1 : template d'action, st2 : état d'automate) : l'automate ne pourra passer à l'état st2 qu'après la fin de l'exécution de n'importe quelle instance d'action du template Tact1.

7.3 La génération de la machine réactive

Cette partie s'intéresse à la machine réactive générée par le canevas Brenda. La première partie est consacrée à la conversion des automates en Esterel. La deuxième partie présente la génération du code de la membrane qui communique avec la machine réactive.

7.3.1 La conversion en Esterel

7.3.1.1 Correspondance entre la grammaire Esterel et les classes Java

Seules les correspondances des éléments de la grammaire utilisés pour la conversion seront donnés. Les termes en italique représentent des catégories syntaxiques, les termes en gras définissent les mots-clefs du langage. Le sigle *opt* signifie que les items sont optionnels.

Grammaire Esterel	Classes Java
root : *ModulesList*	ModulesList
ModulesList : *Module ModulesList*	ModulesList
Module : **module** <*Identifier*> *Interface Body* **end module**	Module

TAB. 7.1 – Correspondance des modules Esterel

Grammaire Esterel	Classe Java
Interface : *InterfaceDeclList*$_{\text{opt}}$	Interface
InterfaceDeclList : *InterfaceDeclList*$_{\text{opt}}$ *InterfaceDecls*	
InterfaceDecls : *TypeDecls* *FunctionDecls* *ProcedureDecls* *InterfaceSignalDecls*	
TypeDecls : **type** *TypeDeclList* **;**	
TypeDeclList : *TypeDecl* *TypeDeclList*, *TypeDecl*	
TypeDecl : <*Identifier*>	TypeDeclaration
FunctionDecls : **function** FunctionDeclList **;**	
FunctionDeclList : *FunctionDecl* *FunctionDeclList*, *FunctionDecl*	
FunctionDecl : <*Identifier*> (Type*IdentifierList*opt) : Type*Identifier*	FunctionDeclaration
IdentifierList : <*Identifier*> *IdentifierList*, <*Identifier*>	
ProcedureDecls : **procedure** ProcedureDeclList **;**	
ProcedureDeclList : *ProcedureDecl* *ProcedureDeclList*, *ProcedureDecl*	
ProcedureDecl : <*Identifier*> (Type*IdentiferList*opt) (Type*IdentifierList*opt)	ProcedureDeclaration
InterfaceSignalDecls : **input** *SignalDeclList* **;** **output** SignalDeclList **;**	
SignalDeclList : *SignalDecl* *SignalDeclList*, *SignalDecl*	
SignalDecl : <*Identifier*> <*Identifier*> : *ChannelType*	SignalDeclaration
ChannelType : Type*Identifier*	

TAB. 7.2 – Correspondance de l'interface du module Esterel

Grammaire Esterel	Classes Java
Body : *Statement*	Body
Statement : *Parallel* *NonParallel*	Statement (interface Java)
Parallel : *NonParallel* || *NonParallel* *Parallel* || *NonParallel*	ParallelStatement
NonParallel : *AtomicStatement* *Sequence*	
Sequence : *Sequence*opt ; *AtomicStatement*	SequenceStatement
AtomicStatement : **nothing** **pause** **halt** *Emit* *Sustain* *ProcedureCall* *If* *Loop* *Abort* *Await* *Every* *Trap* *Exit* *LocalSignalDecl* *RunModule*	 NothingStatement PauseStatement HaltStatement

TAB. 7.3 – Correspondance du corps du module Esterel

Grammaire Esterel	Classes Java
Expression : ...	DataExpression (interface Java)
FunctionCall : Function*Identifier* (*ExpressionList*opt)	FunctionCallExpression
ExpressionList : *Expression* *ExpressionList*, *Expression*	

