Composition flexible et efficace de transformations de programmes

Romain Lenglet

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 projet SARDES de l'INRIA et du laboratoire Logiciels, Systèmes et Réseaux (LSR)

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

présentée et soutenue publiquement

par

Romain Lenglet

le 25 novembre 2004

Composition flexible et efficace de transformations de programmes

Directeur de thèse :
Daniel Hagimont

JURY

M. Jacques Mossière	Président
Mme Laurence Duchien	Rapporteur
M. Bertil Folliot	Rapporteur
M. Daniel Hagimont	Directeur de thèse
M. Thierry Coupaye	Co-directeur de thèse
M. Mark Van Den Brand	Examinateur

Remerciements

Le travail présenté dans ce document a été réalisé dans le contexte d'une collaboration entre le laboratoire DTL/ASR (Architecture des Systèmes Répartis) de France Telecom dirigé par Kathleen Milsted, et le projet INRIA SARDES dirigé par Jean-Bernard Stefani.

Je tiens d'abord à remercier Thierry Coupaye, de l'encadrement qu'il a effectué tout au long de cette thèse. J'ai tout particulièrement apprécié son recul sur mon travail, et aussi bien le cadrage qu'il a effectué pour m'empêcher de me disperser, que l'autonomie qu'il m'a laissée pour explorer de nombreuses pistes de travail.

Je remercie également Daniel Hagimont, directeur de cette thèse, pour ses critiques significatives et détaillées sur mon travail, et notamment sur ce mémoire, qui m'ont donné un éclairage complémentaire et enrichissant.

Les conseils et les critiques de Thierry et Daniel ont été essentiels à l'aboutissement de ce travail, et notamment à la rédaction de ce long mémoire.

Je remercie vivement Laurence Duchien et Bertil Folliot d'avoir accepté d'être les rapporteurs de cette thèse, et Mark Van Den Brand d'avoir accepté d'examiner mon travail.

Je remercie Jacques Mossière de m'avoir accordé l'honneur d'être le président de mon jury.

Je tiens à remercier Kathleen Milsted pour m'avoir accueilli dans le laboratoire DTL/ASR, et pour m'avoir encouragé et appuyé dans mes initiatives. Je remercie également Jean-Bernard Stefani, pour m'avoir accueilli dans le projet SARDES, et pour ses discussions éclairantes.

Je remercie également chaleureusement Eric Bruneton pour nos discussions très constructives (et qui ont été parfois été décisives pour cette thèse), pour nos publications communes, et pour ses relectures de plusieurs parties de ce mémoire.

Je remercie Frédéric Dang Tran d'avoir utilisé et contribué à ce travail de thèse, dans le cadre du projet Jivaro de compilateur Java vers C, et Jacques Pulou pour sa relecture de la partie de ce mémoire qui concerne cet aspect.

Je remercie Shigeru Chiba, pour la discussion que nous avons eue sur la mesure des performances des systèmes de transformation, qui a fortement influencé mon travail sur cet aspect.

Je remercie également Pascal Déchamboux, Mourad, Mikaël, Sébastien, Olivier, Marc et Tahar (et j'en oublie certainement !), pour toutes les discussions animées que nous avons eues sur de nombreux sujets, et qui ont contribué à ce travail.

Je remercie Dominique Bouchot, pour la conception des logos graphiques des logiciels Jabyce et Jaidee.

Je tiens enfin plus généralement à remercier tous les membres du laboratoire DTL/ASR et du projet SARDES à l'INRIA, pour le cadre de travail amical et stimulant qu'ils m'ont offert.

Table des matières

Table des principes de conception

Table des figures

Liste des tableaux

Table des programmes

Première partie

Introduction et problématique

Chapitre 1

Introduction

L'évolution du domaine de l'ingénierie logicielle est itérative : 1) les bonnes pratiques de développement de logiciel sont identifiées, puis 2) une fois stabilisées elles sont formulées sous la forme de principes, puis 3) de nouveaux outils (concepts, langages, logiciels, etc.) sont introduits pour faciliter et systématiser l'application de ces nouveaux principes, puis le cycle recommence. L'un des plus grands enjeux actuels de l'ingénierie logicielle est de développer des outils pour systématiser le Principe de Séparation des Préoccupations, qui spécifie que les parties d'un système qui correspondent à des domaines et des responsabilités différents, doivent être spécifiées séparément et composables automatiquement.

Dans les dernières années, les travaux d'ingénierie logicielle séparent chaque système en deux parties : les préoccupations fonctionnelles, et les préoccupations techniques (non-fonctionnelles). En particulier, les modèles de composants ont été introduits pour séparer la préoccupation fonctionnelle d'un système de la préoccupation de son architecture, c'est-à-dire la description de la structure du système et des relations entre ses parties. Cependant, la plupart des modèles de composants considèrent seulement un ensemble limité de préoccupations techniques : configuration, cycle de vie, etc. De nombreux autres travaux se sont focalisés sur le lien entre préoccupations fonctionnelles et techniques : les approches à conteneurs, et les techniques de lien. Les approches à conteneurs (EJB, CCM, etc.) séparent les composants métier (ou composants fonctionnels) des préoccupations techniques (gestion de transactions, contrôle d'accès, etc.) qui sont mises en œuvre dans un moteur d'exécution appelé conteneur, en définissant un contrat strict entre composants fonctionnels et conteneurs. Les autres approches, telles que la réflexion (MOP - protocoles à méta-objets), la programmation par aspects (AOP) et la transformation de programme, se focalisent sur les mécanismes de mise en œuvre d'un lien entre préoccupations fonctionnelles et préoccupations techniques. Cependant, la plupart des approches à conteneurs et des techniques de lien fonctionnel / technique limitent arbitrairement les préoccupations techniques qui peuvent être liées aux préoccupations fonctionnelles. Surtout, ces approches ne considèrent pas l'architecture technique, c'est-à-dire le développement et la configuration de préoccupations techniques complexes, à l'échelle d'un système.

Le travail de thèse décrit dans ce mémoire s'inscrit dans un projet du laboratoire DTL/ASR de France Telecom, qui propose d'utiliser un même modèle de composant évolué (le modèle Fractal) pour développer et structurer à la fois l'architecture fonctionnelle et l'architecture technique d'un système. Ce projet aborde également le problème du lien entre ces architectures fonctionnelles et techniques développées par composants. La transformation de programme

a été choisie, parmi les techniques de lien fonctionnel / technique, parce que c'est la plus efficace et la plus générale (c'est la seule technique qui ne limite pas a priori les services techniques qui peuvent être liés). Dans ce contexte, la problématique qui est abordée plus précisément dans cette thèse est la conception d'un système de transformation de programme qui permet de composer efficacement de nombreux transformateurs de programme complexes. Cette problématique est donc celle de *l'architecture d'un système de transformation adaptable*. Ces travaux sont motivés par l'inadéquation des systèmes de transformation existants à notre contexte.

Le chapitre 2 détaille le contexte et la problématique introduits dans ce chapitre, et donne le plan de ce mémoire.

Chapitre 2

Problématique du lien fonctionnel / technique

D'un point de vue général, nous considérons que l'évolution du domaine de l'ingénierie logicielle, c'est-à-dire l'évolution des outils (logiciels, langages, concepts...) pour le développement de logiciels, a pour objectif de systématiser les bonnes pratiques qui apparaissent et sont progressivement identifiées au cours du temps. Ce domaine évolue à cause des besoins croissants des systèmes logiciels [Ste03] en :

- *répartition* : les éléments sont géographiquement dispersés et interagissent à travers des interconnexions réseau ;
- *dynamicité* (*ouverture*) : des éléments peuvent être ajoutés ou enlevés pendant l'exécution, ou être déplacés physiquement, et des erreurs peuvent se produire ;
- *irrégularité* (*hétérogénéité*) : les fonctions et les capacités des éléments des systèmes peuvent différer et sont en constante évolution ;
- *échelles multiples* : les systèmes comprennent des éléments de traitement, de stockage et de communication qui ont des capacités très variables et mobilisables à des échelles différentes (niveaux d'abstraction différents), formant des hiérarchies.

Nous décrivons la démarche globalement suivie dans le domaine du développement logiciel, d'abord comme une identification des bonnes pratiques qui apparaissent, pour prendre en compte ces besoins croissants. Lorsque ces bonnes pratiques sont bien établies, elles sont généralement formulées de manière concise sous la forme de *principes*, c'est-à-dire sous la forme de « proposition premières, admises au départ d'un raisonnement ou d'une action » [dlLF]. Ensuite, de nouveaux outils sont conçus pour *systématiser, donc faciliter, l'application de ces nouveaux principes* pour le développement de logiciel. Les nouveaux outils ne permettent pas de garantir le développement de bons logiciels, ou l'application correcte des principes [MK90]. Seulement, les nouveaux outils facilitent l'application de principes lorsque des développeurs souhaitent les appliquer.

Cette « théorie de l'évolution » de l'ingénierie logicielle est vérifiée en pratique. Notamment, les concepts et les langages de programmation par objets (Smalltalk, C++, Java, etc.) ont été introduits pour faciliter, par rapport à la programmation procédurale ou fonctionnelle, l'application des principes du Polymorphisme et du Masquage de l'Information. Il existe de nombreux mécanismes de polymorphisme [MK90]. Dans les langages offrant un mécanisme

d'héritage, le principe de Polymorphisme est aussi reformulé dans le principe de Substitution de Liskov : *« les sous-types doivent pouvoir remplacer leur type de base »* sans modifier le code qui l'utilise [LW94, Mar03]. Le principe du Masquage de l'Information spécifie que *« les détails d'implantation doivent être masqués au public »* [MK90]. Les langages de programmation par objets ont eu un tel succès, qu'il est aujourd'hui rarement envisagé d'appliquer les principes du Polymorphisme et du Masquage de l'Information sans utiliser un langage de programmation par objets.

Ainsi, en poursuivant dans cette évolution, les avancées récentes en architecture et en ingénierie logicielle ont essentiellement pour objectif de définir de nouveaux outils, tels que par exemple les modèles de composants et la programmation par aspects (AOP) décrits dans la suite, pour systématiser l'application d'autres principes. La suite de ce chapitre donne un aperçu de ces principes et de ces nouveaux outils. Surtout, ce chapitre décrit nos objectifs sous la forme des principes dont nous souhaitons systématiser l'application, et introduit les outils que nous avons spécifiés et développés.

2.1 Séparation des préoccupations

Parmi les principes de conception que les outils de développement de logiciel tentent d'appliquer systématiquement, l'un des plus importants et des plus classiques est le principe de *Separation of Concerns* [Par72, HL95]. En français, ce principe est appelé principe de Séparation des Idées [BBMW02] ou principe de Séparation des Préoccupations. Nous l'avons reformulé dans le principe 1. Chaque préoccupation correspond à un domaine d'expertise d'un développeur, et à des responsabilités bien identifiées. Les besoins croissants en répartition, dynamicité, irrégularité et en niveaux d'abstractions multiples, évoqués ci-dessus, impliquent une complexité croissante de mise en œuvre de chaque préoccupation. C'est pourquoi, dans un système, il est nécessaire de pouvoir spécifier / implanter les préoccupations séparément, afin que les développeurs spécialistes dans un domaine puissent se focaliser seulement sur ce domaine. Ainsi, la conception d'un système est plus facilement maîtrisée : comme expliqué dans [Par72], le bénéfice est de limiter l'impact des changements (c'est-à-dire de l'évolution) d'une partie d'un système sur les autres parties de ce système, lorsqu'une seule préoccupation est impactée.

> Les parties d'un système qui correspondent à des domaines et des responsabilités différents, doivent être spécifiées séparément et composables automatiquement. [HL95, Par72]

Principe 1 : Séparation des Préoccupations

Le principal objectif de notre travail est de développer des outils pour l'application efficace du principe 1 de Séparation des Préoccupations.

Trois problèmes se posent pour la séparation des préoccupations : 1) l'identification des différentes préoccupations, 2) la spécification séparée des préoccupations, et 3) leur composition [HL95].

En ce qui concerne l'identification des préoccupations, [HL95] donne une liste non exhaustive de préoccupations couramment considérées : la synchronisation, la localisation dans un système

réparti, les contraintes de temps-réel et la tolérance aux fautes. Autre exemple, le standard ISO RM-ODP (*Reference Model of Open Distributed Processing*) [ISO95b] donne une liste plus complète de *fonctions* : gestion des nœuds, points de reprise (*checkpoints*), reprise après défaillance, activation et réactivation des objets, duplication et migration d'objets, gestion de transactions, persistance, négociation, gestion des activités (*threads*), sécurité (contrôle d'accès, authentification, etc.).

Plus simplement, la majorité des travaux dans la littérature considèrent principalement la séparation d'un système en deux grandes parties : 1) une préoccupation dite *fonctionnelle*, et 2) une ou plusieurs préoccupations dites *non-fonctionnelles*. La préoccupation fonctionnelle est aussi souvent appelée préoccupation *métier* (*business*), car elle correspond au domaine premier de l'utilisateur du système, tel que la finance, la paie, ou la gestion de stocks. Les préoccupations non fonctionnelles sont aussi appelées préoccupations *techniques*, car leur rôle est de fournir le support de l'exécution des préoccupations fonctionnelles, et sont considérées à un niveau d'abstraction inférieur. Ainsi, dans le cadre du principe 2 d'Inversion de Contrôle ces deux parties sont appelées *contrôleur* (qui implante les préoccupations techniques), *contrôlé* (qui implante les préoccupations fonctionnelles) [JAF]. A l'origine, ce principe considérait comme seule préoccupation la gestion des activités d'un système et du séquencement des actions [JAF], et était également appelé principe de Hollywood (*« don't call us; we'll call you »*) [Vli96]. Il est cependant aujourd'hui généralisé à d'autres préoccupations telles que la configuration d'un système, comme formulé dans le principe 3 de Séparation Configuration / Utilisation.

> Le contrôle du séquencement des actions doit être séparé de l'implantation des actions. [JAF]

Principe 2 : Inversion de Contrôle

> La configuration (l'assemblage) des éléments d'un système doit être séparée de leur utilisation. [Fow04]

Principe 3 : Séparation Configuration / Utilisation

La séparation des préoccupations est généralement effectuée à un niveau d'abstraction supérieur à celui de l'implantation du système [Par72, HL95]. Une approche consiste en la définition d'un langage spécifique de haut niveau (DSL - *Domain Specific Language*) pour la spécification de chaque préoccupation [MCM+00]. Un DSL « capture l'expertise » dans le domaine d'une préoccupation donnée, ce qui facilite le travail d'un développeur spécialiste de ce domaine. Les mécanismes sont par exemple des librairies de code ou des machines abstraites. L'enjeu est de pouvoir fournir des mécanismes les plus ouverts possibles, permettant de spécifier une grande variété de politiques, tout en gardant un haut niveau d'abstraction pour l'expression des politiques, ce qui facilite la réutilisation à ces deux niveaux [MCM+00].

2.2 Approches de séparation des préoccupations

Nouveaux principes de conception

Dans les travaux récents, de nouveaux principes ont été définis, pour synthétiser les nouvelles bonnes pratiques apparues avec la programmation par objets. Ce travail d'identification et de sélection des bonnes pratiques existantes est essentiellement réalisé par la communauté des Méthodes Agiles, et en particulier dans la méthode de développement eXtreme Programming [BBMW02]. Notamment, onze principes de conception de logiciels ont été regroupés dans [Mar03]. La plupart de ces principes restent des reformulations ou des spécialisations du principe 1 de Séparation des Préoccupations. Plusieurs approches ont été développées pour appliquer ces principes, telles que les modèles de composants, les conteneurs, la réflexion et la programmation par aspects.

Modèles de composant

Le concept de *composant* a été introduit comme une extension du concept d'objet ou de classe, pour étendre sa portée du domaine de la programmation à celui de la conception d'applications à une plus large échelle, et à un plus haut niveau d'abstraction [Cha99]. Il existe plusieurs modèles de composants, qui ont des objectifs différents. Cependant, le point de vue est toujours le même : l'*architecture* d'un système, c'est-à-dire une description de haut niveau de ce système, notamment de sa décomposition en composants, de la fonctionnalité de ces composants, et de leurs interactions [SDK+95, ISO95a]. Nous considérons donc que l'objectif premier des modèles de composants est d'appliquer le principe 1 de Séparation des Préoccupations pour une préoccupation technique particulière : la *configuration* d'un système. Cette problématique est résumée dans le principe 3 de Séparation Configuration / Utilisation.

En tant qu'évolutions des modèles d'objets, les modèles de composants conservent l'application des principes de Polymorphisme et de Masquage d'Information et introduisent de nouveaux concepts, synthétisés dans [MT97]. Deux composants d'une configuration ne peuvent communiquer que s'ils sont liés par un *connecteur* (ou *liaison*). Ces liens matérialisent les dépendances entre les composants, ce qui facilite le raisonnement et la gestion de ces dépendances, et permettent éventuellement d'effectuer des traitements sur les interactions entre composants. Une *interface* (ou *port*) est un ensemble de points d'interaction d'un composant ou d'un connecteur, pour spécifier un service fourni ou requis.

Un ADL (*Architecture Description Language*) est un langage spécifique (DSL) pour décrire l'architecture d'un système, c'est-à-dire l'assemblage des composants et des connecteurs dans le système. Les mécanismes (compilateurs, etc.) implantés pour chaque modèle de composant permettent d'assembler automatiquement un système en fonction d'une spécification ADL.

En complément de ces travaux sur les modèles de composants, qui considèrent la configuration comme préoccupation technique essentielle, de nombreux autres travaux se sont focalisés sur la séparation entre préoccupations fonctionnelles et techniques, et / ou sur la séparation entre plusieurs préoccupations techniques. Il existe dans la littérature deux catégories d'approches pour ce problème. La première catégorie comprend les systèmes et les standards définis par des acteurs industriels, qui se focalisent sur la définition de *contrats* bien définis entre préoccupations fonctionnelles et techniques de manière à assurer la portabilité des architectures

fonctionnelles. La deuxième catégorie se focalise plutôt sur les *mécanismes d'établissement d'un lien* efficace et flexible entre préoccupations fonctionnelles et techniques. Ces deux catégories sont décrites dans la suite de cette section.

Approches par conteneur

Cette catégorie comprend *les modèles de composant métier* [Cha99], dont les plus représentatifs sont San Francisco d'IBM (SF) [BJNR98], Enterprise Java Beans (EJB) de Sun Microsystems [DÜYK01], le modèle de composants CORBA (CCM) [ea99] et Component Object Model (COM) de Microsoft [MC95] (puis .Net). Dans ces modèles, les composants sont appelés *composants métier*, car ils implantent seulement des préoccupations fonctionnelles. Les moteurs d'exécution de ces composants, appelés *conteneurs*, fournissent des *services techniques* à ces composants métier, automatiquement et de manière transparente. Les services technniques typiquement offerts sont : la persistance, la gestion des transactions, la gestion de la concurrence, et le contrôle d'accès. Le paramétrage des services techniques est effectué de manière déclarative, au moment du déploiement, sous la forme de *descripteurs de déploiement*. Ces descripteurs sont écrits dans un langage (DSL) spécifique à chaque modèle de composant métier. Ces différents modèles spécifient essentiellement des modèles de programmation et des *contrats précis entre les composants et les conteneurs*, généralement sous la forme d'interfaces de programmation (API).