TAB. 7.4 – Correspondance des Data Expressions

Grammaire Esterel	Classes Java
SignalExpression : Signal*Identifier* *SignalExpression* **and** *SignalExpression* *SignalExpression* **or** *SignalExpression* (*SignalExpression*)	SignalExpression (interface Java) EsterelSignal AndSignalExpression (MultiSignalExpression) OrSignalExpression (MultiSignalExpression)

TAB. 7.5 – Correspondance des Signal Expressions

Grammaire Esterel	Classes Java
DelayExpression : *BracketedSignalExpression* **immediate** *BracketedSignalExpression* *Expression BracketedSignalExpression*	DelayExpression (interface Java) ImmediateExpression
BracketedSignalExpression : Signal*Identifier* [*SignalExpression*]	BrackedSignalExpression EsterelSignal

TAB. 7.6 – Correspondance des Delay Expressions

Grammaire Esterel	Classes Java
Emit : **emit** Signal*Identifier* **emit** Signal*Identifier* (*Expression*)	EmitStatement EsterelSignal EsterelSignal SignalValue
Sustain : **sustain** Signal*Identifier* **sustain** Signal*Identifier* (*Expression*)	SustainStatement EsterelSignal EsterelSignal SignalValue
ProcedureCall : **call** Procedure*Identifier* (*VariableIdentifierList*opt) (*ExpressionList*opt)	ProcedureCallStatement
If : **if** *Expression ThenPart*opt *ElsifPartList*opt *ElsePart*opt **end if**opt	IfStatement
ThenPart : **then** *Statement*	
ElsePart : **else** *Statement*	
ElsifPartList : *Elsif* *ElsifPartList Elsif*	
Elsif : **elsif** *Expression ThenPart*opt	
Loop : **loop** *Statement* **end loop**opt	LoopStatement
Abort : **abort** *Statement* **when** *DelayExpression* **abort** *Statement* **when** *DelayExpression* **do** *Statement* **end abort**opt **abort when** *AbortCaseList* **end abort**opt	AbortStatement
AbortCaseList : *AbortCaseList*opt *AbortCase*	
AbortCase : **case** *DelayExpression* **do** *Statement* **case** *DelayExpression*	AbortCase
Await : **await** *DelayExpression* **await** *DelayExpression* **do** *Statement* **end await**opt **await** *AwaitCaseList* **end await**opt	AwaitStatement
AwaitCaseList : *AwaitCaseList*opt *AwaitCase*	
AwaitCase : **case** *DelayExpression* **do** *Statement* **case** *DelayExpression*	AwaitCase
Every : **every** *DelayExpression* **do** *Statement* **end every**opt	EveryStatement
Trap : **trap** *ExceptionDeclList* **in** *Statement* **end trap**opt	TrapStatement
ExceptionDeclList : *ExceptionDecl* *ExceptionDeclList*, *ExceptionDecl*	
ExceptionDecl : *Identifier*	
Exit : **exit** Exception*Identifier*	ExitStatement
LocalSignalDecl : **signal** *SignalDeclList* **in** *Statement* **end signal**opt	LocalSignalDeclarationStatement
RunModule : **run** Module*Identifier*	RunStatement

TAB. 7.7 – Correspondance des Statements

7.3.1.2 Conversion de l'automate d'un composant

L'exemple suivant illustre la conversion de l'automate du composant *StorageComponent* en code Esterel.

Afin de ne pas surcharger la lecture du code, les paramètres des méthodes ne sont pas considérés.

Algorithm 57 Interface du module Esterel de l'automate *storageAutomaton*

```
module component_6_component_2_storageAutomaton :
```

L'automate *storageAutomaton* du composant *StorageComponent* a été traduit en module Esterel. Un module Esterel est divisé en deux parties : une première correspond à l'interface (les signaux qui sont échangés avec l'extérieur), la deuxième à l'implantation.

```
% INTERFACE of module component_6_component_2_storageAutomaton
procedure component_6_component_2_load_impl()();
procedure component_6_component_2_store_impl()();
```

Ces procédures correspondent aux méthodes de l'implantation du composant *StorageComponent* qui seront effectivement appelées par l'automate.

```
input server_load_signal, server_store_signal,
      end_compress_compress_signal, end_compress_decompress_signal,
      component_6_component_2_synchronizationPoint_1,
      component_6_component_2_synchronizationPoint_0;
```

Il s'agit de la déclaration des signaux reçus par le module. Les signaux *server_load_signal* et *server_store_signal* correspondent respectivement aux méthodes *load* et *store* de l'interface serveur nommée *server* du composant. Les signaux *end_compress_compress_signal* et *end_compress_decompress_signal* correspondent respectivement à la fin de l'exécution des méthodes *compress* et *decompress* de l'interface cliente *compress*. Les signaux *component_6_component_2_synchronizationPoint_0* et *component_6_component_2_synchronizationPoint_1* sont les signaux permettant la gestion de la synchronisation entre les actions effectuées par les composants.

```
output end_server_load_signal, end_server_store_signal,
   compress_compress_signal, compress_decompress_signal,
   endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26,
   endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27;
```

Il s'agit de la déclaration des signaux émis par le module. Les signaux *end_server_load_signal* et *end_server_store_signal* correspondent à la fin de l'exécution des méthodes *load* et *store* de l'interface serveur nommée *server* du composant. Les signaux *compress_compress_signal* et *compress_decompress_signal* correspondent respectivement aux méthodes *compress* et *decompress* de l'interface cliente *compress*. Les signaux *endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26* et *endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27* sont les signaux qui indiquent la fin de l'exécution des actions qui sont gérées par les contraintes d'ordonnancement.

```
% END INTERFACE of module component_6_component_3_storageAutomaton
```

Algorithm 58 Corps du module Esterel de l'automate *storageAutomaton*

```
% BODY of module component_6_component_3_storageAutomaton
signal Sst in
   emit Sst ;
```

L'état initial de l'automate est l'état *st* : l'émission du signal Sst permet à l'automate Esterel de passer dans un état où il pourra réagir aux signaux correspondant aux événements qui le feront changer l'automate du composant d'état.

```
   loop
      await case immediate Sst do
         trap Sst_exit in
```

Le signal *Sst_exit* correspond à la sortie de l'automate du composant de l'état *st*.

```
            signal sTransition_transition_1, sTransition_transition_0 in
```

Les signaux *sTransition_transition_0* et *sTransition_transition_1* sont émis lorsque la transition 0, respectivement la transition 1, s'effectue, et donc les actions associées à cette transition seront exécutées.

```
%% voir code défini dans l'algorithme 59
%% voir code défini dans l'algorithme 60
%% voir code défini dans l'algorithme 61

            end signal
         end trap
      end await
   ||
      await case [tick]
      end await
   end loop
end signal
% END BODY of module component_6_component_3_storageAutomaton
end module
```

Algorithm 59 Code Esterel définissant la transition 0 (corps du module Esterel de l'automate *storageAutomaton*)

```
abort halt
     when case sTransition_transition_0 do
          await case immediate
             component_6_component_3_synchronizationPoint_0
          end await ;
```

Le point de synchronisation 0 est inséré par l'application de la contrainte d'ordonnancement qui gère l'ordonnancement entre l'exécution des méthodes *checkPermissions* du composant *SecurityComponent* et *load* du composant *StorageComponent*.

```
call component_6_component_3_load_impl()() ;
```

L'implantation de la méthode *load* est exécutée.

```
pause ;
emit
endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26 ;
```

Le signal *endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26* est émis lorsque l'action correspondant à l'exécution de l'implantation de la méthode *load* sur la transition 0 est effectuée (l'action est clairement identifiée sur l'automate : le principal avantage est ainsi d'ordonnancer sur une instance d'action et non pas sur n'importe quelle instance).

```
await case immediate
   component_6_component_3_synchronizationPoint_1
end await ;
```