Un premier problème est que ces modèles de composant métier industriels sont moins riches que les modèles de composant issus de la recherche en architecture logicielle, tels que le modèle Fractal [BCS02]. Notamment, la notion de connecteur est absente. Surtout, la notion de configuration, c'est-à-dire d'assemblage de composants, est soit inexistante (SF, EJB, COM), soit embrionnaire : il existe un concept d'*agencement* dans CCM, correspondant à une composition de composants, mais il n'est pas formellement défini et un agencement n'est pas lui-même un composant. Ainsi, la composition est « plate », et il n'est pas possible de décomposer un composant en sous-composants. Ceci limite la capacité de décrire des systèmes à de multiples niveaux d'abstraction, ce qui est pourtant un besoin croissant (cf. le début de ce chapitre). En résumé, les approches à conteneurs ne considèrent pas comme essentielle la préoccupation de la configuration d'un système, aussi bien pour la partie fonctionnelle que pour la partie technique.

Un autre problème, est que les contrats « lourds » imposés aux composants métier rendent ces modèles de composant inadaptés à la conception de services techniques, et limitent leur utilisation aux préoccupations fonctionnelles, et à des composants « à gros grain ». De plus, ces contrats entre composants et conteneurs sont rigides, c'est-à-dire que l'ensemble des services techniques mis en œuvre par un conteneur est fixé par le standard (SF, EJB, CCM ou COM), et ne peut pas être étendu ou spécialisé dans un contexte donné. Par exemple, il n'est pas possible de ne pas mettre en œuvre la persistance pour l'exécution « d'entity beans » dans un conteneur EJB. De plus, les « boutons » de paramétrage de ces services techniques, dans les descripteurs de déploiement, sont limités et fixés par les standards. Par exemple, il n'est pas possible de paramétrer une gestion de transactions imbriquées dans les EJB, avec les paramètres standards : les EJB ne supportent que les transactions plates.

Des approches plus récentes de conteneurs offrent des contrats plus simples pour la conception de composants métier [Fow04]. Notamment, PicoContainer [HHT] offre comme seul service

technique le contrôle de la configuration des systèmes, mais en offrant seulement des concepts basiques ne permettant pas de raisonner sur l'architecture (pas de connecteurs, etc.). Egalement, le projet Arctic Beans définit une architecture de conteneur, où il est possible de configurer facilement les services techniques qui sont déclenchés lors de l'occurrence d'interactions avec les composants (les objets d'interposition sont des chaînes de « filtres ») [ABG+01]. Cependant, ces systèmes ont toujours une même limitation que les modèles de composant métier ci-dessus : ils n'offrent pas d'adaptibilité dans la conception et la composition des services techniques, du point de vue de leur architecture. Ceci est la principale limite des approches actuelles basées sur les conteneurs : ils ne se préoccupent pas de la configuration, et ne décrivent pas *l'architecture technique*, c'est-à-dire l'architecture interne d'un conteneur. D'un point de vue plus général, les modèles de composants métier se focalisent sur l'architecture fonctionnelle des systèmes, et sur la réutilisation et la portabilité des composants métiers, sans se préoccuper de l'adaptabilité de l'architecture technique.

En ce qui concerne plus précisément le problème du lien entre architectures fonctionnelle et technique, la plupart des implantations de conteneurs utilisent des objets d'interposition. Dans cette technique, basée sur l'utilisation du patron de conception Décorateur (cf. l'annexe B.2), des objets interceptent toutes les interactions avec un composant, pour déclencher des services techniques, tels que la démarcation des transactions. Cette technique est l'une des moins efficaces et des moins générales, car elle impose des indirections parfois coûteuses dans les interactions, et qu'elle se limite aux actions externes des composants, ce qui limite a priori les services techniques implantables. La suite de cette section présente des approches de lien fonctionnel / technique plus efficaces et plus générales.

Les approches qui se focalisent sur la *technique* d'établissement automatique et efficace d'un lien entre préoccupations fonctionnelles et techniques, comprennent : la programmation par aspects (AOP), la réflexion et la transformation de programme. D'autres approches, similaires mais moins utilisées, et que nous n'abordons pas ici, sont : les filtres de composition [ABV92] et la programmation par sujets (SOP - *Subject-Oriented Programming*) [OHBS94].

Programmation par aspects

Dans la *programmation par aspects* (AOP - *Aspect-Oriented Programming*), une préoccupation est spécifiée dans un *aspect*, c'est-à-dire une partie d'un système qui représente complétement et exclusivement une propriété donnée du système [KLM+97, BSL01]. Les aspects sont spécifiés séparément, chacun dans un *langage d'aspect* spécifique (DSL), et sont automatiquement mélangés (tissage - *weaving*) pour former un système complet. Le mélange s'effectue à des *points de jonction* (points de mélange) choisis par un programmeur. Les approches d'AOP diffèrent principalement pour les langages d'aspects et les points de jonction disponibles. Les travaux sur l'AOP mettent l'accent principalement sur la sémantique des langages d'aspects et du mélange automatique des aspects. Les approches d'AOP pour Java comprennent notamment AspectJ [KHH+01], JAC (*Java Aspect Components*) [PSD+04], et Prose [PGA01]. Parmi des travaux récents, il est à noter que le conteneur EJB Jboss offre une technique d'AOP (JBoss AOP) pour configurer le déclenchement des services techniques dans un conteneur [BFB+], et donc établir un contrat adaptable entre composants fonctionnels et services techniques.

Réflexion

La *réflexion*, ou *réflexivité*, est une démarche qui consiste à considérer un même système à deux niveaux d'abstractions : 1) un niveau de base, et 2) un méta-niveau [HL95]. Le niveau de base concerne les entités fonctionnelles du système, par exemple des objets Java. Le méta-niveau manipule des abstractions de ces entités du niveau de base. A ce méta-niveau, ces abstractions sont *réifiées* (*concrétisées*) dans des entités appelées *méta-objets*. Ces méta-objets permettent de raisonner sur les objets du niveau de base, à un plus haut niveau d'abstraction. Par exemple, le langage Java fournit des classes de méta-objets (Class, Method, Field, etc.) qui permettent de raisonner sur la structure des objets Java pendant leur exécution. Les interactions entre un niveau de base et son méta-niveau forment un protocole à méta-objets (MOP - *Meta-Object Protocol*), dans deux directions : l'introspection et l'intercession. L'*introspection* est la réification des méta-objets pour permettre à un système de raisonner sur lui-même pendant son exécution. L'*intercession* est la capacité de modifier les propriétés et le comportement du système pendant son exécution, en interagissant avec des méta-objets. Par exemple, un protocole peut permettre de déclencher des traitements au méta-niveau en fonction des actions se produisant au niveau de base. Nous considérons que la réflexion n'est pas en elle-même une technique de lien entre préoccupations fonctionnelles et techniques. En revanche, les concepts définis par la réflexion permettent de décrire de nombreuses approches. En effet, dans de nombreuses approches, les préoccupations techniques sont généralement décrites à un niveau méta par rapport aux préoccupations fonctionnelles. Par exemple, la réflexion peut être utilisée pour mettre en œuvre la programmation par aspects (AOP) [Sul01].

Transformation de programme

La *transformation de programme* est un domaine très vaste [Vis], qui peut entre autres être utilisé pour établir un lien entre code fonctionnel et code technique. Plusieurs approches existent, qui sont équivalentes pour les transformations implantables : la *génération de code* d'objets d'interposition (utilisée dans les approches à conteneurs, cf. la section précédente), les compilateurs extensibles (p.ex. OpenC++ [Chi95], OpenJava [TCKI99]), la transformation de code source (p.ex. COMPOST [LH00], AspectJ), la transformation de code intermédaire (p.ex. Javassist [Chi00] pour la transformation de bytecode Java, JAC [PSDF01]), la transformation de code binaire (p.ex. Detours [HB99]), et la modification de machine virtuelle (p.ex. Prose [PGA01], Guaraná [OB99]). Ce domaine est tellement général que toutes les techniques de réflexion et d'AOP, et plus généralement toutes les techniques de lien fonctionnel / technique, peuvent être décrites comme se basant sur la transformation de programme, ou comme des catégories particulières de transformation de programme [PGA01]. Par exemple, Guaraná est une machine virtuelle Java étendue pour mettre en œuvre un protocole à méta-objets, et JBoss AOP [BFB+] s'appuie sur le système de transformation Javassist [Chi00].

Synthèse

La plupart des techniques de transformation de programme, de réflexion et d'AOP sont plus générales que la seule technique de génération d'objets d'interposition mise en œuvre dans les modèles de composant métier décrits (EJB, CCM, etc.). Ces techniques offrent une plus grande

flexibilité dans les services techniques qui peuvent être liés avec une architecture fonctionnelle donnée : elles n'imposent pas de contrat rigide.

Cependant, la réflexion et l'AOP sont des approches plus restrictives que la transformation de programme en général, dans le sens où elles restreignent les liens qui peuvent être mis en œuvre. Notamment, dans un protocole à méta-objets, un compromis doit être fait entre le détail des informations réifiées dans des méta-objets, et les performances et la simplicité d'utilisation du protocole. De même, dans l'AOP, un compromis doit être fait entre le nombre et le détail des points de jonction disponibles, et les performances et la simplicité. En revanche, ces approches apportent des concepts qui facilitent l'implantation des services techniques : méta-objets, aspects, etc. Cependant, nous considérons que ces concepts sont encore peu mûrs, et surtout ces concepts sont à un trop bas niveau d'abstraction pour pouvoir être utilisés dans la conception d'architectures techniques complexes, par rapport par exemple aux modèles de composants (cf. la section 2.2).

En effet, la grande limitation des approches à conteneur et des techniques de lien (réflexion, etc.) est qu'elles se focalisent sur le lien entre architectures fonctionnelles et techniques. Mais elles se situent à un très bas niveau, sans considérer la préoccupation de la configuration d'un système, et l'architecture du système dans son ensemble, pour lesquelles les modèles de composant sont les mieux adaptés.

2.3 Proposition générale et problématique

Implantation de préoccupations fonctionnelles et techniques avec Fractal

Nous considérons qu'il est nécessaire d'utiliser des abstractions appropriées pour maîtriser la complexité des systèmes, autant pour les préoccupations fonctionnelles que techniques. En effet, l'enjeu actuel est surtout de maîtriser la complexité de l'implantation des services techniques, qui augmente constamment. Ainsi, nous proposons classiquement de concevoir et d'implanter séparément les préoccupations fonctionnelles et techniques, en application du principe 1 de Séparation des Préoccupations. Comme illustré dans la figure 2.1, le cycle de développement forme un Y.

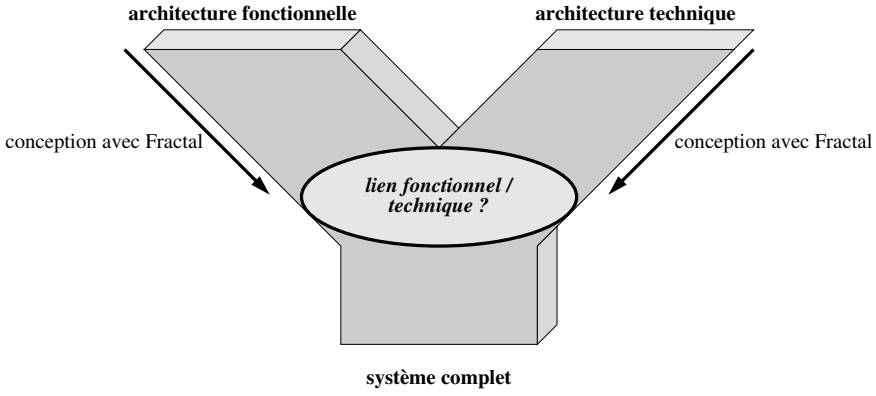


Fig. 2.1 : Cycle de développement en Y

Plus originalement, le projet de recherche et développement général dans lequel s'incrit cette thèse propose de concevoir et d'implanter les préoccupations fonctionnelles et techniques à l'aide d'un même modèle de composant, de manière à pouvoir décrire et gérer de manière homogène les systèmes, à tous les niveaux d'abstraction. Ainsi, notre principale préoccupation est la configuration des parties fonctionnelles et techniques d'un système. Le modèle Fractal a été spécifié dans cet objectif [CLB+01]. Le modèle Fractal, dans la lignée des modèles de composants issus de la recherche en architecture logicielle (cf. la section 2.2), offre des abstractions riches et adaptées pour maîtriser des systèmes complexes à large échelle, contrairement aux approches basées sur les conteneurs, les protocoles à méta-objets et la programmation par aspects. Notamment, le modèle Fractal est *minimal* et *extensible* (il n'impose que le contrôle de la configuration au minimum, mais peut être étendu pour d'autres contrôles), il permet des *composants à grain fin*, offre une capacité de *composition hiérarchique* des composants jusqu'à la conception de composants à très gros grain, etc. Surtout, la grande originalité de Fractal est de considérer les composants comme des entités existant à l'exécution, et non pas comme des fragments de code comme dans la plupart des autres modèles de composant, ce qui permet d'aborder directement les problèmes de dynamicité des systèmes. Fractal est décrit plus en détails dans la section 3.2.1. Un avantage important de Fractal est qu'il est adapté à la conception de services techniques comme des composants, à un bas niveau d'abstraction, contrairement aux modèles de composants industriels, tels que EJB.

Besoin d'un système de transformation général et efficace

Il reste à traiter le problème du lien entre des architectures fonctionnelles et techniques, conçues comme des configurations de composants Fractal. Il est nécessaire de choisir la technique de lien la plus générale, parce que 1) nous n'imposons pas de contrats entre architectures fonctionnelles et techniques, contrairement aux approches par conteneurs, et que 2) nous ne souhaitons pas limiter arbitrairement les services techniques qui peuvent être liés, au détriment de la facilité de construction d'un tel lien, contrairement à la réflexion et à la programmation par aspects. Comme montré dans la section 2.2, la transformation de programme, dans le sens général, est la technique à la fois la plus générale et la plus efficace pour établir un lien fonctionnel / technique. Nous choisissons donc d'utiliser la transformation de programme.

En généralisant l'utilisation de la transformation de programme pour établir un lien entre des configurations complexes de composants fonctionnels et techniques, le nombre de transformateurs de programmes augmente, et leur composition devient difficile. En d'autres termes, le système de transformation devient lui-même un système complexe, et nous considérons le problème nouveau de *l'architecture du système de transformation.*

Travail réalisé : conception d'un système de transformation de bytecode général, adaptable et efficace

Le travail réalisé dans cette thèse est la conception d'un système de transformation de programme général, baptisé Jabyce, qui permet de réaliser efficacement des compositions complexes de transformateurs de programme. Principalement, dans le cadre de cette thèse, Jabyce est appliqué à la mise en œuvre d'un lien fonctionnel / technique, mais Jabyce peut également être utilisé pour tout autre usage. Comme pour n'importe quel système complexe, nous

proposons d'appliquer les principes de conception présentés dans ce chapitre, notamment le principe 3 de Séparation Configuration / Utilisation pour séparer la composition de transformateurs de programme et leurs implantations. Surtout, un apport plus original est l'utilisation du modèle de composant Fractal pour la conception des transformateurs dans Jabyce, pour maîtriser la complexité du système de transformation. Ainsi, dans notre approche, à la fois les architectures fonctionnelles et techniques, et le système établissant le lien entre elles, sont conçus avec le même modèle de composants (Fractal), ce qui permet de considérer de manière homogène un système à tous les niveaux.

Ce travail de conception d'un système de transformation adaptable est nécessaire car, comme nous le montrons dans ce mémoire, les autres systèmes de transformation existants ne sont pas adaptés à des configurations complexes et / ou flexibles de transformateurs de programmes, notamment en termes de généralité et de performances.

Nous nous plaçons dans le cadre de la transformation de programmes Java compilés dans le langage intermédiaire interprété par une machine virtuelle Java (JVM), appelé aussi *bytecode Java*. Il est préférable de transformer des programmes sous cette forme, plutôt que sous la forme de code source Java, parce que cette représentation est plus générale (il existe d'autres langages que Java qui peuvent être compilés en bytecode Java), et plus facile à transformer. En effet, la JVM est une machine à pile, et les instructions de bytecode manipulent explicitement une pile, ce qui les rend en pratique plus faciles à manipuler et à transformer que des expressions du langage Java. D'autre part, le code source des applications Java à transformer n'est pas toujours disponible.

Dans la suite de ce mémoire, nous ne considérons que la problèmatique particulière de la construction de Jabyce, notre système de transformation de bytecode Java. Nous abordons la problématique de la construction d'un lien fonctionnel / technique, seulement pour la validation expérimentale de Jabyce dans le chapitre 7.

2.4 Démarche scientifique et plan du document

Cette section décrit la démarche générale qui est suivie dans ce mémoire. Elle ne constitue pas une justification ni le fond du travail qui est décrit, mais donne un éclairage ludique sur ce travail.

Concepts

Dans [Hof82], Douglas Hofstadter définit métaphoriquement un *concept* comme une boîte noire, offrant des « boutons de réglage » permettant de décrire l'ensemble des êtres compris dans ce concept. Chaque être correspond à une variation du concept, c'est-à-dire à une position des boutons. D. Hofstadter définit la créativité comme étant essentiellement la capacité à « glisser », c'est-à-dire à « tourner les boutons » d'un concept de manière à explorer ses variations. Les activités mentales humaines s'appuient fortement sur cette capacité. Un concept a potentiellement un nombre infini de boutons, et certaines variations peuvent permettre de découvrir de nouveaux boutons, ce qui est un autre aspect essentiel de la créativité. L'ensemble des variations potentielles d'un concept forme son *implicosphère*, ou « sphère implicite des variations conditionnelles ». Les implicosphères de deux concepts peuvent s'intersecter, si

ces concepts sont proches, c'est-à-dire que ces deux concepts peuvent comprendre des mêmes êtres.

Métaphores

D'après D. Hofstadter, deux concepts peuvent avoir certains boutons similaires, qui forment ainsi un *squelette conceptuel* commun à ces deux concepts. Il est ainsi possible mentalement de glisser simultanément dans deux concepts, en tournant simultanément les boutons de leur squelette conceptuel commun. Un moyen de provoquer un tel glissement chez un individu par la communication est la *métaphore*. D'un point de vue cognitiviste, la métaphore est l'expression linguistique du mécanisme mental d'un transfert à partir d'un concept-source familier vers un concept-but moins précis [Pra02]. Ainsi, la métaphore est un mécanisme très efficace pour provoquer mentalement un glissement dans un concept, en s'appuyant sur un glissement dans un concept plus familier. Un exemple de métaphore est justement la description par D. Hofstadter d'un concept comme une boîte à boutons.