Le point de synchronisation 0 est inséré par l'application de la contrainte d'ordonnancement qui gère l'ordonnancement entre l'exécution des méthodes *checkPermissions* du composant *SecurityComponent* et *load* du composant *StorageComponent*.

```
emit compress_decompress_signal ;
```

Dès que la méthode *load* est exécutée, le signal *compress_decompress_signal* est émis.

```
await case end_compress_decompress_signal
end await ;
```

On attend le signal de fin de l'exécution des actions de l'automate du composant *CompressComponent* liée à la réception du signal *compress_decompress_signal*.

```
emit
endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27 ;
```

Le signal *endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27* est émis lorsque l'action correspondant à l'interaction avec l'automate du composant *CompressComponent* sur la transition 0 est effectuée.

```
emit end_server_load_signal ;
```

Le signal end_server_load_signal correspond à la fin de l'exécution des actions liée à la réception du signal *server_load_signal*.

```
emit Sst
```

L'automate du composant retournant dans l'état initial, le signal *Sst* est donc émis.

Algorithm 60 Code Esterel définissant la transition 1 (corps du module Esterel de l'automate *storageAutomaton*)

```
case sTransition_transition_1 do
     emit compress_compress_signal
     await case end_compress_compress_signal
     end await ;
     call component_5_component_3_store_impl()() ;
     pause ;
     emit end_server_store_signal ;
     emit Sst
end abort ;
exit Sst_exit
```

Algorithm 61 Code Esterel définissant les conditions pour que les transitions se produisent (corps du module Esterel de l'automate *storageAutomaton*)

```
||
     every server_load_signal do
        emit sTransition_transition_0
     end every
```

A chaque réception du signal *server_load_signal*, le signal sTransition_transition_0 correspondant au fait que la transition 0 doit s'effectuer (et donc les actions associées à celle-ci seront exécutées).

```
||
     every server_store_signal do
        emit sTransition_transition_1
     end every
```

A chaque réception du signal server_store_signal, le signal sTransition_transition_1 correspondant au fait que la transition 1 doit s'effectuer (et donc les actions associées à celle-ci seront exécutées).

Plusieurs points de synchronisation sur une même action se traduit par la mise en parallèle d'instructions await.

7.3.1.3 Conversion du comportement d'un composant

L'exemple précédent montrait la conversion de l'automate du composant. L'exemple qui suit montre la conversion du comportement de ce même composant.

Un point intéressant est que l'implantation du canevas ne génère que les signaux qui sont réellement utilisés (dont les templates ont été créés), de même pour les procédures et les fonctions.

7.3.1.4 Conversion de l'automate d'une contrainte d'ordonnancement

L'exemple suivant illustre la conversion de l'automate de la contrainte d'ordonnancement définie dans la figure 3.6.

Algorithm 62 Module Esterel du composant *StorageComponent*

```
module component_6_component_3_Behaviour :
```

Le comportement du composant se traduit lui aussi en un module Esterel. Le nom *component_6_component_3_Behaviour* est construit lors de la phase d'instantiation. Le nom *component_3* est généré puis concaténé au nom du composite *component_6* afin de respecter le modèle hiérarchique introduit dans Fractal.

```
% INTERFACE of module component_6_component_3_Behaviour
procedure component_6_component_3_load_impl()();
procedure component_6_component_3_store_impl()();
```

Les méthodes sont déclarées.

```
input server_store_signal, server_load_signal,
   end_compress_compress_signal, end_compress_decompress_signal,
   component_6_component_3_synchronizationPoint_1,
   component_6_component_3_synchronizationPoint_0;
```

Les signaux en entrée correspondent à ceux qui sont utilisés par les automates (plus précisément à l'automate *storageAutomaton*) du composant .

```
output end_server_store_signal, end_server_load_signal,
   compress_compress_signal, compress_decompress_signal,
   endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27,
   endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26;
```