La terminologie propre au domaine de l'informatique est relativement restreinte. La plupart des termes utilisés en informatique sont des utilisations métaphoriques de termes d'autres domaines : armée (« instruction », « stratégie », etc.), bureau (« document », « fichier », etc.), usine (« machine », « patron », « équipement », etc.), route (« bus », « traffic », etc.), etc. [Izw03]. Cette prolifération des métaphores en informatique montre que la créativité (dans le sens de D. Hofstadter) est une activité fondamentale en informatique. De plus, dans le domaine du génie logiciel, l'utilisation de la métaphore est une pratique classique, et recommandée dans de nombreuses méthodes de conceptions comme par exemple eXtreme Programming [Mar03], car c'est un outil de communication particulièrement efficace pour la compréhension de logiciels. Par exemple, les patrons de conception par objets systématisent l'utilisation de métaphores [GHJV94]. Ainsi, dans ce document nous utilisons également des métaphores, pour communiquer efficacement à la fois les variations dans les domaines que nous abordons, et l'introduction des nouveaux boutons dans le domaine de la transformation de programme.

Domaines scientifiques

Dans cette thèse, nous avons suivi une démarche créative, en abordant plusieurs domaines scientifiques que nous avons considérés comme des concepts au sens de D. Hofstadter. Nous avons abordé les domaines de l'architecture logicielle, de la sûreté de fonctionnement, de la recherche d'information, et surtout de la transformation de programme. Le travail présenté dans ce document consiste essentiellement en la définition des variations dans le domaine de la transformation de programme qui concernent notre problématique, et surtout en la définition de nouveaux boutons, ou dimensions, pour répondre à notre problématique. Notamment, l'une des dimensions que nous avons introduites est celle du « mécanisme de représentation interne » des éléments de programmes (par des objets ou par des interactions) dans les systèmes de transformation, qui nous permet d'introduire l'idée de représenter les programmes comme des séquences d'interactions. Pour les trois autres domaines abordés (architecture logicielle, sûreté de fonctionnement et recherche d'information), nous avons « tourné les boutons » de ces domaines, définis dans leurs états de l'art respectifs, pour explorer les intersections entre ces domaines et le domaine de la transformation de programmes. Notamment, une originalité

de notre travail est l'utilisation d'un modèle de composants logiciels général (Fractal) pour le développement et la composition de transformations de programmes complexes. Les relations entre les domaines scientifiques que nous avons abordés sont illustrées dans la figure 2.2. La partie grisée illustre les variations de ces comaines couvertes dans notre travail.

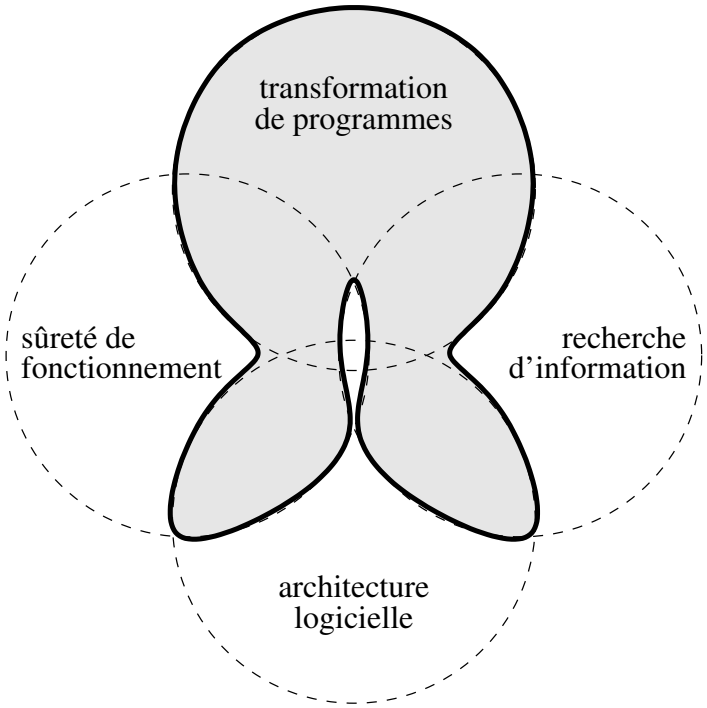


FIG. 2.2 : Domaines scientifiques abordés

Plan du document et guide de lecture

Il n'était pas possible d'établir un plan thématique pour ce document, avec par exemple un domaine scientifique abordé par chapitre, à cause des relations entre ces domaines, et de dépendances entre les concepts abordés dans chaque domaine. Nous avons donc choisi un plan qui permet une progression continue entre les chapitres, en limitant les références en avant dans le document.

Le chapitre 3 présente les objectifs de la conception de Jabyce, le système que nous proposons pour résoudre notre problématique, et nos choix pour cette conception, principalement en termes d'architecture logicielle. Le chapitre 4 présente notre proposition dans le domaine de la sûreté de fonctionnement, pour garantir la correction des programmes transformés avec Jabyce. Le chapitre 5 présente la catégorisation et les dimensions que nous avons définies dans le domaine de la transformation de programme, et compare Jabyce et les autres systèmes généraux de transformation de bytecode Java, en rapport avec nos objectifs. Le chapitre 6 complète cette comparaison qualitative par une comparaison quantitative, avec des mesures de performances. les chapitres 7, 8 et 9, qui présentent des validations expérimentales, respectivement dans le cadre de la mise en œuvre d'un lien entre une architecture fonctionnelle et une architecture technique pour la persistance d'objets, et dans le cadre de la compilation

bytecode Java → C. Le chapitre 10 est une synthèse de ce travail. Le chapitre 11 et le chapitre 12 décrivent des perspectives de travail, pour la simplification de l'implantation de transformateurs Jabyce, et dans le domaine de la recherche d'information pour la gestion de grandes bibliothèques de transformateurs de programmes.

Les quatre chapitres 3, 5, 6 et 10 constituent le cœur de ce mémoire. Le chapitre 4, sur la sûreté de fonctionnement, peut être lu indépendamment des autres chapitres, après le chapitre 3. De même, les chapitres 7, 8, 9, 11 et 12 ne sont pas essentiels, et peuvent être abordés lors d'une deuxième lecture seulement.

Deuxième partie

Jabyce : un système adaptable de transformation de programme

Chapitre 3

Aspects architecturaux de Jabyce[1]

Comme décrit dans la section 2.3, notre objectif est de concevoir un système de transformation de programmes Java compilés, général et efficace, permettant d'implanter et de composer des transformateurs de programme complexes. Ce chapitre présente la conception de Jabyce, le système de transformation que nous proposons, ses caractéristiques, et les motivations pour nos choix de conception.

Il existe deux grandes catégories de transformations [Vis01] de programme : la *réexpression* (*rephrasing*) et la *traduction* (*translation*). Les systèmes de réexpression transforment des programmes d'un langage vers le même langage, alors que les systèmes de traduction, p.ex. les compilateurs, traduisent des programmes d'un langage vers un autre. La réexpression est la catégorie de transformations qui nous intéresse principalement dans notre problématique, pour lier du code Java compilé fonctionnel avec du code technique. Cependant, Jabyce peut également être combiné avec d'autres systèmes de transformation similaires pour construire un système de traduction, comme démontré par le développement de Jivaro, un traducteur modulaire de bytecode Java → C décrit dans le chapitre 9.

3.1 Propriétés attendues du système de transformation

Notre objectif est de généraliser l'utilisation de Jabyce pour mettre en œuvre un lien entre code fonctionnel et services techniques. Nous considérons des ensembles de services techniques arbitrairements complexes. Cela se traduit par une grande complexité des transformations et

[1] La prononciation du nom « Jabyce » est notée ['dʒæbɪs] en phonétique, c'est-à-dire que « ja » est prononcé comme dans le mot anglais « jacket », et « abyce » comme le mot anglais « abyss ».

de leur composition. Dans ce cadre, le système de transformation Jabyce devient lui-même un système logiciel complexe, et se pose le problème de *l'architecture du système de transformation*, de même que pour tout logiciel complexe. Nous avons défini *trois principes de conception* pour synthétiser les objectifs de la conception de Jabyce, spécifiques à la conception de systèmes de transformation :

> Le système de transformation doit permettre *d'implanter toute transformation* de programme, c'est-à-dire il doit permettre n'importe quel traitement sur n'importe quelle partie d'un programme.

Principe 4 : Généralité du Système de Transformation

> Le système de transformation doit permettre de *séparer les transformations* en composants individuels réutilisables, qui peuvent ensuite être composés ensembles sans modifier leur implantation. Il est important de pouvoir réutiliser les programmes de transformations, qui sont généralement des logiciels complexes.

Principe 5 : Encapsulation et Composabilité des Transformations

> Le système de transformation doit permettre de *composer efficacement* ces composants de transformation. En particulier, le système doit au maximum éviter d'effectuer des traitements redondants dans les transformations composées.

Principe 6 : Composition Efficace des Transformations

Dans le cadre de la transformation de code compilé Java, la complexité du développement des transformations est en partie dûe à la nécessité, pour un développeur de transformations, de connaître la spécification de la JVM [LY99]. Notre objectif, avec Jabyce, n'est pas de faciliter le travail des développeurs, par exemple en offrant des abstrations qui masquent la complexité de la specification de la JVM, mais plutôt de permettre une réutilisation aisée de ces implantations complexes de transformations, en permettant de les composer sans modifier leur code. Cet objectif est synthétisé dans le principe 5 d'Encapsulation et de Composabilité des Transformations. De plus, il est envisageable de développer des outils de développement complémentaires à Jabyce, comme des langages de règles déclaratives et des compilateurs associés, pour faciliter le développement de transformations, mais ce n'est pas l'objectif de cette thèse.

Les aspects architecturaux de Jabyce sont décrits dans la suite de ce chapitre, de manière à montrer comment ces objectifs sont atteints, par l'application de patrons de conception et de principes de conception plus généraux, et par l'utilisation d'un modèle de composants logiciels général (Fractal). Du point de vue de notre démarche générale décrite dans la section 2.4, ce chapitre propose de nouvelles variations dans le domaines de la transformation de programme, et décrit l'intersection de ce domaine avec celui de l'architecture logicelle.

Nous définissons les termes suivants, qui sont utilisés dans ce chapitre et les chapitres suivants, pour décrire Jabyce et les autres systèmes de transformation de programme.

- Les *transformations* sont les traitements effectués par des transformateurs pour produire un programme transformé à partir d'un programme original.
- Les *transformateurs* sont des entités existant pendant la transformation des programmes, et qui peuvent être composées en différentes *configurations de transformateurs*.
- Jabyce lui-même est un *système de transformation*, c'est-à-dire une infrastructure permettant de développer des implantations de transformateurs et de composer des transformateurs.

Dans Jabyce, nous utilisons le terme *opération* pour désigner une forme particulière de transformateur. La distinction entre opération et transformateur, qui concerne leur architecture, est expliquée dans la section 3.2.6.

3.2 Architecture de Jabyce

Dans la littérature, les transformateurs sont généralement spécifiés soit 1) par des *règles de réécriture déclaratives* [Vis01], aussi appelées règles schématiques ou règles de remplacement de patron [PS83], et qui sont interprétées ou compilées, soit 2) par des *règles procédurales* [PS83], codées dans un langage de programmation généraliste. Cette distinction est similaire à celle faite dans le cadre des DSL (cf. section 2.1) : les langages de règles déclaratives limitent les transformateurs implantables, au profit d'une plus grande facilité d'expression, et d'un pouvoir de raisonnement plus grand sur les spécifications. Il est à noter que la qualification de « procédurale » pour une règle n'implique pas l'utilisation d'un langage de programmation procédural, mais seulement l'utilisation d'un langage de programmation généraliste, qu'il soit fonctionnel, procédural, orienté objet, ou autre. Dans Jabyce, nous avons fait le choix d'implanter les transformateurs comme des *règles procédurales codées en Java*, pour offrir une expressivité maximale, suivant notre principe 4 de Généralité du Système de Transformation. Cependant, dans ce chapitre nous présentons le paradigme de transformation que nous avons proposé pour Jabyce, c'est-à-dire les abstractions pour la conception de transformations, de manière orthogonale à cette distinction entre règles déclaratives et règles procédurales. En effet, une règle déclarative peut être compilée en une règle procédurale, avec le même paradigme de transformation.

L'application de nos principes 5 d'Encapsulation et de Composabilité des Transformations et 6 de Composition Efficace des Transformations a des implications sur la conception des abstractions offertes par le système de transformation pour l'implantation de transformateurs. La principale implication est que la composition des transformateurs est un *contrôle* dont les transformateurs doivent être indépendants, selon le principe 2 d'Inversion de Contrôle. Pour cela, nous avons choisi d'utiliser le modèle de composants logiciels généraliste Fractal, et Julia [Objb, BCL+04] une implantation de référence en Java, pour l'implantation des transformateurs. La section 3.2.1 présente le modèle Fractal, et la section 3.2.2 justifie ce choix, en montrant l'apport de ce modèle de composants pour l'encapsulation, la réutilisation et la composition des transformateurs Jabyce. Les sections 3.2.3 et 3.2.4 décrivent le modèle de représentation interne des programmes transformés dans Jabyce. Les sections suivantes détaillent l'architecture des éléments architecturaux de Jabyce.

3.2.1 Le modèle Fractal

Le modèle de composant Fractal, et Julia son implantation de référence en Java, sont réalisés dans le cadre du consortium Objectweb, principalement par le laboratoire DTL/ASR de France Telecom R&D et le projet SARDES (dirigé par Jean-Bernard Stefani) à l'INRIA.

Le modèle de composants logiciels Fractal [BCS02, BCS04, CLB+01, Objb] est une extension de modèles d'objets généraux comme celui du standard ISO RM-ODP [ISO95a, ISO95b], ou celui de Java. La terminologie utilisée dans Fractal s'inspire de celle de RM-ODP, et plus particulièrement du langage de traitement (*computational language*) [ISO95b], résumé dans l'annexe C. Fractal exhibe les propriétés suivantes :

- *dynamicité* : les composants sont des entités existant à l'exécution : ils peuvent êtres créés, détruis et (re)configurés pendant l'exécution du système.
- *encapsulation* : les composants interagissent, seulement à travers des points d'accès bien définis, appelés *interfaces*. Une *liaison* (*binding*) explicite peut lier deux interfaces. Les liaisons sont des canaux de communication arbitraires, concrétisés par exemple par des références d'objets Java ou des liaisons réparties.
- *identité* : les composants ont une identité bien définie, qui distingue de manière non-ambigüe un composant d'un autre, et qui permet d'interagir avec eux.
- *composition imbriquée* : les composants peuvent contenir des composants, recursivement. La récursion se termine avec des composants *primitifs* (ou *atomiques*), qui ont un contenu vide de composants et encapsulent directement des entités du langage sous-jacent, par exemple des objets Java. Les composants qui ont un contenu, c'est-à-dire qui contiennent des *sous-composants*, sont appelés des composants *composites*.
- *partage* : un même composant peut être contenu dans différents composites. Cette propriété est utile pour mutualiser des composants encapsulant des *ressources*, sans exposer de liaison vers ces composants ressources hors des composants composites qui les utilisent.
- *contrôle* : les composants offrent de manière transparente des capacités d'introspection et de d'intercession (cf. la section 2.2), permettant ainsi d'exercer un contrôle réflexif arbitraire pendant leur exécution. Par exemple, les contrôles offerts en standard par Fractal comprennent : l'ajout ou l'extraction de sous-composants de composants composites, le démarrage et l'arrêt de composants, l'attachement ou le détachement de composants en manipulant les liaisons.

Ces propriétés du modèle Fractal permettent de systématiser l'application de principes de conception lors la construction d'un système, et facilitent la construction de systèmes complexes, répartis, dynamiques et hétérogènes. Notamment, Fractal applique le principe 1 de Séparation des Préoccupations, en séparant la partie fonctionnelle (dans le contenu) et la partie technique (dans la membrane). Des préoccupations techniques peuvent être arbitrairement composées dans la membrane [BCL+04]. Ainsi, la spécification de Fractal est modulaire, et toutes les fonctionnalités ne doivent pas obligatoirement être toutes implantées dans une plate-forme conforme au modèle Fractal. Il existe ainsi plusieurs niveaux de conformité au modèle, correspondant à des capacités croissantes de contrôle. Le niveau de conformité le plus bas n'impose même rien du tout, et correspond par exemple directement à la plate-forme Java seule. Nous ne considérons ici que les niveaux de conformité élevés, qui offrent un contrôle riche sur les composants, notamment le contrôle de la composition imbriquée des composants.

A priori, toutes les propriétés de Fractal ne sont pas intéressantes pour la construction de systèmes de transformation de programmes. Notamment, la propriété de dynamicité ne nous

semble pas complétement nécessaire ici. Par exemple il est généralement inutile de pouvoir reconfigurer une configuration de transformateurs une fois qu'elle a été assemblée. En revanche, toutes les autres propriétés de Fractal, notamment en ce qui concerne l'encapsulation et le contrôle, sont mises à profit pour la conception de Jabyce, comme détaillé dans les sections suivantes.

Un composant Fractal expose des *interfaces* pour interagir avec d'autres composants. Les *interfaces serveur* spécifient les fonctionnalités offertes à d'autres composants, et les *interfaces client* spécifient les fonctionnalités qu'un composant requiert pour son fonctionnement. Dans la projection du modèle Fractal en Java, les signatures des interfaces sont des interfaces Java, c'est-à-dire des ensembles de signatures d'opérations, qui spécifient des ensembles d'interactions possibles entre deux composants. Une *liaison* est un lien orienté entre une interface client et une interface serveur, qui permet aux composants qui les exposent d'interagir. Une interaction est toujours initiée par l'interface client dans une liaison. Seulement des interfaces de types compatibles peuvent être liées. Le typage des interfaces dans Fractal est minimal, car il ne requiert qu'un prédicat de compatibilité entre types, mais il est extensible pour s'adapter aux différentes plate-formes d'exécution. Ainsi, dans la projection de Fractal sur Java, ce sont les relations de compatibilité entre interfaces Java qui sont utilisées. Les composants et les liaisons sont créés et manipulables pendant l'exécution du système.

Un composant Fractal est constitué de deux parties [BCS04, BCL$^+$04] : une *membrane* et un *contenu* (*plasme*). Ce modèle est inspiré par une métaphore entre un composant et une cellule biologique. Le contenu d'un composant composite est l'ensemble de ses sous-composants. Dans la projection de Fractal en Java, le contenu d'un composant primitif est le graphe des objets Java qu'il encapsule. Ce graphe d'objets est enraciné, car l'un de ces objets (la racine) implante les interfaces serveur du composant. Ainsi, seuls les composants primitifs ont une implantation sous la forme de code Java. La membrane peut offrir à la fois des *interfaces externes*, qui sont celles accessibles de l'extérieur du composant, et des *interfaces internes*, accessibles seulement par son contenu. Sur une membrane toutes les interfaces ont un nom local sous la forme d'une chaîne de caractères, dans un contexte de nommage pour les interfaces externes, et un domaine de nommage pour les interfaces internes. Dans le typage de Fractal, une interface interne ne peut exister que symétriquement à une interface externe, de même nom et de même type, pour *exporter* une interface externe de l'un de ses sous-composants. La figure 3.1 illustre un composant composite contenant deux composants primitifs. L'interface externe a du composant composite correspond à l'interface interne du même nom, lié par une liaison à l'interface d de l'un des sous-composants.