Les signaux en sortie correspondent à ceux qui sont utilisés par les automates (plus précisément à l'automate *storageAutomaton*) du composant.

```
% END INTERFACE of module component_6_component_3_Behaviour
% BODY of module component_6_component_3_Behaviour
run component_6_component_3_storageAutomaton
```

Le comportement du composant n'est décrit qu'à partir du seul automate *storageAutomaton*.

```
% END BODY of module component_6_component_3_Behaviour
end module
```

Algorithm 63 Interface du module Esterel de la contrainte

```
module component_6_component_7_checkLoadConstraint :
```

La contrainte est un composant : elle se traduit donc en module Esterel. La génération du nom est identique à n'importe quel autre composant.

```
% INTERFACE of module component_6_component_7_checkLoadConstraint
input endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26,
 endaction_component_6_component_2_securityAutomaton_state_0_transition_0_action_19;
```

Les signaux en entrée du module sont les signaux émis par les composants dont les actions doivent être ordonnées par cette contrainte. Dans cet exemple, les signaux correspondent à la fin de l'exécution des actions *checkPermissions* et *load*.

```
output component_7_synchronizationPoint_1, component_7_synchronizationPoint_0;
```

Les signaux en sortie du module sont les signaux que la contrainte émet pour gérer l'ordonnancement entre les actions des automates.

```
% END INTERFACE of module component_6_component_7_checkLoadConstraint
```

Algorithm 64 Corps du module Esterel de la contrainte

```
% BODY of module component_6_component_7_checkLoadConstraint
signal Sstate_2, Sstate_1, Sstate_0 in
```

L'automate de la contrainte a trois états : le nom de ces états a été généré.

```
emit Sstate_0;
```

L'automate rentre dans son état initial.

```
loop
   await
   case immediate Sstate_2 do
      halt;
```

Le dernier état correspond à l'état final de l'automate.

```
   case immediate Sstate_1 do
      trap Sstate_1_exit in
         signal sTransition_transition_0 in
            [
            abort
               sustain component_7_synchronizationPoint_1;
```

Lorsque l'automate de la contrainte est dans cet état, le signal *component_7_synchronizationPoint_1* est émis, ce qui signifie, dans cet exemple, que l'action *load* peut s'exécuter.

```
               halt
            when case sTransition_transition_0 do
               emit Sstate_2
            end abort;
            exit Sstate_1_exit
            ]
            ||
            every
endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26 do
               emit sTransition_transition_transition_0
            end every
```

Lorsque le signal *endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26* est reçu, ce qui signifie que l'action *load* est terminée, l'automate de la contrainte effectue donc la transition et passe dans l'état *state_2*.

```
         end signal
      end trap
   case immediate Sstate_0 do
      trap Sstate_0_exit in
         signal sTransition_transition_0 in
            [
            abort
               sustain component_7_synchronizationPoint_0;
```

Lorsque l'automate est dans cet état, le signal *component_7_synchronizationPoint_0* est émis, ce qui signifie, dans cet exemple, que l'action *checkPermissions* peut s'exécuter.

```
               halt
            when case sTransition_transition_0 do
               emit Sstate_1
            end abort;
            exit Sstate_0_exit
            ]
            ||
            every
endaction_component_6_component_2_securityAutomaton_state_0_transition_0_action_19 do
               emit sTransition_transition_0
            end every
```

Lorsque le signal *endaction_component_6_component_3_securityAutomaton_state_0_transition_0_action_19* est reçu, cequi signifie que l'action checkPermissions est terminée, l'automate de la contrainte effectue donc la transition et passe dans l'état *state_1*.

```
         end signal
      end trap
   end await
```

7.3.1.5 Conversion du comportement d'un composant composite

L'exemple suivant montre la conversion du composite. Il est important de noter comment se traduit le fait que le comportement du résultat de la composition des composants est donné par la composition des comportements de ces composants.