Le rôle le plus courant d'une membrane est *d'intercepter les interactions* sur ses interfaces, pour mettre en œuvre différentes formes de contrôle. Par exemple, une membrane peut suspendre les interactions initiées sur ses interfaces lorsque le composant est stoppé. La membrane peut offrir également des interfaces externes, appelées *interfaces de contrôle*, qui offrent des fonctionnalités d'introspection et d'intercession sur le contenu du composant. Fractal spécifie entre autres des interfaces pour la gestion d'attributs (AttributeController), pour l'attachement des liaisons sur les interfaces client d'un composant (BindingController) et pour la gestion des sous-composants d'un composite (ContentController) [BCL$^+$04]. La figure 3.2 illustre la membrane d'un composant offrant deux interfaces de contrôle. Le contenu du composant n'est pas illustré.

Le contrôle d'un composant s'effectue en interagissant avec ses interfaces de contrôle, à travers

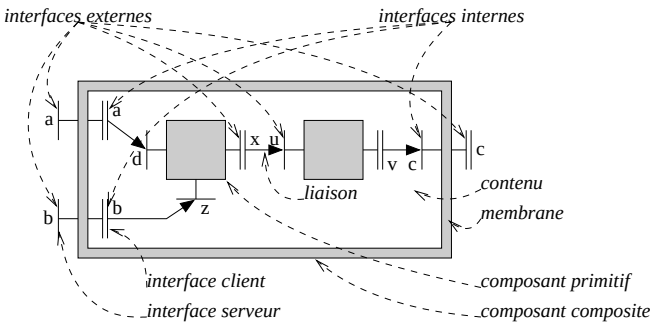


FIG. 3.1 : Composant Fractal composite

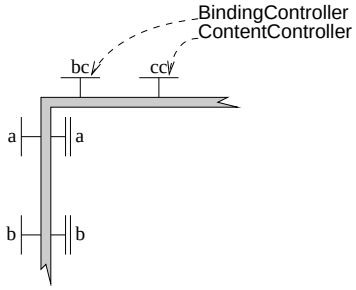


FIG. 3.2 : Interfaces de contrôle d'un composant Fractal

une liaison, comme avec n'importe quelle interface fonctionnelle. Le contrôle est donc réalisé *par programmation*, en implantant des composants qui interagissent avec de telles interfaces, de manière à permettre la plus grande expressivité possible. Il est cependant envisageable d'offrir des langages spécifiques (DSL, cf. la section 2.1) pour la spécification d'un contrôle particulier. Notamment, Fractal définit un *langage de description d'architecture* (ADL - *Architecture Description Language*) [BCS04], basé sur XML, qui permet entre autre de décrire l'image initiale d'une configuration de composants, en termes de composants, de sous-composants, d'interfaces offertes et requises par les composants, et de liaisons. Ce langage est *extensible*, pour supporter par exemple l'expression de directives de déploiement. Cette extensibilité permet concrètement de définir de nouveaux types d'éléments XML, grâce à une définition DTD extensible, et d'ajouter des fonctionnalités dans l'analyseur de fichiers ADL, sous la forme de composants Fractal, pour traiter ces nouveaux types d'éléments.

Les interfaces de contrôle d'une membrane sont implantées dans des entités appelées *contrôleurs*, et situées dans la membrane. Les mécanismes d'implantation des contrôleurs sont spécifiques aux implantations particulières de Fractal. Une implantation de Fractal peut offrir des mécanismes de configuration des contrôleurs présents dans la membrane de chaque composant. Par exemple, dans notre mise en œuvre de Jabyce, nous utilisons les mécanismes de configuration de Julia, pour utiliser une version de contrôleur de liaisons (interface BindingController) qui optimise les liaisons entre les composants, comme décrit dans la section 3.2.6.

Comparé à la grande majorité des autres modèles de composants, Fractal est *minimal*, dans le sens où il n'aborde à la base que le contrôle de la configuration des applications pendant

leur exécution, c'est-à-dire l'ensemble des composants et de leurs liaisons. Nous considérons ce contrôle de la configuration comme le contrôle le plus élémentaire, et qui est nécessaire même pour implanter d'autres contrôles [CLB+01]. Fractal aborde ce problème de manière générale et ouverte, en permettant de traiter de nombreux autres types de contrôle. Grâce à la séparation entre membrane et contenu, Fractal systématise l'application du principe 2 d'Inversion de Contrôle, en n'imposant quasiment pas de contraintes aux programmeurs de composants (primitifs) par rapport aux contrôles. Notamment, il est ainsi possible de concevoir un système avec de nombreux composants Fractal, définis à un grain arbitrairement fin, dans des configurations arbitrairement complexes, sans avoir de limitations en termes de performances ou de programmation. Cela permet d'utiliser Fractal non seulement pour développer des composants fonctionnels, mais aussi pour *développer des services techniques*, et plus généralement des intergiciels (*middlewares*), et des systèmes d'exploitation, sous la forme de composants. Le développement de services techniques par composants a été l'une des principales motivations pour la définition de Fractal. Nous considérons notamment le développement de gestionnaires de transactions, de services de persistance, de canevas de communication répartie, de serveurs d'application, etc. Dans notre contexte, un système de transformation de programme comme Jabyce est assimilé à un service technique. Fractal apparaît donc bien adapté à la conception de Jabyce.

3.2.2 Eléments d'architecture de Jabyce

De manière classique [dJVV01], une configuration de transformateurs est généralement conçue comme une *chaîne* constituée des éléments suivants :

- un *analyseur* (*parser*), aussi appelé *front-end*, qui analyse les programmes pour en produire une *représentation interne*.
- plusieurs *transformateurs* qui transforment les programmes en manipulant leur représentation interne.
- un *générateur de code*, aussi appelé *pretty-printer* ou *back-end*, qui génère des programmes à partir de leur représentation interne.

Dans Jabyce, nous appelons *désérialiseurs* les analyseurs, et *sérialiseurs* les générateurs de code. Cette dénomination reflète la distinction importante entre la forme des programmes en entrée et en sortie d'une configuration de transformateurs, que nous appelons la *forme sérialisée* des programmes, et la représentation interne de ces programmes. D'une part, une forme sérialisée d'un programme est non structurée, comme par exemple une séquence de caractères, mais le format est généralement fixe et standardisé. Dans le cas de Jabyce, un programme sérialisé est un ensemble de classes Java compilées, chacune représentée comme une séquence d'octets formattée selon la spécification de la JVM [LY99] (c'est le format des fichiers .class). D'autre part, une représentation interne de programmes est généralement bien structurée, par exemple sous la forme de graphes d'objets Java bien encapsulés, et n'est pas standardisée. La figure 3.3 illustre une configuration de transformateurs, avec un désérialiseur, deux transformateurs, et un sérialiseur.

Dans Jabyce, à ce niveau, nous appelons les transformateurs des *opérations de transformation*. Cette distinction entre transformateur et opération est motivée et expliquée dans la section 3.2.6. Dans Jabyce, les désérialiseurs, opérations et sérialiseurs sont des *composants Fractal implantés en Java*, qui sont liés et qui interagissent en échangeant des programmes dans

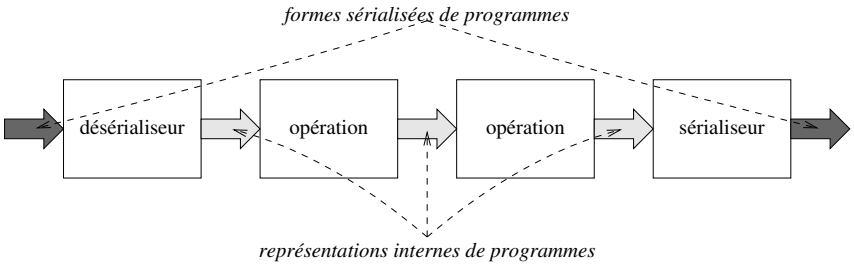


FIG. 3.3 : Configuration de transformateurs

une représentation interne. Les interfaces de ces composants sont décrites dans les sections suivantes.

Nous définissons un modèle de représentation interne de programmes comme la combinaison d'un *modèle conceptuel* et d'un *mécanisme de représentation interne*. Un modèle conceptuel définit le découpage des éléments des programmes en *types d'éléments* dans la représentation internes, et les relations manipulables entre les types d'éléments (par exemple relations de composition ou de dépendance sémantique). Un mécanisme de représentation interne est le choix d'une représentation interne des éléments de programmes soit 1) par des objets formant un graphe, soit 2) par des interactions. Le modèle conceptuel et le mécanisme de représentation interne choisis dans Jabyce sont présentés dans les sections 3.2.3 et 3.2.4. Ces deux dimensions sont orthogonales.

3.2.3 Modèle conceptuel de classe Java dans Jabyce

Dans de nombreux systèmes de transformation, le modèle conceptuel de programme de la forme sérialisée (le format) et celui de la représentation interne sont très similaires, voire confondus. Le format des classes Java compilées est décrit dans l'annexe A. Dans ce cas, les éléments formant la représentation interne d'un programme correspondent directement aux éléments syntaxique du langage des programmes transformés. Cependant, nous considérons plus généralement que le choix d'un modèle conceptuel d'une représentation interne de programme comme une *conséquence des objectifs du système de transformation.*

Un programme est modélisé conceptuellement par un graphe, dont nous appelons les nœuds des *éléments*, et les arcs sont par exemple des relations de composition. Cette section décrit le modèle conceptuel de Jabyce, c'est-à-dire l'ensemble des types d'éléments permettant de représenter des classes Java compilées et les relations entre ces types d'éléments.

La JVM permet de charger dynamiquement des classes Java compilées, pendant l'exécution du système. Chaque classe compilée qui forme un programme est en effet chargée par la JVM à la première utilisation de cette classe, par un objet appelé *chargeur de classe* (*class loader*) [LY99]. Un chargeur de classe retourne à la JVM la séquence d'octets qui forme une classe Java compilée, c'est-à-dire le contenu d'un fichier .class, en fonction du nom de la classe Java demandée. La JVM exécute ensuite le programme, en compilant ou en interprétant ces classes. Il est possible de *transformer au chargement* les classes Java, en implantant un chargeur de classe qui désérialise, transforme et sérialise les classes avant de les passer à la JVM [Chi00]. Jabyce doit permettre de transformer chacune des classes Java au chargement.

Cette granularité du chargement de programmes compilés, impose que Jabyce transforme une classe Java à la fois. Une classe compilée Java est donc la plus grosse *unité de transformation.*

Le choix du modèle conceptuel de Jabyce, c'est-à-dire des types d'éléments ou *granules de transformation*, correspond plus précisément à deux considérations que nous avons identifiées :

- la *granularité des transformations* : on doit faire un compromis entre la taille et la complexité des éléments dans une représentation interne d'un programme, et le nombre de transformations nécessaires pour transformer ce programme complétement.
- le *niveau d'abstraction des éléments* : quels détails des programmes et de leur forme sérialisée le modèle abstrait-t'il ?

La granularité a un impact sur les performances et la simplicité d'implantation des transformations. Par exemple, un modèle à *gros grain* d'une classe Java compilée pourrait comprendre seulement trois types d'éléments complexes : « classe », « méthode », et « attribut ». Le type d'élément « méthode » contiendrait les instructions de bytecode non décodées, sous la forme d'un tableau d'octets. Avec ce modèle, toute opération de transformation d'instructions doit elle-même décoder et encoder ces instructions, ce qui est coûteux à l'exécution et complexe à implanter. En revanche, un modèle à *grain fin* pourrait comprendre un type d'élément pour chaque type d'instruction défini dans la spécification de la JVM [LY99]. Avec ce modèle, le décodage est réalisé par le désérialiseur. Les opérations de transformation d'instructions sont alors plus facilement implantables. Cependant, si dans une configuration aucune opération ne transforme d'instructions, le décodage et l'encodage des instructions est effectué inutilement. Le choix de la granularité est donc motivé par les objectifs, c'est-à-dire par les transformations que le système permettra d'implanter, et le modèle résultant doit être optimisé pour cette classe de transformations.

Dans Jabyce, nous avons spécifié un modèle de classes Java 1) qui respecte notre principe 4 de Généralité du Système de Transformation, c'est-à-dire qui permet de transformer tout élément d'une classe Java compilée, 2) tout en permettant de *factoriser* au maximum dans les (dé)sérialiseurs le traitement des aspects difficiles du format des classes Java compilées. Notamment, les (dé)sérialieurs sont chargés du décodage et de l'encodage des intructions. Cette factorisation permet d'obtenir une bonne efficacité lorsque de nombreux transformateurs sont composés, conformément à notre principe 6 de Composition Efficace des Transformations. Ces aspects sont décrits en détails dans la section 5.4, avec une comparaison avec les modèles d'autres systèmes de transformation de bytecode Java.

Dans Jabyce, le modèle conceptuel d'une classe Java compilée est un graphe dont les nœuds sont des types d'éléments, et les arcs sont entre autres des relations de composition. Ce modèle est illustré dans la figure 3.4. Dans ce modèle conceptuel, le type d'élément racine est Class. Une Class est un *élément composite* qui peut être composé d'éléments InterfaceInheritanceDeclaration et InnerClassDeclaration, pour les déclarations des interfaces implantées par les classes et pour les classes internes. Une Class peut contenir des éléments Field composites pour les attributs, qui peuvent contenir un élément FieldConstantValue. Une Class peut contenir des éléments Method composites. Une Method peut contenir des éléments ThrownExceptionDeclaration et MethodFormalArgument, pour la déclaration de la signature des méthodes. Une Method peut aussi contenir des éléments pour le code : des éléments LocalVariable, MethodFormalArgumentLocalVariable et ThisLocalVariable pour la déclaration des variables locales, des éléments InstructionLabel pour identifier les instructions (pour les sauts, etc.), et des éléments pour les instructions. Jabyce définit 30 types d'éléments pour les instructions, comme ReturnInstruction,

LocalVariableAccessInstruction et FieldAccessInstruction. Au total, Jabyce définit 50 types d'éléments pour modéliser les classes Java compilées, dont 3 types d'éléments composites (Class, Method et Field). Une classe Java compilée est modélisée dans Jabyce comme un graphe, avec un élément Class comme *racine*. Le découpage des types d'instructions est détaillé dans la section 5.4.

La spécification de ce modèle conceptuel de représentation interne est une composante essentielle de la spécification de Jabyce. En revanche, aucune forme sérialisée n'est imposée par Jabyce, et il est possible d'implanter des désérialiseurs et sérialiseurs pour analyser et produire des programmes dans n'importe quelle forme sérialisée. Jabyce comprend en standard l'implantation d'un désérialiseur et d'un sérialiseur pour une forme sérialisée de classes Java compilées sous la forme de tableaux d'octets. Ces deux implantations de composants sont basées sur le système de transformation de bytecode Java ASM [BLC02, Objа], développé également dans le laboratoire DTL/ASR de France Telecom R&D. ASM est décrit plus en détails dans le chapitre 5.

3.2.4 Représentation des classes Java par des séquences d'interactions

Une fois qu'un modèle conceptuel a été défini, il reste encore à choisir un mécanisme pour concrétiser et communiquer des représentations de programmes. Il existe deux *mécansimes de représentation interne* : par des graphes d'objets, ou par des séquences d'interactions.

A notre connaissance, tous les systèmes existants de transformation de programmes, à part ASM et Jabyce (cf. la section 5.3), représentent les programmes par des *graphes d'objets*, p.ex. des graphes d'objets Java. Typiquement, avec ce mécanisme de représentation, une classe Java est définie pour chaque type d'élément défini dans le modèle conceptuel, par exemple des classes Java Class, Method, etc. Les arcs sont les références entre les objets de ces classes. Dans ce cas, le modèle conceptuel et le mécanisme de représentation interne sont généralement confondus. Cependant, avec Jabyce et ASM nous avons introduit, comme mécanisme de représentation interne, la représentation des programmes par des *séquences d'interactions*. Dans cette approche, chaque élément d'un programme est représenté par une interaction, c'est-à-dire par exemple par un appel de méthode. Cette approche a d'abord été appliquée dans ASM [BLC02], puis appliquée dans Jabyce. C'est pour pouvoir introduire cette approche que distinguons le modèle conceptuel et le mécanisme de représentation interne, pour la représentation interne de programmes.

Notre approche a été motivée par une analogie entre un programme et un document semi-structuré XML, donc entre un système de transformation de programmes et un système de traitement de documents XML. D'après [Proc], il y a deux approches pour la représentation interne des documents XML et SGML dans des programmes Java :

- en représentant un document comme *un graphe d'objets* Java, comme avec les APIs DOM (*Document Object Model*) [HHW+00].
- en représentant un document comme *une séquence d'appels de méthodes* (séquence d'événements), comme avec les APIs SAX (*Simple API for XML*) [MRBM+].

Avec une représentation en graphe d'objets DOM, un document est transformé par un transformateur en appelant des méthodes sur les objets dans le graphe, pour ajouter des objets nœuds, enlever des sous-graphes, modifier des objets nœuds, etc.

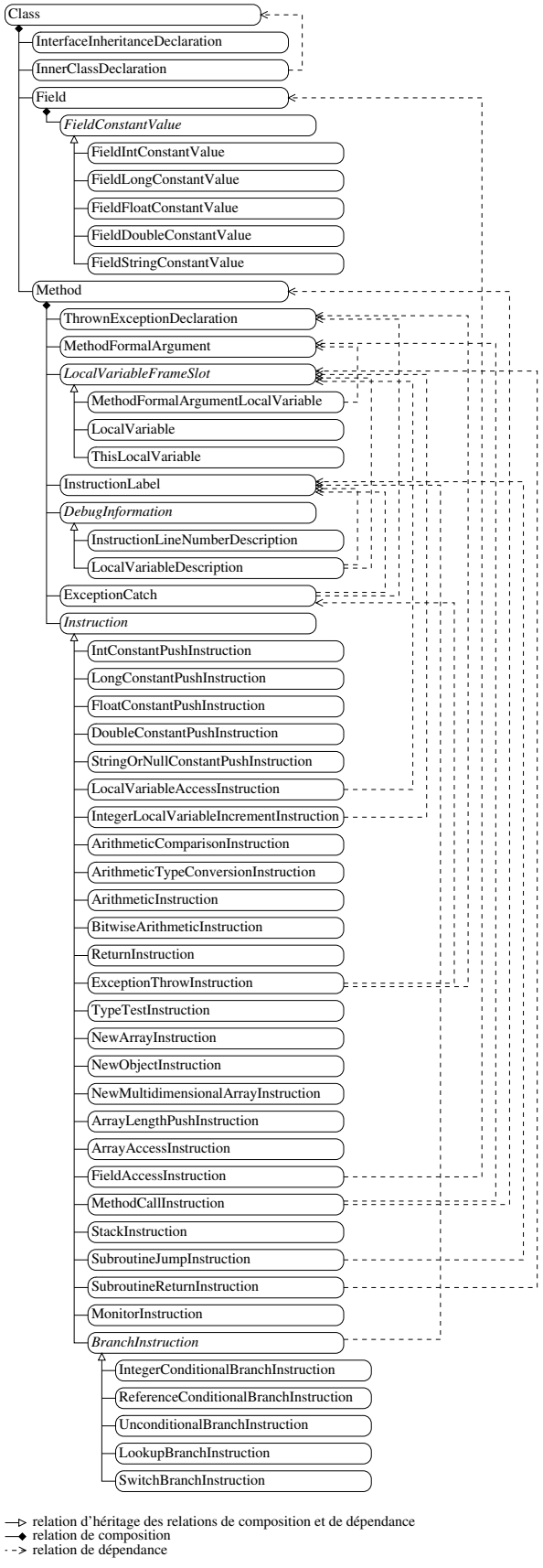


Fig. 3.4 : Modèle conceptuel de Jabyce

Avec SAX, la représentation interne d'un document XML est une séquence d'appels de méthodes sur un objet implantant l'interface ContentHandler. Chaque appel représente le début d'un élément XML, ou la fin d'un élément XML, ou le contenu d'un élément XML, etc. Avec une représentation en séquence d'appels de méthodes SAX, il est possible de concevoir un transformateur de document XML comme un objet implantant l'interface ContentHandler, et qui « décore » les appels vers un autre objet ContentHandler, selon le patron de conception Décorateur [GHJV94] (cf. l'annexe B.2). Un document est transformé par un tel transformateur en produisant une nouvelle séquence d'appels représentant un nouveau document, en fonction d'une séquence d'appels reçue par le transformateur, et en *décorant* cette séquence d'appels reçue. Par exemple, il est possible d'enlever un élément XML d'un document simplement en ne réémettant pas les appels qui notifient le début et la fin de l'analyse de cet élément. Les autres appels de la séquence reçue sont conservés lors de l'initiation d'une séquence d'appels par le transformateur. Le programme 3.1 montre le code source d'un transformateur qui transforme les documents XML analysés, en délégant les appels reçus à l'objet décoré dont la référence est stockée dans l'attribut traitantSuivant, et en représentant un nouvel élément <nouvelElement/> dans chaque élément XML représenté. Cet exemple peut être généralisé, pour implanter des transformateurs comme des objets qui peuvent être liés entre eux pour se décorer mutuellement via leurs attributs traitantSuivant, pour former une chaîne comme celle illustrée dans la figure 3.3.

Dans la perspective de notre démarche générale introduite dans la section 2.4, en ce qui concerne ce « bouton » du mécanisme de représentation interne de programmes, nous résumons notre démarche par la métaphore projective suivante : *Jabyce est à la transformation de programmes ce que SAX est à l'analyse de documents XML.*

Le choix d'un mécanisme de représentation interne, c'est-à-dire soit par des graphes d'objets, soit par des séquences d'interactions, est motivé par deux caractéristiques contradictoires des systèmes de transformation : 1) la *performance* de transformation, en termes de consommation de ressources, et 2) la *simplicité* d'implantation, (c'est-à-dire la consommation du temps de travail d'un développeur). Globalement, nous considérons que deux ressources sont consommées par une configuration de transformateurs : la mémoire, et le temps d'exécution par un processeur. Cette consommation doit être minimisée. Avec une représentation de programmes en graphes d'objets, la mémoire est consommée pour stocker les objets en mémoire, et le temps d'exécution est consommé pour la création des objets, proportionnellement à la taille des programmes à transformer. Avec une telle représentation, la taille des programmes qui peuvent être représentés et transformés est donc limitée par la taille de la mémoire disponible [LH00], même si cette représentation peut être fortement optimisée comme dans la bibliothèque de programmation ATerm [vdBdJKO00] avec notamment une factorisation de sous-graphes identiques dans un même graphe. Au contraire, avec une représentation en séquences d'interactions, il n'est pas nécessaire de stocker une représentation complète d'un programme en mémoire pour le transformer. La mémoire est utilisée seulement pour les piles d'appels, et pour l'état minimal nécessaire à chaque opération de transformation pour effectuer ses transformations. Par exemple, une opération de transformation peut avoir comme seul état l'ensemble des noms d'attributs de chaque classe en cours de transformation, afin de pouvoir créer un nouvel attribut avec un nom unique, dans chaque classe. Cependant, une telle transformation consomme moins de temps d'exécution qu'avec une représentation en graphes d'objets, car il n'est pas nécessaire de créer de nombreux objets. La représentation des programmes par des séquences d'interactions est donc la meilleure si l'on considère la consommation des ressources,

```
public class Transformateur implements ContentHandler {

  public ContentHandler traitantSuivant;

  public void startElement(String namespaceURI,
      String localName, String qualifiedName,
      Attributes attributes) throws SAXException {

    /* re-représenter l'élément XML dans le document */
    traitantSuivant.startElement(namespaceURI, localName,
      qualifiedName, attributes);

    /* implantation de la transformation :
      un nouvel élément XML est représenté */
    traitantSuivant.startElement(null, "nouvelElement",
      null, null);
    traitantSuivant.endElement(null, "nouvelElement",
      null);
  }

  public void endElement(String namespaceURI,
      String localName, String qualifiedName)
      throws SAXException {
    /* re-représenter l'élément XML dans le document */
    traitantSuivant.endElement(namespaceURI, localName,
      qualifiedName);
  }

  public void characters(char[] chars, int offset,
      int length) throws SAXException {
    /* re-représenter le contenu de l'élément XML */
    traitantSuivant.characters(chars, offset, length);
  }

  /*...*/

}
```

Prog. 3.1 : Code d'un transformateur de documents XML basé sur SAX

comme nous l'avons vérifié en pratique (cf. le chapitre 6).

D'un autre côté, les transformations complexes sont plus faciles à implanter avec une représentation par des graphes d'objets qu'avec une représentation par des séquences d'interactions. Par exemple, une transformation qui optimise l'allocation des variables dans le code des méthodes, nécessite de réaliser des analyses statiques complexes sur le flot de contrôle des méthodes. Une telle transformation est plus facile à implanter lorsqu'il est possible à un moment donné de visiter la représentation complète d'une méthode et de tous ses sous-éléments, comme cela est possible avec une représentation par graphes d'objets. Ainsi, l'utilisation d'une représentation par séquences d'interactions impose au développeur de maintenir explicitement, dans l'implantation de l'opération de transformation, un état correspondant à l'analyse du flot de contrôle de chaque méthode en cours de transformation. La représentation des programmes par des graphes d'objets est donc la plus simple lorsqu'on considère l'implantation de transformateurs qui ont un état.

En conclusion, le choix d'un mécanisme de représentation interne de programmes (objets *vs.* interactions), lors de la conception d'un système de transformation de programmes, est un compromis entre les performances, et la simplicité d'implantation des opérations de transformation.

Nous privilégions le critère de la performance des transformations, au détriment de la simplicité d'implantation, parce que Jabyce doit être utilisable pour transformer les classes pendant leur chargement par la JVM, comme décrit dans la section 3.2.3, et/ou dans des environnements contraints, et pour passer à l'échelle lors de la transformation de grands programmes. C'est pourquoi nous avons introduit la représentation des programmes par séquences d'interactions, et nous l'avons expérimentée dans ASM et Jabyce. En pratique, nous n'avons pas rencontré de grandes difficultés dans l'implantation de transformateurs, comme ceux décrits dans les chapitres 7 et 9.

3.2.5 Fabriques de programmes

La spécification dans Jabyce des interfaces des composants désérialiseurs, transformateurs et sérialiseurs, est une conséquence directe du choix du modèle conceptuel décrit dans la section 3.2.3, et du choix d'une représentation des programmes par séquences d'interactions, motivé dans la section précédente. Cet ensemble de spécifications d'interfaces est décrit dans cette section.

Pour simplifier, considérons d'abord une configuration comprenant seulement un désérialiseur et un sérialiseur directement liés, sans opérations de transformation, comme illustré dans la figure 3.5. Cette configuration reproduit les classes Java sans les transformer.

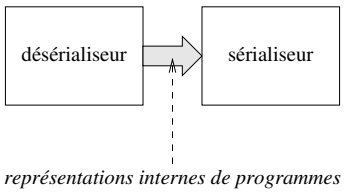


FIG. 3.5 : Configuration avec un désérialiseur et un sérialiseur

Nous utilisons la métaphore suivante : un programme est représenté comme *une séquence de demandes de création des éléments du programme*, émise vers *une fabrique de programmes*, en l'occurrence un sérialiseur qui fabrique une forme sérialisée du programme. D'après le patron de conception Fabrique Abstraite[GHJV94] (cf. l'annexe B.1), un sérialiseur est une *fabrique concrète*, et ses interfaces serveur correspondent à la spécification d'une *fabrique abstraite*. Un désérialiseur est client d'une fabrique abstraite de programmes, et a donc comme ensemble d'interfaces client l'ensemble des interfaces serveur d'un sérialiseur.

Chaque élément d'un programme, p.ex. une classe, une méthode ou une instruction, est représentée par une interaction spécifique, car nous avons défini un élément comme un granule de représentation, et donc de transformation, d'un programme. Les interfaces serveur de fabrique d'un sérialiseur doivent donc offrir une spécification d'opération de création pour chacun des 50 types d'élément que nous avons définis dans la section 3.2.3 : chaque opération doit offrir 50 méthodes de représentation d'éléments de programmes. Il reste à regrouper ces spécifications d'opérations dans des interfaces.

> Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas. [Mar96b, Mar03]

Principe 7 : Séparation des Interfaces

Nous avons appliqué le principe 7 de Séparation des Interfaces (*Interface Segregation Principle*) [Mar96b, Mar03]. Dans le cadre de la conception de systèmes de transformation de programmes, nous interprétons ce principe comme suit : les implantations de transformateurs ne doivent pas implanter de traitements sur les représentations d'éléments qui ne sont pas liés aux éléments qu'ils transforment. Par exemple, un transformateur qui ne fait que renommer des méthodes, ne doit pas être forcé de traiter les attributs des classes transformées. Dans les systèmes avec des règles déclaratives, par exemple Stratego [Vis01], ce principe est toujours appliqué car une règle est déclarée pour réécrire des éléments d'un seul type donné. Ce n'est cependant généralement pas le cas dans les systèmes avec des règles procédurales, et l'application de ce principe est une approche originale. L'application de ce principe est motivée par l'architecture interne que nous proposons pour les opérations de transformations, décrite dans la section 3.2.6.

Afin de séparer les opérations pour la représentation des différents types d'élément, nous avons donc défini dans Jabyce une interface Java de fabrique pour chaque type d'élément : ClassFactory, FieldFactory, MethodFactory, FieldAccessInstructionFactory, etc. Dans Jabyce, un sérialiseur offre donc 50 interfaces serveur, de noms class-factory, field-factory, method-factory, etc. Un désérialiseur offre 50 interfaces client des mêmes types, appelées client-class-factory, client-field-factory, client-method-factory, etc. Ces interfaces Java et ces noms d'interfaces constituent l'essentiel de la spécification du système Jabyce. Un désérialiseur et un sérialiseur sont connectés par des liaisons, une liaison pour chacune de leur 50 interfaces de fabrique, comme illustré dans la figure 3.6.

Une opération de transformation Jabyce a pour fonction de transformer la séquence des interactions initiées sur ces liaisons. Nous proposons donc qu'une opération soit un *décorateur d'une fabrique de programmes*, selon le patron de conception Décorateur [GHJV94] (cf. l'annexe B.2). En conséquence, une opération offre les 50 mêmes interfaces serveur qu'un sérialiseur,

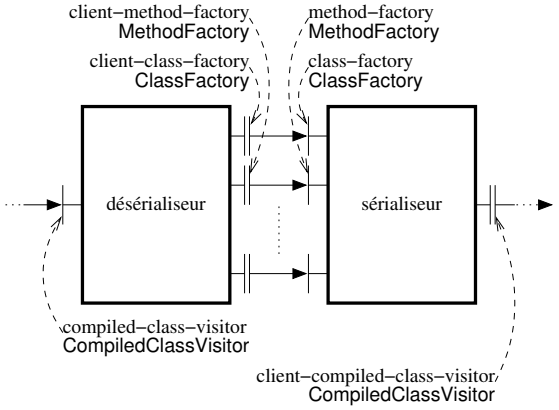


FIG. 3.6 : Configuration de composants désérialiseur et sérialiseur

et les 50 mêmes interfaces client qu'un désérialiseur. La connexion de plusieurs opérations à la suite, aboutit à une configuration de transformateurs sous la forme d'une *chaîne*. La figure 3.7 illustre une configuration de transformateurs avec un désérialiseur, deux opérations de transformation, et un sérialiseur, et les liaisons entre ces composants.

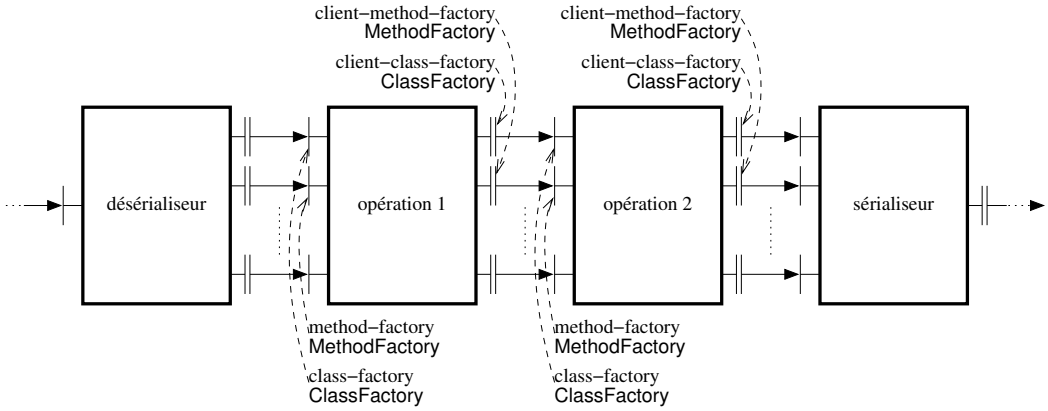


FIG. 3.7 : Chaîne de composants opérations de transformation

Les composants opération sont similaires aux objets Transformateur de documents XML que nous avons définis dans l'exemple de la section 3.2.4. La construction des liaisons entre composants opérations est similaire à la connexion des objets ContentHandler par l'affectation des attributs traitantSuivant, pour former une chaîne.

Chaque interface définit une opération pour la représentation d'éléments d'un certain type. Par exemple, le code source de l'interface Java FieldAccessInstructionFactory est donné partiellement dans le programme 3.2. Les arguments des méthodes, que nous appelons des *attributs d'éléments* (*entity fields*), servent à paramétrer la représentation de chaque élément. Ils font partie du modèle conceptuel de Jabyce. Les classes Java ObjectType et FieldDescriptor, utilisées dans la signature de la méthode createFieldAccessInstruction, et d'autres classes similaires, sont définies dans Jabyce pour identifier des types Java utilisés dans les représentations d'éléments. L'objet MethodBuildingContext identifie la méthode dans laquelle l'instruction doit être

représentée.

```
public interface FieldAccessInstructionFactory {
  /**
   * Creates the instruction that accesses a field
   * with the specified fields.
   *
   * @param compositeMethodBuildingContext
   *    the context for the building of the composite
   *    method that will contain the created
   *    instruction that accesses a field.
   * @param type the type of the target object that
   *    defines the accessed field.
   * @param fieldName the name of the accessed field.
   * @param fieldIsStatic the flag that indicates that if
   *    true, the field is static.
   * @param fieldType the accessed field type.
   * @param loadOrStore the flag that indicates that if
   *    true, the field is loaded; otherwise it is stored.
   * @pre compositeMethodBuildingContext
   *    != null
   * @pre type != null
   * @pre fieldName != null
   * @pre fieldType != null
   */
  void createFieldAccessInstruction(
    MethodBuildingContext compositeMethodBuildingContext,
    ObjectType type, String fieldName,
    boolean fieldIsStatic, FieldDescriptor fieldType,
    boolean loadOrStore);
}
```

PROG. 3.2 : Interface FieldAccessInstructionFactory

Nous avons introduit le concept *d'objets de contexte de construction* (*building context objects*) pour identifier les éléments composites lors de la représentation d'éléments à l'intérieur. Nous avons défini dans Jabyce les interfaces Java ClassBuildingContext, FieldBuildingContext et MethodBuildingContext. De tels objets sont retournés par les opérations de représentation des éléments composites. Par exemple, le code source de l'interface Java MethodFactory est donné en partie dans le programme 3.3. La fin de la représentation d'un élément composite doit être notifiée explicitement par l'appel de la *méthode de terminaison de construction* correspondante. Il n'y a en effet pas d'autre moyen de déterminer la fin de la représentation d'un élément composite. Par exemple, lorsque tous les éléments ont été représentés dans une méthode, en utilisant son objet de contexte en argument, la méthode finishMethodBuilding doit être appelée avec cet objet de contexte.

Ce mécanisme permet de représenter pour une classe Java l'arbre dont les nœuds sont les éléments formant la classe et les arcs les relations de composition, sous la forme d'une séquence d'interactions bien délimitée, correspondant à un parcours de l'arbre. Les opérations de terminaison de construction sont appelées lors de la « remontée dans l'arbre ». Cette manière de représentée les graphes d'éléments, dans Jabyce, est illustrée dans la section 5.3.

```
public interface MethodFactory {

  MethodBuildingContext createMethod(
    ClassBuildingContext
      compositeClassBuildingContext, ...);

  void finishMethodBuilding(
    MethodBuildingContext buildingContext);

  /* ...  */

}
```

PROG. 3.3 : Interface MethodFactory

Jabyce n'impose généralement pas d'ordre pour la représentation des éléments, c'est-à-dire pour le « parcours » des graphes modélisant des classes Java compilées. Par exemple, les attributs et les méthodes d'une classe peuvent être représentés dans n'importe quel ordre. De plus, les séquences d'interactions représentant des éléments composites peuvent être entrelacées. Par exemple, les séquences d'interactions représentant plusieurs classes peuvent être entrelacées sur les interfaces d'une même opération. Cela permet notamment à une opération de représenter une nouvelle classe alors qu'elle transforme une séquence d'interactions représentant une autre classe. Jabyce définit cependant certaines contraintes sur l'ordre des interactions, qui sont décrites dans le chapitre 4, pour garantir la correction syntaxique et sémantique des programmes produits. Par exemple, dans la représentation d'une méthode, Jabyce impose que les variables locales soient représentées avant la représentation d'une instruction qui l'utilise, et que les arguments formels soient représentés avant la déclaration de variables locales initialisées avec leur valeur.

Il est à noter que notre application du patron de conception Fabrique Abstraite n'est pas orthodoxe, parce qu'à part pour les éléments composites, les méthodes que nous avons définies ne retournent pas de valeur. En effet, *la seule sémantique d'une invocation d'une telle méthode est d'être une représentation interne d'un élément d'un programme.* Pour simplifier la compréhension de Jabyce, nous aurions pu définir par exemple une méthode fieldAccessInstructionFactory(...) au lieu de createFieldAccessInstructionFactory(...). Cependant, pour éviter des conflits de noms de méthodes, nous avons été contraints d'ajouter un préfixe à tous les noms de méthodes, ce qui a motivé l'application du patron de conception Fabrique Abstraite et le choix du préfixe create pour les noms de méthodes. Pour éviter toute ambiguïté, dans la suite de ce document nous ne faisons plus référence à ce patron de conception, et nous désignons ces invocations de méthodes comme des représentation internes d'éléments de programmes et pas comme des créations d'éléments.