Algorithm 65 Interface du module Esterel du composite

```
module component_6_Behaviour :
% INTERFACE of module component_6_Behaviour
input itf_preinvoke_signal;
output end_itf_preinvoke_signal;
% END INTERFACE of module component_6_Behaviour
```

Algorithm 66 Corps du module Esterel du composite

```
% BODY of module component_6_Behaviour
signal
endaction_component_6_component_3_storageAutomaton_st_transition_0_action_27,
endaction_component_6_component_3_storageAutomaton_st_transition_0_action_26,
endaction_component_6_component_2_securityAutomaton_state_0_transition_0_action_19,
endaction_component_6_component_2_securityAutomaton_state_0_transition_0_action_20
in
  signal
   component_6_component_5_server_compress_signal,
   end_component_6_component_5_server_compress_signal,
   component_6_component_3_compress_compress_signal,
   end_component_6_component_3_compress_compress_signal,
...
   component_7_synchronizationPoint_1,
   component_7_synchronizationPoint_0,
   component_8_synchronizationPoint_1,
   component_8_synchronizationPoint_0,
   component_6_synchronizationPoint_3,
   component_6_synchronizationPoint_2,
   component_6_synchronizationPoint_1,
   component_6_synchronizationPoint_0 in
```

Les signaux échangés entre les sous-composants sont déclarés en local. Les signaux échangés entre les composants et les contraintes d'ordonnancement sont déclarés en local dans le module du composant composite. Si la contrainte d'ordonnancement impliquait un composant qui ne serait pas un sous-composant du composite, les signaux échangés entre eux seraient déclarés dans l'interface du module.

```
run component_6_component_3_Behaviour
      [signal
        component_6_component_3_server_load_signal
          / server_load_signal,
        component_6_component_3_compress_compress_signal
          / compress_compress_signal,
        component_6_synchronizationPoint_2
          / component_6_component_3_synchronizationPoint_1,
      ...]
```

Le module Esterel du composite est la mise en parallèle des modules issus de la conversion des comportements des composants de ce composite. Les signaux émis et reçus par le module *component_6_component_3_Behaviour* sont renommés ...

```
||
every component_6_component_3_compress_compress_signal do
   emit component_6_component_5_server_compress_signal
end every
||
every end_component_6_component_5_server_compress_signal do
   emit end_component_6_component_3_compress_compress_signal
end every
```

La liaison des interfaces des sous-composants se traduit par une émission du signal sur une interface serveur pour chaque réception du signal sur une interface cliente.

```
||
every component_8_synchronizationPoint_1 do
   emit component_6_synchronizationPoint_2
end every
||
run component_6_component_7_Behaviour
      [...]
...
```

Le module de la contrainte d'ordonnancement est exécuté en parallèle des modules des autres composants.

```
  end signal
end signal
% END BODY of module component_6_Behaviour
end module
```

7.3.2 La génération du code de la membrane

Le code de la membrane est constitué de deux parties :

1. la première partie concerne le code qui utilise la machine réactive,
2. la deuxième partie concerne le code utilisé par la machine réactive.

7.3.2.1 Première partie

Algorithm 67 Classe *component_6_MembraneCode*

```
/**
* Generated by
* BasicEsterelMembraneCodeGenerator
*/
public class component_6_MembraneCode
     extends component_6_BehaviourUserInterfaceImpl
     implements
       com.francetelecom.rd.behaviourlayer.
              libs.process.instantiation.generated.MembraneCode,
       apis.CompositeItf
```

Cette classe étend la classe *component_6_BehaviourInterfaceUserInterfaceImpl*, qui correspond au code qu'utilise la machine réactive. Elle implante les interfaces *MembraneCode* et *CompositeItf* (interface serveur du composite).

```
{
  private component_6_Behaviour _the_behaviour;
```