3.2.6 Conception d'opérations de transformation

La section précédente a décrit l'architecture externe d'une opération de transformation Jabyce, telle que définie dans Jabyce. Cette section décrit une proposition d'architecture interne d'une opération, ainsi que des principes de conception d'une opération. Cette architecture n'est pas

imposée par Jabyce, mais facilite en pratique la conception d'une opération.

L'opération de transformation la plus simple possible est une opération qui n'implante aucune transformation, que nous appelons une *opération vide*. Nous définissons une opération vide comme un composant Fractal composite sans sous-composants, dont toutes les interfaces client internes sont directement liées aux interfaces serveur internes, comme illustré dans la figure 3.8. Comme une opération vide n'intercepte ni ne décore aucune interaction, les interactions qu'elle reçoit sur ses interfaces serveur, qui représentent des éléments programmes, sont réémises à l'identique sur ses interfaces client.

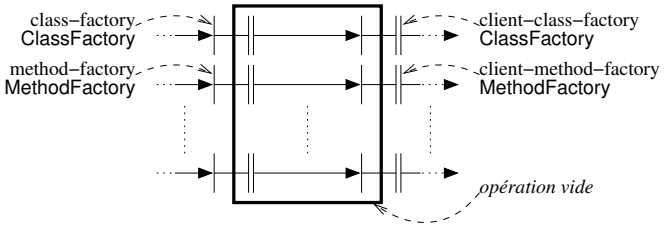


FIG. 3.8 : Architecture d'une opération de transformation vide

Nous proposons de concevoir toute opération de transformation comme une *extension d'une opération vide*, qui contient au moins un *composant transformateur* lié à une ou plusieurs de ses interfaces internes. La figure 3.9 illustre une opération de transformation qui contient un transformateur. Ce transformateur intercepte les interactions représentant des éléments de programmes d'un certain type, reçues sur une des interfaces serveur, et représente des éléments de deux types différents en utilisant deux interfaces client. Toutes les interfaces client internes non liées à un transformateur restent directement liées aux interfaces serveur internes correspondantes, les interactions reçues sur ces interfaces n'étant pas transformées.

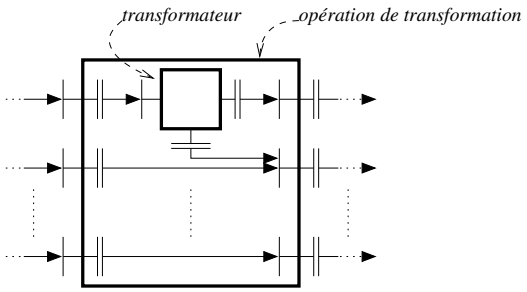


FIG. 3.9 : Architecture d'une opération de transformation

Dans Jabyce, les concepts d'opération et de transformateur sont complémentaires, et correspondent à des objectifs et des principes de conception différents. Du point de vue d'un développeur de composant transformateur, la spécification de l'architecture interne d'une opération se base sur l'utilisation des outils associés au langage de description d'architecture (ADL) de Fractal. Jabyce fournit la définition ADL d'une opération vide (EmptyOperation). Pour définir l'architecture d'une opération, un développeur doit alors seulement fournir une définition ADL d'un composant composite qui étend la définition EmptyOperation, qui ajoute un sous-composant transformateur dans ce composite, et crée des liaisons entre certaines

interfaces internes du composite et celles du sous-composant. Par exemple, la figure 3.10 donne la définition ADL TraceOperation d'une opération contenant un transformateur implanté par la classe TraceTransformer. Ce transformateur ajoute au début de chaque méthode du code qui affiche un message sur la console. Le sous-composant transformateur est appelé trace-transformer. Il offre une interface serveur method-factory, pour recevoir les interactions représentant des méthodes, et des interfaces client client-method-factory, client-method-call-instruction-factory, etc. pour représenter des éléments transformés ou de nouveaux éléments. Ces interfaces sont liées aux interfaces internes correspondantes du composant opération composite (balises <binding/>). La déclaration d'une liaison avec une interface client interne du composite a pour effet de supprimer la liaison déjà définie sur cette interface dans la définition qui est étendue (ici, EmptyOperation). L'architecture de cette opération est similaire à celle illustrée dans la figure 3.9.

```
<definition name="...TraceOperation"
    extends="...EmptyOperation">
  <component definition="...TraceTransformer"
    name="trace-transformer"/>
  <binding client="this.method-factory"
    server="trace-transformer.method-factory"/>
  <binding client="trace-transformer.client-class-factory"
    server="this.client-class-factory"/>
  <binding client="trace-transformer.client-method-factory"
    server="this.client-method-factory"/>
  <binding client=
"trace-transformer.client-method-call-instruction-factory"
    server="this.client-method-call-instruction-factory"/>
  ...
</definition>
```

Fig. 3.10 : Définition ADL d'une opération d'ajout de trace d'exécution

Un composant transformateur offre des interfaces serveur seulement pour les types des éléments qu'il transforme, et des interfaces client seulement pour les types des éléments qu'il représente dans les programmes transformés. De cette manière, l'implantation d'un composant transformateur ne concerne que les éléments réellement transformées, ce qui simplifie cette implantation. C'est pour permettre cette simplification que nous avons appliqué le principe 7 de Séparation des Interfaces pour la conception des interfaces, comme décrit dans la section 3.2.5.

Il peut exister des *dépendances causales entre des transformations*. Par exemple, renommer une méthode impose de transformer également toutes les instructions d'appel à cette méthode. En application du principe 8 de Fermeture Commune (*Common Closure Principle*) [Mar96a, Mar03], nous spécifions que *les transformations dépendantes causalement doivent être implantées dans le même transformateur*, de manière à être toujours mises en œuvre conjointement. Nous définissons donc une implantation de transformateur comme *le granule de réutilisation d'implantations de transformations*. Il n'existe pas de moyen général pour contraindre l'application de ce principe, car les dépendances entre transformations sont liées à leur sémantique. Par exemple, un transformateur peut encapsuler l'implantation d'une transformation qui renomme une méthode, et l'implantation d'une transformation qui génère une

nouvelle méthode avec le nom et la signature originaux de cette méthode, sans avoir besoin de transformer les instructions d'appel à la méthode originale. Le problème des dépendances sémantiques entre transformations est décrit plus précisément dans la section 4.1.2.

> Les parties d'un logiciel impactées par les mêmes changements doivent être placées dans un même module. [Mar96a, Mar03]

Principe 8 : Fermeture Commune

La séparation des interfaces, et l'architecture que nous proposons, permettent d'appliquer automatiquement des optimisations. Julia, l'implantation de référence de Fractal en Java [Objb, BCL+04], permet notamment d'optimiser automatiquement les liaisons avec les interfaces externes de composants composites. Ainsi, la configuration de composants illustrée dans la figure 3.9 peut être automatiquement optimisée pour être similaire à la configuration illustrée dans la figure 3.11 : les interactions ne sont plus interceptées par la membrane des composites. Une configuration complète d'opérations Jabyce peut ainsi être optimisée, ce qui contribue à l'application de notre principe 6 de Composition Efficace des Transformations. Une configuration optimisée n'est plus reconfigurable pendant l'exécution, mais ce n'est généralement pas un problème pour les systèmes de transformation de programme. Grâce à ces optimisations automatiques, notre introduction du concept d'opération a un impact nul sur les performances de transformation.

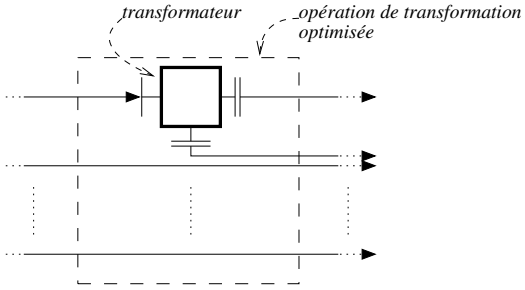


Fig. 3.11 : Opération de transformation optimisée par Julia

Le concept d'opération de transformation est indispensable pour simplifier la composition des transformateurs, en considérant que toutes *les opérations sont homogènes*. Les opérations ont toutes exactement les mêmes interfaces externes, et sont liées par les mêmes ensembles de liaisons, bien que leur contenus puissent être différents. C'est une application du principe 9 d'Ouverture-Fermeture (*Open-Closed Principle*) [Mey97]. Les opérations sont fermées car la spécification de leurs interfaces est stable, indépendante de transformations particulières, et bien définie dans Jabyce, et ouvertes car elles peuvent contenir chacune une configuration quelconque de transformateurs.

Les composants Fractal opération et transformateur peuvent offrir des interfaces en plus des interfaces permettant de représenter des éléments d'un programme. Par exemple, un composant transformateur qui implante une transformation d'ajout d'un préfixe au nom de tous les attributs des classes, peut offrir une interface d'attribut pour paramétrer le préfixe à utiliser. Un même transformateur peut déléguer la décision du renommage de chaque attribut à un

> Tout module doit être à la fois ouvert et fermé. Un module est dit *ouvert* (*aux extensions*) s'il est possible de l'étendre. Un module est dit *fermé* (*aux modifications*) si sa description est stable et bien définie, et s'il est disponible pour être utilisé par d'autres modules. [Mey97]

Principe 9 : Ouverture-Fermeture

autre composant, à travers une interface client spécifique de ce composant, en appliquant le patron de conception Stratégie [GHJV94] (cf. l'annexe B.3). Nous introduisons le concept de *règle procédurale paramétrée*, pour désigner de tels composants. L'utilisation d'un modèle de composant général comme Fractal offre ainsi une grande expressivité dans l'implantation des transformations, en comparaison des langages de règles déclaratives.

3.3 Généralisation de l'approche

Notre démarche de conception du système de transformation de programme Jabyce, décrite dans les sections précédentes, est reproductible et partiellement automatisable avec des outils logiciels, pour construire tout système de transformation 1) basé sur le modèle de composant Fractal, 2) avec une représentation des programmes par séquences d'interactions, et 3) qui applique nos principes 5 et 6 d'Encapsulation et de Composabilité des Transformations, et de Composition Efficace des Transformations. Cette section présente Jaidee[2], un logiciel qui génère la majeure partie du code source de tels systèmes de transformation, à partir de spécifications de modèles conceptuels de représentations internes de programmes. Nous avons notamment conçu Jaidee pour générer le code source de Jabyce. Jaidee n'est cependant pas limité à la construction de systèmes de transformation de programmes, et pourrait par exemple être utilisé pour la construction d'un système de transformation de documents LaTeX.

3.3.1 Génération automatique de systèmes de transformation

La principale différence entre les différents systèmes de transformation qui peuvent être générés avec Jaidee, est le modèle conceptuel des éléments qui peuvent être représentés par les systèmes dans les représentations internes, comme par exemple le modèle conceptuel des classes

[2]Dans la langue thaï, « jaidee » est un adjectif qui signifie « gentil ». La prononciation du nom « Jaidee » est la même qu'en thaï, et est notée [ˈdʒæɪdiː] en phonétique, c'est-à-dire que « ja » est prononcé comme dans le mot anglais « jacket », et « idee » comme le mot anglais « ID » avec le « dee » final prononcé comme dans « deejay ».

Java compilées défini pour Jabyce dans la section 3.2.3. Chaque modèle conceptuel est traduit en un ensemble de définitions d'interfaces Java offertes par les composants (dé)sérialiseurs et opérations, comme les interfaces de Jabyce définies dans la section 3.2.5. Nous avons développé Jaidee pour automatiser la génération du code source de ces interfaces Java, entre autres, à partir d'une définition formelle d'un modèle conceptuel. Jaidee génère aussi 1) le code source des interfaces des objets de contexte de construction d'éléments composites (par exemple ClassBuildingContext, etc. dans Jabyce), 2) le code source de classes de base implantant ces interfaces, et 3) le code source de classes abstraites facilitant l'implantation de composants transformateurs (par exemple AbstractFieldAccessInstructionTransformer). Un modèle conceptuel est donné dans un fichier XML, qui spécifie :

- le nom des types d'élément, avec une description textuelle.
- les relations de composition entre types d'élément.
- les relations de dépendance entre types d'élément.
- les attributs (nom, type et description) qui constituent chaque type d'élément.
- des contraintes, sous forme d'assertions, sur les attributs de chaque type d'élément.

Par exemple, la définition du type d'élément FieldAccessInstruction de Jabyce est donnée dans la figure 3.12. Pour Jabyce, une telle définition est donnée pour chacun des 50 types d'élément définis pour Jabyce, cf la section 3.2.3. Les éléments de texte sont utilisés pour générer la documentation Javadoc dans les fichiers source, comme illustré dans le programme 3.2 pour l'interface FieldAccessInstructionFactory. Les attributs (*fields*) sont utilisés pour définir la signature des méthodes de représentation d'éléments.

Les éléments XML <depTargetEntityType/> expriment des dépendances entre types d'élément. Par exemple, la dépendance du type FieldAccessInstruction vers le type Field, définit qu'un transformateur d'attributs d'objets ou de classes, doit généralement être aussi un transformateur d'instruction d'accès à des attributs. En effet, ces relations entre types d'élément expriment donc des *dépendances causales* entre transformations. Comme expliqué dans la section 3.2.6, il est souhaitable d'encapsuler dans un même transformateur des transformations causalement dépendantes, bien qu'il n'existe pas de moyen général d'imposer ce principe. Cependant, afin de *guider la conception* des transformateurs, Jaidee génère le code de classes Java abstraites pour l'implantation de transformateurs. Ces classes abstraites déclarent implanter les interfaces de fabrique d'éléments en fonction des dépendances entre types d'élément. Par exemple, la classe abstraite AbstractFieldTransformer implante l'interface FieldFactory, mais aussi l'interface FieldAccessInstructionFactory. Ainsi, tout transformateur d'attributs implanté en étendant cette classe abstraite doit obligatoirement encapsuler les transformations dépendantes.

Les assertions (*validity test expressions*) sont utilisées pour générer des pré- et post-conditions, sous la forme d'éléments @pre et @post dans la documentation Javadoc. Par exemple, dans la définition de FieldAccessInstruction, l'assertions {1} != null spécifie que l'attribut numéro 1, c'est-à-dire l'attribut contenant le nom de l'attribut de classe Java accédé, ne doit pas être null. Cette assertion est traduite dans le code source de FieldAccessInstructionFactory par le commentaire : @pre fieldName != null. Ces assertions en commentaires sont traitées par l'outil iContract [Kra98, Kra] pour insérer automatiquement, dans le code source des classes Java, du code qui vérifie que ces assertions sont vraies pendant l'exécution. Cette approche de conception par contrats (*Design by Contract*) [Mey97] nous permet de donner une sémantique forte à la définition des éléments représentés, et de rendre encore plus stables et abstraites

```
<entityType name='FieldAccessInstruction'>

  <singularTextName>
    instruction that accesses a field
  </singularTextName>
  <pluralTextName>
    instructions that access fields
  </pluralTextName>
  <compositeEntityType name='Method'/>
  <depTargetEntityType name='Field'/>

  <physicoLogicalField index='0'>
    <name name='type'/>
    <javaType name=
      'com.francetelecom.rd.jabyce.model.api.type.ObjectType'/>
    <text>type of the target object
      that defines the accessed field</text>
  </physicoLogicalField>
  <physicoLogicalField index='1'>
    <name name='fieldName'/>
    <javaType name='java.lang.String'/>
    <text>name of the accessed field</text>
  </physicoLogicalField>
  <physicoLogicalField index='2'>
    <name name='fieldIsStatic'/>
    <javaType name='boolean'/>
    <text>flag that indicates that if true,
      the field is static</text>
  </physicoLogicalField>
  <physicoLogicalField index='3'>
    <name name='fieldType'/>
    <javaType name='...jabyce.model.api.type.FieldDescriptor'/>
    <text>accessed field type</text>
  </physicoLogicalField>
  <physicoLogicalField index='4'>
    <name name='loadOrStore'/>
    <javaType name='boolean'/>
    <text>flag that indicates that if true,
      the field is loaded; otherwise it is stored</text>
  </physicoLogicalField>

  <validityTestExpression>{0} != null</validityTestExpression>
  <validityTestExpression>{1} != null</validityTestExpression>
  <validityTestExpression>{3} != null</validityTestExpression>

</entityType>
```

FIG. 3.12 : Description avec Jaidee du type d'entité FieldAccessInstruction

les interfaces des systèmes de transformation générés. Cette approche renforce l'application du principe 9 d'Ouverture-Fermeture à la conception des opérations de transformation (cf. la section 3.2.6). Nos propositions complémentaires et les perspectives pour cette problématique, à l'intersection des domaines de la transformation de programmes et de la sûreté de fonctionnement, sont discutées plus en détails dans le chapitre 4.

Jabyce est pour l'instant le seul système de transformation développé avec Jaidee. 165 des fichiers source de Jabyce sont générés automatiquement par Jaidee, sur 226 fichiers source. Les 61 fichiers codés manuellement sont les types utilisés dans les signatures de méthodes dans les interfaces, par exemple ObjectType, des implantations de composants désérialiseur et sérialiseur, et des implantations de composants pour l'accès à des espaces de stockage de classes Java compilées (*repositories*). L'utilisation de Jaidee dans la construction de Jabyce a démontré en pratique que Jaidee minimise l'effort de conception et d'implantation d'un tel système de transformation.

3.3.2 Systèmes de traduction de programmes

Un système de transformation comme Jabyce, utilisé isolément, permet uniquement l'implantation de transformations de *réexpression* (*rephrasing*), c'est-à-dire des transformations d'un modèle conceptuel vers le même modèle. Par exemple, Jabyce utilisé seul permet d'implanter seulement des transformations de classes Java compilées vers des classes Java compilées, dans le même modèle conceptuel. La généralisation de notre approche de conception permet cependant de concevoir facilement des systèmes de *traduction*, en combinant deux systèmes de transformation générés par Jaidee. Cette approche nécessite seulement d'implanter un *composant traducteur*, qui offre les interfaces serveur pour le modèle conceptuel source, et les interface client pour le modèle conceptuel cible, et qui transforme les séquences d'interactions représentant ces éléments, entre ces interfaces.

Par exemple, la figure 3.13 illustre une configuration de transformateurs, qui combine une configuration de transformateurs d'un langage imaginaire d'expressions arithmetiques et logiques, et une configuration de transformateurs Jabyce, reliées par un traducteur qui compile des expressions arithmétiques en classes Java compilées.

Ainsi, Jabyce utilisé seul ne permet d'implanter que des transformations de réexpression de classes Java compilées. Mais Jabyce peut être intégré avec tout autre système de transformation pour construire tout système de transformation manipulant des classes Java compilées comme modèle conceptuel source ou comme modèle conceptuel cible. Ainsi, le chapitre 9 décrit Jivaro, un traducteur de code compilé Java vers du code source C, qui est basé sur Jabyce.

3.4 Conclusion

Les deux caractéristiques originales du système de transformation Jabyce sont : 1) la conception à l'aide d'un modèle de composant logiciel général (Fractal), et 2) la représentation interne des éléments de programme par des interactions [LCB04].

En ce qui concerne l'architecture d'une composition de transformateurs, nous avons distingué différents types de composants : 1) les désérialiseurs qui convertissent les formes sérialisés de

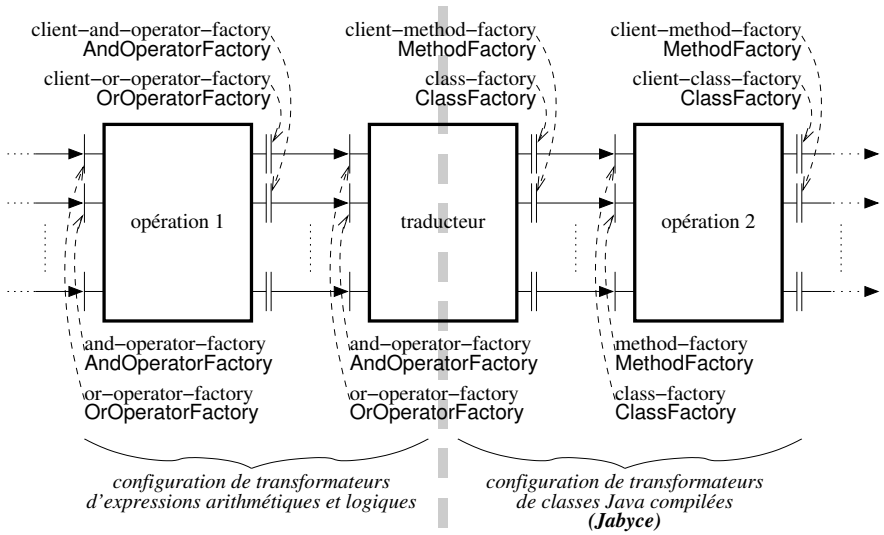


FIG. 3.13 : Système de traduction d'expressions arithmétiques et logiques en classes Java compilées

programmes en séquences d'interactions représentant les éléments de programmes, 2) les sérialiseurs qui effectuent une conversion dans l'autre sens, et 3) les opérations de transformation. Les opérations de transformation sont des composants tous similaires, en ce qui concerne les interfaces offertes par ces composants. Nous proposons une démarche de conception de ces composants opération, en les considérant comme des composants composites qui contiennent des composants transformateur qui transforment effectivement les représentations internes de programmes. Cette architecture interne « creuse » des composants opération permet de mettre en œuvre automatiquement des optimisations, afin éviter d'effectuer des traitements inutiles pour les éléments de programmes non transformés.

L'utilisation du modèle de composant Fractal offre directement l'avantage de pouvoir spécifier des compositions d'opérations de transformation (*stratégies de transformation*) de manière très facile sous la forme de liaisons entre composants. Notamment, il est possible d'utiliser le langage ADL (*Architecture Description Language*) comme un langage spécifique de haut niveau, pour décrire de telles compositions. De plus, les composants opérations et transformateurs sont facilement directement réutilisables et composables.

Le système de transformation de programmes Java compilés Jabyce est un *canevas logiciel* (*framework*). Un canevas logiciel est défini comme *« une conception réutilisable de tout ou partie d'un système, qui est représentée par un ensemble de classes abstraites et par la manière dont leurs instances interagissent »* [Joh97], et est décrit comme *« liant certains choix à propos du partitionnement de l'état et du flot de contrôle ; l'utilisateur complète ou étend le canevas logiciel pour produire une application »* [Deu89]. Les parties abstraites réutilisables qui forment Jabyce sont les interfaces Java qui spécifient les interfaces des composants opération, des sérialiseurs et des désérialiseurs, et les classes Java abstraites et utilitaires utilisées pour l'implantation des transformateurs. Jabyce est également une spécification de la manière de composer les composants, et une spécification des programmes Java pouvant être représentés par des séquences d'interactions entre des composants. Jabyce est étendu par un utilisateur en

développant de nouvelles implantations de composants transformateurs, désérialiseurs et/ou sérialiseurs, et en définissant de nouvelles configurations de composants.

Dans la perspective de notre démarche générale introduite dans la section 2.4, nous considérons qu'un canevas logiciel comme Jabyce définit des abstractions logicielles stables, en l'occurrence des interfaces Java, semblables aux « boutons de réglage » d'un concept. Les implantations possibles de ces interfaces, c'est-à-dire les transformations implantables avec Jabyce, forment « l'implicosphère » de ce canevas logiciel, c'est-à-dire l'ensemble des variations possibles. Nous avons conçu Jabyce de manière à obtenir une grande implicosphère de ce canevas, c'est-à-dire à permettre un large spectre de transformations de programmes implantables, conformément à notre principe 4 de Généralité du Système de Transformation. Il est à noter que tous les canevas logiciels de transformations générés par notre logiciel Jaidee, tels que Jabyce, ont tous des abstractions logicielles similaires, et offrent tous une représentation interne des programmes par des séquences d'interactions, et non pas par des graphes d'objets. Ces canevas logiciels ont donc tous en commun un même squelette conceptuel.

Chapitre 4

Sûreté de fonctionnement

Dans ce chapitre, nous abordons le problème de la sûreté de fonctionnement, uniquement en ce qui concerned les systèmes de transformation de programmes complexes tels que Jabyce. Nous n'abordons pas le problème général de la sûreté de fonctionnement dans les systèmes à composants indépendamment de ce contexte, bien que certains aspects de notre réflexion puissent être généralisés. Nous faisons référence à [Lap96], pour les concepts et la terminologie.

Nous nous plaçons ici principalement dans le cadre du *test des transformateurs*. Globalement, notre objectif est de vérifier, pendant l'exécution des transformations de programmes, que les transformations produisent seulement des programmes corrects. La définition de la correction des programmes est celle donnée pour les vérifications effectuées par le vérificateur de programme (*verifier*) d'une JVM [LY99]. Il est important de prévenir l'occurrence de telles erreurs pendant la transformation de programmes, parce qu'elles peuvent se traduire par l'introduction de fautes dans les programmes transformés, qui peuvent causer des erreurs pendant le chargement de ces programmes par une JVM, ou pendant leur exécution. La sûreté de fonctionnement des transformations de programmes a donc un impact sur la sûreté de fonctionnement des programmes transformés.

L'avantage de vérifier la correction des programmes pendant la transformation, au lieu de déléguer cette vérification au vérificateur d'une JVM, est de permettre un *diagnostic* précis des fautes dans les transformateurs à l'origine de l'incorrection d'un programme. De plus, nous envisageons d'utiliser notre proposition pour implanter un vérificateur dans la plateforme Java Jivaro, présentée dans le chapitre 9, dont l'environnement d'exécution n'implante pas de vérificateur. Le système de vérification que nous proposons dans ce chapitre a donc un intérêt, en complément du vérificateur d'une JVM.

La section 4.1 décrit notre approche globale. Nous avons séparé les problèmes liés à l'architecture du système de vérification de contrats de comportement, et ceux liés à la spécification des contrats. Cette séparation est une application du principe de séparation entre politiques et mécanismes, comme dans l'approche des langages spécifiques (DSL), cf. la section 1. La section 4.2 décrit notre proposition d'architecture pour la vérification, et la section 4.3 décrit les différents formalismes de spécification de contrats avec lesquels nous avons expérimenté.

4.1 Approche générale

Dans cette section, nous abordons l'intersection entre les domaines de la sûreté de fonctionnement, et de la transformation de programme. Selon notre démarche générale introduite dans la section 2.4, nous explorons les variations dans le domaine de la sûreté de fonctionnement, qui sont intéressantes dans le cadre de la transformation de programme. Cette exploration est réalisée selon les trois « classes de notions et concepts » introduites dans [Lap96] : attributs (utilité de la sûreté de fonctionnement) dans la section 4.1.1, entraves (fautes, erreurs, défaillances) dans la section 4.1.2, et moyens dans la section 4.1.3. Dans chaque classe, nous définissons les concepts intéressants, et les choix que nous faisons dans notre contexte, concernant par exemple le modèle des fautes. La section 4.2 décrit l'architecture que nous proposons pour la vérification pendant l'exécution des transformations de programme.

4.1.1 Intégrité des programmes

Les différents *attributs de la sûreté de fonctionnement* sont : la disponibilité, la fiabilité, la sécurité-innocuité, la confidentialité, l'intégrité et la maintenabilité. Nous n'abordons ici que l'attribut d'*intégrité*, parce qu'il est un pré-requis à d'autres attributs (disponibilité, fiabilité, sécurité-innocuité et confidentialité), et parce que c'est le seul que nous avons identifié comme pertinent pour les systèmes de transformation de programmes. L'intégrité est définie comme la non-occurrence d'altérations inappropriées de l'information. Dans notre contexte, nous considérons les représentations internes de programmes transformés, communiquées entre les composants d'une configuration de transformateurs. Nous définissons ici l'intégrité comme la restriction des programmes représentables, ou *autorisés*, à l'ensemble des programmes corrects. Notre définition de la correction des programmes est celle donnée pour les vérifications effectuées par le vérificateur de programme (*verifier*) d'une JVM [LY99]. Ainsi, les programmes corrects sont ceux qui seraient validés par un vérificateur d'une JVM, pendant le chargement de ces programmes.

Nous considérons les composants comme des boîtes noires, et seules sont observables les interactions entre les composants. Dans Jabyce, ce sont justement les interactions entre les composants qui représentent les programmes transformés. Nous proposons ici des moyens de mise en œuvre de *politiques d'autorisation des interactions [Lap96]* qui représentent des programmes, dans Jabyce, pour garantir que seulement des programmes corrects peuvent être représentés.

4.1.2 Modèle d'erreur et de faute

Les *entraves à la sûreté de fonctionnement* sont : les défaillances, les erreurs et les fautes. Cette section décrit le modèle des erreurs et des fautes que nous considérons dans le contexte de la transformation de programme. Une *défaillance* survient lorsque le système fournit un service qui dévie de la fonction du système. Dans notre contexte, la fonction d'une configuration d'opérations Jabyce est de *transformer correctement des programmes* Java compilés. Plus précisément, si une opération Jabyce reçoit une séquence d'interactions représentant un programme correct sur ses interfaces serveur, elle doit initier une séquence d'interactions représentant également un programme correct sur ses interfaces client. Une *erreur* est la partie

de l'état du système qui est susceptible d'entraîner une défaillance : une erreur affectant le service est une indication qu'une défaillance survient ou est survenue. La cause adjugée ou supposée d'une erreur est une *faute*.

Modèle d'erreur

Dans notre contexte, nous considérons seulement les erreurs consistant en l'occurrence d'une interaction, dans une séquence d'interactions, qui conduit à la représentation d'un programme incorrect. Nous avons défini deux niveaux d'abstraction pour la définition de *contraintes de correction* des programmes Java compilés, et dont les violations correspondent à des erreurs de natures différentes :

- syntaxe des programmes ;
- sémantique des programmes.

Ces deux niveaux correspondent aux différentes passes de vérification réalisées par un vérificateur d'une JVM. L'algorithme du vérificateur est appelé *processus de vérification* (*verification process*), et est exécuté au chargement de chaque classe Java. L'algorithme comprend quatre passes : la passe 1 concerne le format des classes compilées, et les trois passes suivantes concernent la sémantique. Notre niveau de la syntaxe des programmes correspond essentiellement à la passe 1 du vérificateur, et notre niveau de la sémantique des programmes correspond aux passes 2, 3 et 4. Le critère que nous avons choisi pour distinguer plus clairement ces deux niveaux est la considération, ou non, de relations entre différents éléments de classes Java compilées.

Syntaxe des programmes Au niveau de la *syntaxe des programmes*, doivent être vérifiées des contraintes de correction qui ne concernent pas des relations entre différents éléments de classes Java compilées. Ainsi, chaque classe Java compilée, dans sa forme sérialisée, doit respecter le format défini dans la spécification de la machine virtuelle [LY99]. Par exemple, chaque fichier .class doit commencer par les quatre octets encodés en hexadécimal par 0xcafebabe. De même, les instructions ont une structure bien définie dans la spécification, par exemple l'instruction iload a pour structure son *opcode* (octet 0x15) suivi de l'index de la variable locale sur un octet qui doit être ≥ 0. Dans Jabyce, de telles contraintes de correction sont principalement vérifiées par les implantations de (dé)sérialiseurs. De plus, dans Jabyce nous avons choisi un modèle conceptuel de classes Java compilées qui abstrait de nombreux détails du format des classes compilées (cf. la section 3.2.3), ce qui limite les erreurs liées au format. Par exemple, dans Jabyce les index de variables locales ne sont pas manipulables par les transformations.

En revanche, même si les détails du format sont masqués, les contraintes sur le format se transposent en des contraintes de syntaxe, à un plus haut niveau, sur les représentations internes de programmes. Par exemple, lors de la représentation d'une instruction d'accès à un attribut dans Jabyce, par un appel à la méthode createFieldAccessInstruction de l'interface Java FieldAccessInstructionFactory illustrée dans le programme 3.2, le paramètre type ne doit pas être null. Cette contrainte de syntaxe est une transposition des contraintes de la spécification de la JVM sur la structure des instructions getfield, putfield, getstatic et putstatic.

Sémantique des programmes Au niveau de la *sémantique des programmes*, doivent être vérifiées des contraintes portant sur des relations entre les éléments formant une ou plusieurs classes Java compilées. Ces contraintes peuvent porter, par exemple, sur des éléments dans des classes différentes. Ces contraintes sont celles définies pour les passes 2, 3 et 4 du vérificateur de la JVM.

La passe 2 vérifie par exemple qu'il n'existe pas de sous-classe d'une classe déclarée final. La passe 3 vérifie des contraintes statiques et structurelles, par une analyse statique des implantations de méthodes. Par exemple, l'algorithme vérifie qu'une variable locale ne peut pas être lue avant d'avoir été affectée à une valeur. La passe 4 est en réalité réalisée après le chargement des classes, à l'exécution des instructions par la JVM. Par exemple, l'exécution d'une instruction d'appel de méthode vérifie que l'objet cible de l'appel n'est pas null. Cependant, certaines des contraintes de la passe 4 peuvent être vérifiées avant l'exécution. Par exemple, la vérification que la méthode appelée par une instruction d'appel doit exister et être accessible, peut être réalisée au chargement de la classe appelante. La délégation de ces vérifications à l'exécution permet à la JVM de réaliser un chargement paresseux des classes, en n'imposant pas de charger toutes les classes à la fois pour la vérification. En ce qui concerne les contraintes sur la correction des programmes dans un système de transformation, nous considérons donc seulement les contraintes vérifiées dans les passes 2 et 3, et les contraintes vérifiées dans la passe 4 qui peuvent être vérifiées statiquement.

Synthèse Nous considérons seulement deux types d'erreurs, qui correspondent aux deux niveau de correction de programmes définis ci-dessus : 1) erreurs de correction syntaxique des programmes, et 2) erreurs de correction sémantique des programmes. Comme indiqué au début de cette section, notre critère pour distinguer ces deux niveaux de correction de programme, et donc ces deux types d'erreurs, est la considération, ou non, de relations entre différents éléments d'un programme. C'est-à-dire que nous considérons que la détection d'une erreur de correction sémantique nécessite de considérer plusieurs éléments d'un programme représenté, mais ce n'est pas le cas pour la détection d'une erreur de correction syntaxique. Cette distinction concerne donc la facilité de la détection des erreurs. En effet, la détection d'erreurs de correction sémantique nécessite de *conserver un historique des interactions* représentant des éléments, ou au moins une synthèse d'un tel historique. Cette caractéristique a un impact sur les formalismes de spécification utilisés pour définir les contraintes de correction sémantique. Un formalisme doit offrir un moyen de raisonner sur un historique des interactions passées, pour pouvoir être utilisé dans notre contexte. Cette problématique est abordée plus en détails dans la section 4.3.1.

Des erreurs d'autres types peuvent se produire pendant l'exécution de transformations, qui se manifestent généralement par des levées d'exceptions Java. Par exemple, un appel de méthode avec une référence d'objet null, est une erreur qui se traduit par le lancement par la JVM d'une exception de type NullPointerException. Cependant, de telles erreurs ne sont pas spécifiques à notre contexte de la sûreté de fonctionnement des transformations de programme. Nous ne considérerons donc pas dans la suite le traitement de telles erreurs générales.

Modèle de faute

Notre modèle de faute considère les causes des erreurs de correction syntaxique et sémantique des programmes produits par des transformateurs. Nous introduisons et discutons ici particulièrement d'une catégorie particulière de fautes, celles liées à la composition des transformateurs, parce qu'elles font en pratique l'objet de nombreuses questions de la part des utilisateurs de systèmes de transformation de programmes (« comment peut-on être sûr que l'on a correctement assemblé les transformateurs ? »).

Sémantique de la composition des transformateurs Il peut exister des contraintes sur l'ordre de composition des transformateurs dans une même chaîne. De telles contraintes sont dûes à l'existence de dépendances entre les transformateurs, qui peuvent être soit 1) des dépendances entre des implantations de transformateurs, introduites par un développeur, et que nous appelons *dépendances intrinsèques*, soit 2) des dépendances liées au contexte d'utilisation de la transformation de programme, que nous appelons *dépendances contextuelles*. Dans les deux cas, ces dépendances entre transformateurs sont implicites.

Dans le cas de *dépendances intrinsèques*, typiquement, deux transformateurs (composants primitifs) sont développés pour être utilisés conjointement dans une même chaîne, mais sont contenus dans deux opérations (composants composites) séparées. Si l'une des opérations est utilisée sans l'autre, ou si un certain ordre n'est pas respecté, un transformateur peut générer des erreurs. Nous considérons la présence de dépendances intrinsèques comme une faute de développement (ou d'empaquetage), car elles violent notre principe 5 d'Encapsulation et de Composabilité des Transformations. En effet, de telles opérations ne peuvent pas être composées arbitrairement avec d'autres opérations. Un moyen simple de casser une telle dépendance est de « livrer » simultanément les deux transformateurs, en les composants correctement dans un même composant opération, en application du principe 8 de Fermeture Commune.

Les *dépendances contextuelles* des transformateurs sont liées à des objectifs de haut niveau, ou aux programmes transformés. Il n'est pas possible de raisonner sur les objectifs de haut niveau des utilisateurs. Par exemple, considérons une chaîne comprenant deux transformateurs : un transformateur qui insère du code dans chaque méthode pour afficher une trace d'exécution, et un transformateur qui renomme des méthodes. Si l'utilisateur souhaite afficher le nom original de chaque méthode à l'exécution, il doit assembler le transformateur de trace avant le transformateur de renommage. Cependant, dans un contexte différent, un autre utilisateur pourrait choisir l'ordre inverse, pour afficher les noms de méthodes après renommage. En ce qui concerne les dépendances avec les programmes transformés, un transformateur peut être par exemple développé spécifiquement pour une convention de programmation particulière. Si un programme transformé ne respecte pas cette convention de programmation, le transformateur peut générer une erreur.

Que les dépendances soient intrinsèques ou liées au contexte, le problème est que les dépendances entre transformateurs, s'il en existe, sont implicites et ne peuvent pas être déterminées automatiquement par un logiciel. En effet, de telles dépendances ne se concrétisent pas par des liaisons entre composants. Ces dépendances doivent être explicitement exprimées par un utilisateur, en spécifiant la composition des composants. La spécification d'une composition de transformateurs est appelée *stratégie de transformation* [Vis01]. Jabyce, comme la plupart des

autres systèmes de transformation, permet à un utilisateur d'exprimer ses propres stratégies, cf. la section 5.6.