La classe *component_6_Behaviour* a été générée par la compilateur Esterel. Elle correspond à la machine réactive.

```
  public component_6_MembraneCode() {}
```

Le constructeur par défaut.

```
  public void MC_start() {
```

La méthode *MC_start* est issue de l'interface MembraneCode. Le code généré constitue l'instantiation de la machine réactive.

```
    try {
     component_6_component_3_Instance =
        (libs.StorageImpl)(Class.forName("libs.StorageImpl").newInstance());
    } catch (Exception e) {
        throw new BehaviourInstantiationException(e.getMessage());}
    try {
     component_6_component_2_Instance =
        (libs.SecurityImpl)(Class.forName("libs.SecurityImpl").newInstance());
    } catch (Exception e) {
        throw new BehaviourInstantiationException(e.getMessage());}
    try {
     component_6_component_0_Instance =
        (libs.LogImpl)(Class.forName("libs.LogImpl").newInstance());
    } catch (Exception e) {
        throw new BehaviourInstantiationException(e.getMessage());}
    try {
     component_6_component_1_Instance =
        (libs.EncryptionImpl)(Class.forName("libs.EncryptionImpl").newInstance());
    } catch (Exception e) {
        throw new BehaviourInstantiationException(e.getMessage());}
    try {
     component_6_component_5_Instance =
        (libs.CompressImpl)(Class.forName("libs.CompressImpl").newInstance());
    } catch (Exception e) {
        throw new BehaviourInstantiationException(e.getMessage());}
    _the_behaviour = new component_6_Behaviour(this);
```

La machine réactive est instantiée.

```
    _the_behaviour.reset();
    _the_behaviour.react();
```

La machine réactive est dans son état initial.

```
  }
  private void MC_beforeReact()
  {
  }
  public void preInvoke(){
```

7.3.2.2 Deuxième partie

Algorithm 68 Classe *component_6_BehaviourUserInterfaceImpl*

```
/**
* Generated by
* BasicEsterelMembraneCodeGenerator
*/
public class component_6_BehaviourUserInterfaceImpl
    implements
        component_6_BehaviourUserInterface,
        org.objectweb.fractal.api.control.UserBindingController
```

La classe générée implante les interfaces *component_6_BehaviourUserInterface*, qui correspond à l'interface générée par le compilateur Esterel et *UserBindingController*, qui correspond au contrôleur de liaisons du composite.

```
{
  protected libs.StorageImpl component_6_component_3_Instance;
  public void component_6_component_3_store_impl() {
    component_6_component_3_Instance.store_impl();
  }
```

Cette méthode implante l'appel de la méthode store contenue dans l'implantation du composant *StorageComponent*.

```
  public void component_6_component_3_load_impl() {
    component_6_component_3_Instance.load_impl();
  }
  protected libs.SecurityImpl component_6_component_2_Instance;
  public void component_6_component_2_checkPermissions_impl() {
    component_6_component_2_Instance.checkPermissions_impl();
  }
  protected libs.LogImpl component_6_component_0_Instance;
  public void component_6_component_0_logNotify() {
    component_6_component_0_Instance.logNotify();
  }
  protected libs.EncryptionImpl component_6_component_1_Instance;
  public void component_6_component_1_decrypt() {
    component_6_component_1_Instance.decrypt();
  }
  public void component_6_component_1_encrypt() {
    component_6_component_1_Instance.encrypt();
  }
  protected libs.CompressImpl component_6_component_5_Instance;
  public void component_6_component_5_decompress() {
    component_6_component_5_Instance.decompress();
  }
  protected boolean MC_end_server_preInvoke_signal=false;
  public void _0_end_itf_preInvokeM1_signal() {
    MC_end_server_preInvoke_signal = true;
  }
...
  public void addFcBinding (String clientItfName, Object serverItf){}
  public Object getFcBindings (String clientItfName){return null;}
  public void removeFcBinding (String clientItfName, Object serverItf){}
}
```