Une violation de dépendance dans l'ordre de composition de transformateurs peut se manifester soit 1) par des erreurs de correction syntaxique ou sémantique du programme transformé, soit 2) par des effets non désirés lors de l'exécution du programme transformé. Ce dernier type de manifestation n'est pas détecté par un vérificateur d'une JVM, et donc ne fait pas partie des que nous souhaitons détecter (notre « cahier des charges »). Il n'y a pas d'erreur spécifique à la sémantique de la composition des transformateurs. Il n'est ainsi pas possible de détecter spécifiquement de telles violations, donc de diagnostiquer spécifiquement de telles fautes.

Modèle de fautes Nous considérons que la cause des erreurs se produisant pendant la transformation sont soit 1) des *fautes de développement* des transformateurs (*« bugs »*), soit 2) des *fautes de composition* des transformateurs, soit 3) des *fautes intermittentes*. Les autres types de fautes définis dans la catégorisation donnée dans [Lap96] ne concernent pas notre contexte. Une faute de développement est par exemple une implantation incorrecte d'un algorithme dans un transformateur, qui provoque le passage d'une valeur `null` en paramètre dans un appel de méthode qui représente un élément d'un programme, alors que cela est une erreur de correction syntaxique du programme. Nous considérons comme fautes intermittentes des fautes ponctuelles et internes, c'est-à-dire liées à l'activation de traitements dans le système. Ces fautes intermittentes peuvent être dans notre cas dues à des phénomènes physiques, ou dues à l'homme (p.ex. erreurs lors de l'implantation de la JVM qui exécute les transformateurs).

Comme décrit ci-dessus, les fautes de composition de transformateurs, c'est-à-dire leur composition dans un ordre ne respectant pas des dépendances entre transformateurs, provoquent des erreurs de même nature que les fautes de développement. Pour simplifier, nous considérons donc dans la suite seulement deux types de fautes : 1) les fautes de développement, et 2) les fautes intermittentes.

Conclusion

En conclusion, notre contribution consiste à proposer un moyen de réaliser toutes les vérifications réalisées par le vérificateur d'une JVM (*verifier*), pendant l'exécution des transformations, pour le test des transformateurs. Un moyen de détection et de diagnostic précis d'erreurs liées à des fautes de développement ou de composition de transformateurs est un *outil de test* très efficace, en permettant de localiser précisément ces fautes avant une utilisation des transformateurs en environnement de production.

4.1.3 Traitement d'erreurs de transformation

Nous nous limitons dans cette section, à l'étude des *moyens pour la sûreté de fonctionnement* qui sont liés à l'exécution de transformations de programmes. Cette problématique est nouvelle, et est donc décrite en détails dans cette section. Le seul moyen de sûreté de fonctionnement que nous considérons est la *tolérance aux fautes*, par un *traitement des erreurs*, et nous ne considérons pas par exemple l'élimination des fautes de développement par le test ou l'analyse

statique des implantations de composants. Comme décrit ci-dessus dans notre modèle de faute, nous considérons essentiellement 1) des fautes de développement, dont la suppression nécessite une modification de l'implantation de composants, et 2) des fautes intermittentes, qui ne nécessitent aucun traitement correctif. Nous ne considérons donc pas le traitement des fautes, qui est le pendant du traitement des erreurs. D'après [Lap96], le traitement d'erreur repose sur trois primitives :

- *détection d'erreur*, qui permet d'identifier un état erroné comme tel ;
- *diagnostic d'erreur*, qui permet d'estimer les dommages créés par l'erreur qui a été détectée et par les erreurs éventuellement propagées avant la détection ;
- *recouvrement d'erreur*, qui permet de substituer un état exempt d'erreur à l'état erroné ; la substitution peut elle-même prendre trois formes :
  - *reprise*, où le système est ramené dans un état survenu avant l'occurrence d'erreur ;
  - *poursuite*, où un nouvel état est trouvé à partir duquel le système peut fonctionner (habituellement en mode dégradé) ;
  - *compensation d'erreur*, où l'état erroné comporte suffisamment de redondance pour permettre la transformation de l'état erroné en un état exempt d'erreur.

Détection et diagnostic des erreurs dans Jabyce

La détection et le diagnostic d'erreur, dans notre contexte, consistent à détecter les interactions représentant des éléments de programmes transformés, qui violent des règles de correction syntaxique ou sémantique de programmes. Le mécanisme général que nous mettons en œuvre est *l'interception des interactions*, entre les composants opérations et (dé)sérialiseurs d'une chaîne. Une telle chaîne est illustrée dans la figure 3.7. En pratique, seule l'interception sur les interfaces serveur des composants est nécessaire dans notre cas, parce que cela suffit à intercepter les interactions qui représentent, sur une liaison, des éléments d'un programme. Les interactions sont interceptées sur toutes les interfaces permettant des interactions représentant des éléments de programmes. Il est nécessaire d'*observer les interactions vers chaque composant*, pour pouvoir localiser précisément quel composant est fautif lors de la détection d'une erreur.

Un programme à transformer peut être incorrect à l'origine, avant même d'être transformé. Cela se traduit par une défaillance dans un composant désérialiseur, par la levée d'une exception, ou par l'initiation sur ses interfaces client d'une séquence d'interactions représentant ce programme incorrect. Dans tous les cas, l'erreur est détectée et la faute est facilement localisable.

Recouvrement d'erreur dans Erlang/OTP

Le principal problème dans le traitement des erreurs de transformation est le choix d'une forme de recouvrement d'erreur, c'est-à-dire d'une *politique de recouvrement d'erreur*. Notre proposition s'inspire fortement de l'approche choisie dans le langage Erlang [R03] et le canevas logiciel OTP (*Open Telecom Platform*) [Tor97, Arm03]. Nous avons repris la politique de recouvrement d'erreur qui est mise enœuvre dans OTP.

Erlang est basé sur un calcul de processus : l'abstraction de base est le *processus*. Les processus Erlang échangent des messages pour offrir des services, et sont éventuellement répartis sur plusieurs nœuds. Une machine virtuelle, similaire à une machine virtuelle Java, supporte

l'exécution des processus. Un mécanisme d'exception similaire à celui de Java permet de notifier l'occurrence d'erreurs (instructions throw et exit), et de mettre en œuvre une « panique organisée » (instruction catch). Il est possible de *lier des processus* entre eux, avec les fonctions link et spawn_link, pour que lorsqu'un processus se termine à cause d'une exception, la machine virtuelle propage cette exception à tous ses processus liés. Il est également possible de configurer la machine virtuelle pour ne pas propager les exceptions mais plutôt émettre un message à tous les processus liés, en positionnant le drapeau trap_exit à la valeur true pour tous les processus « observés ». Un lien de propagation d'exception est bidirectionnel. Il est ainsi possible de former des groupes de processus qui doivent tous se terminer ensemble si un seul des processus se termine à cause d'une exception.

En s'appuyant sur ces mécanismes de base de Erlang, le canevas logiciel OTP propose une démarche et une architecture pour la sûreté de fonctionnement de systèmes Erlang. OTP permet ainsi de mettre en œuvre un recouvrement d'erreur automatique. Dans OTP, une erreur se traduisant par la terminaison anormale d'un processus, et les processus s'exécutant idéalement sans effet de bord, la politique de recouvrement consiste à redémarrer des processus. Plusieurs politiques peuvent être appliquées, par paramétrage [RÓ3] :

- *one for one* : seuls les processus qui se terminent anormalement sont redémarrés ;
- *one for all* : si un processus se termine anormalement, tous les autres processus gérés ensemble sont terminés, puis tous ces processus sont redémarrés ;
- *rest for one* : politique similaire à *one for all*, où seuls les processus qui avaient été démarrés après le processus erronné, sont terminés et redémarrés.

Il est possible de spécifier une fréquence maximale de redémarrage des processus, pour prévenir l'occurrence de boucles infinies de redémarrage. Le seuil est spécifié comme un nombre de redémarrages ($MaxR$) dans une période donnée ($MaxT$ secondes). Passé ce seuil, le processus qui met en œuvre le recouvrement (processus *superviseur*) termine tous les processus qu'il supervise, et se termine lui-même, pour que le processus superviseur qui le surveille prenne en charge le traitement de l'erreur.

Recouvrement d'erreur dans Jabyce

Dans le contexte de Jabyce, nous proposons d'appliquer une politique similaire à celle des superviseurs Erlang/OTP, en redémarrant des composants lors de l'occurrence d'erreurs, et en recommençant la transformation de tout le programme qui était en cours de transformation. Nous mettons donc en œuvre un recouvrement d'erreur uniquement sous la forme d'une *poursuite* des traitements.

En effet, il n'est pas possible de réaliser une *compensation d'erreur* dans notre contexte, parce qu'aucune redondance n'est prévue pour les transformations. Bien que cela puisse être utile en considérant des fautes intermittentes, il faudrait mettre en œuvre des implantations différentes et redondantes de chaque transformateur pour pallier à des fautes de développement. Une telle redondance est trop complexe à mettre en œuvre en pratique. D'autre part, il n'est pas possible de contrôler l'état des composants opérations et (dé)sérialiseurs, et il n'est pas possible de déterminer si l'exécution de transformations a des effets de bord sur l'état d'autres composants auxquels ils sont liés. Il n'est donc pas possible d'établir des *points de reprise*, donc de mettre en œuvre un recouvrement des erreurs par *reprise*. Par exemple, il n'est pas envisageable de redémarrer seulement un composant opération dans une configuration, et de

continuer la transformation à la dernière classe Java transformée dont la transformation a provoqué une erreur, parce que l'état d'une opération Jabyce peut dépendre de l'ensemble des classes Java déjà transformées. Il faut donc *réinitialiser tous les composants*, et recommencer la transformation de toutes les classes du programme transformé, en ignorant les transformations qui auraient déjà pu être effectuées. Ainsi, bien que Fractal offre en standard un ensemble riche d'interfaces de contrôle sur chaque composant, il n'est pas pertinent de les utiliser pour la poursuite des erreurs dans notre contexte, à part l'interface de type LifeCycleController.

Le mécanisme de réinitialisation des composants que nous proposons s'appuie sur l'interface de contrôle standard de Fractal pour la gestion du cycle de vie, définie dans l'interface Java LifeCycleController. Cette interface définit des opérations pour démarrer et arrêter un composant : startFc(), stopFc() et getFcState(). Ainsi, un recouvrement d'erreur, par la réinitialisation des composants d'une configuration de transformateur, se traduit par l'appel de stopFc() sur l'interface LifeCycleController de chaque composant, puis par l'appel de startFc() sur chaque composant. Cette interface doit être correctement implantée par la classe implantant chaque composant primitif, pour réinitialiser l'état du composant lors d'un appel à stopFc().

De manière similaire à la politique de recouvrement de Erlang/OTP, nous proposons également de mettre en œuvre un mécanisme pour empêcher des boucles infinies de recouvrement d'erreur, en vérifiant un seuil du nombre de recouvrements de chaque type d'erreur dans une période donnée. Tant que les recouvrements pour un type d'erreur se produisent en dessous de la fréquence seuil, nous considérons que ces erreurs sont dûes à des fautes intermittentes, qui peuvent être corrigées par un redémarrage des composants. Si le seuil est dépassé, nous considérons alors que l'erreur est dûe à une faute de développement ou de composition des composants, qui ne peut pas par nature être corrigée par un redémarrage des composants. Dans ce cas, tout le système se termine, et la transformation du programme ne peut pas être effectuée. Dans tous les cas, l'intégrité est préservée : une configuration de transformateurs ne peut pas générer des programmes incorrects.

4.1.4 Conclusion

Nous proposons de détecter, diagnostiquer et recouvrer des erreurs, dans le cadre du test de composants transformateurs avec Jabyce. Notre objectif est de préserver l'intégrité (la correction) des programmes transformés, en mettant en œuvre un traitement d'erreur. Nous proposons que ce traitement d'erreur soit complétement séparé du code des transformateurs. Notre modèle d'erreur consiste en des erreurs de correction des programmes, au niveau de la syntaxe des programmes, et au niveau de leur sémantique. La section 4.2 présente notre proposition d'architecture pour la mise en œuvre de cette approche générale. La section 4.3 présente des formalismes de spécification de contrats pour la détection d'erreur de correction de programmes transformés.

4.2 Architecture pour le traitement d'erreur

Cette section décrit notre proposition d'architecture modulaire pour le traitement des erreurs de correction des programmes transformés dans Jabyce, selon le modèle de faute et le modèle d'erreur définis dans la section précédente. Notre proposition s'inspire fortement de travaux

existants, présentés dans cette section. La section 4.2.1 présente les principes généraux de conception de notre architecture de traitement d'erreur pour Jabyce, inspirés par ceux qui sont appliqués dans Erlang/OTP (*Open Telecom Platform*). La section 4.2.2 présente et critique les architectures de traitement d'erreur de ConFract et TLO, que nous avons réutilisées et étendues dans l'architecture que nous proposons et qui est présentée dans la section 4.2.3.

4.2.1 Inversion du traitement d'erreur

En application du principe 2 d'Inversion de Contrôle, nous proposons de séparer complétement le traitement des erreurs, du code fonctionnel des composants formant une chaîne de transformateurs. La mise en œuvre du traitement d'erreur doit être transparente pour les composants. Ainsi, nous spécifions dans Jabyce que les implantations de composants ne doivent pas traiter les erreurs, c'est-à-dire ne pas mettre en œuvre de programmation défensive, ne pas traiter les exceptions Java, etc. En pratique, nous avons vérifié que l'application de ce principe simplifie beaucoup le développement de composants, notamment des composants transformateurs.

Notre proposition s'inspire fortement de l'approche choisie dans le langage Erlang et le canevas logiciel OTP, présentés dans la section 4.1.3. Le principe de conception qui est appliqué de manière généralisée dans Erlang/OTP, que nous avons baptisé le principe 10 d'Inversion du Traitement d'Erreur, est exprimé dans la littérature sur Erlang/OTP sous la forme de quatre « slogans » [Arm03].

- Laissez une autre partie du système réaliser le traitement d'erreur.
- Si vous ne pouvez pas faire ce que vous voulez faire, mourrez.
- Laissez planter. (*let it crash*)
- Ne programmez pas de manière défensive.

Principe 10 : Inversion du Traitement d'Erreur

L'idée est de séparer complétement le code fonctionnel du code de traitement d'erreur. Ainsi, par exemple, l'utilisation de l'instruction `catch` est proscrit. La généralisation de ce principe est une des propriétés originales de Erlang/OTP. Nous avons repris et appliqué ce principe, en l'identifiant comme un cas particulier du principe 2 d'Inversion de Contrôle. Notamment, nous spécifions que l'interception des interactions pour la détection des erreurs, et le recouvrement des erreurs, doit être transparent pour les composants Jabyce.

4.2.2 ConFract et TLO : observateurs pour Fractal

Nous avons repris le concept d'Observateur, décrit dans [DJC94], et qui est l'architecture classique pour la validation de systèmes pendant leur exécution (*on-line validation*), c'est-à-dire pour la détection et le diagnostic d'erreur. Dans cette architecture, un composant appelé *observateur* observe le comportement du système, pour vérifier qu'il correspond à une spécification formelle. Le système observé, qui est dans notre contexte une configuration de composants Fractal, est appelé *travailleur*. L'observation doit être transparente, et ne pas nécessiter d'action du travailleur. Cela correspond à l'application, pour la détection et le diagnostic des erreurs, du principe 10 d'Inversion du Traitement d'Erreur. Dans notre contexte, les composants

observés sont des composants opérations (transformateurs) Jabyce, comme illustré dans la figure 4.1. Plus précisément, dans notre contexte, l'observateur doit intercepter les interactions entre des composants Jabyce, qui représentent des parties de programmes transformés, pour vérifier qu'elles représentent des programmes corrects.

Comme décrit dans la section 4.1.3, chaque composant doit être observé individuellement, pour pouvoir localiser précisément le composant fautif, voire le fragment de code fautif dans ce composant, lors de la détection d'une erreur. Cependant, nous considérons qu'un seul observateur peut observer plusieurs composants individuellement.

Nous présentons dans cette section les deux implantations d'observateurs qui existent pour les composants Fractal implantés en Java : ConFract et TLO.

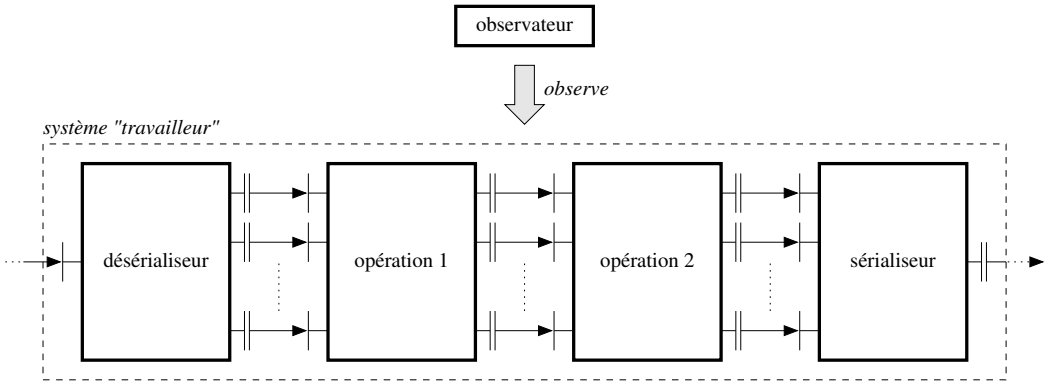


FIG. 4.1 : Architecture globale d'un observateur pour Fractal

ConFract est un modèle et un système de contrat pour les composants Fractal qui a été défini par l'équipe OCL du laboratoire I3S, en collaboration avec le laboratoire DTL/ASR de France Telecom R&D. L'architecutre de ConFract est décrite dans [CO03]. L'apport principal de ConFract est de prendre en compte les caractéristiques du modèle Fractal, notamment la séparation des interfaces des composants et la composition hiérarchique des composants. Dans la terminologie utilisée dans ConFract, l'observateur est un *gestionnaire de contrat*, qui vérifie que les interactions observées entre les composants respectent un *contrat* établi entre les composants participant à ce contrat.

Les différents éléments architecturaux définis dans ConFract sont des *unités de traitement* qui sont placés dans la membrane des composants Fractal. L'interception des interactions et la détection/diagnostic des erreurs sont séparés dans des éléments architecturaux dictincts. La figure 4.2 illustre un composant opération Jabyce intégré avec ConFract et contenant un composant transformateur. Deux éléments sont ajoutés dans la membrane de chaque composant pouvant participer à un contrat : un *gestionnaire de service* (*service controller - SeC*), et un *gestionnaire de contrats* (*contract controller - CtC*).

Un SeC est un intercepteur sophistiqué, qui peut être paramétré dynamiquement pour filtrer les interactions à intercepter. L'interception est transparente pour les composants observés. Typiquement, dans ConFract, un SeC interagit avec un ou plusieurs CtC, dans la même membrane ou dans d'autres membranes, pour leur notifier des occurrences d'interactions concernées par des contrats auxquels participe le composant observé par le SeC. Lors d'occurrence d'interactions, un CtC *évalue* les dispositions des différents contrats qu'il gère et notifie toute

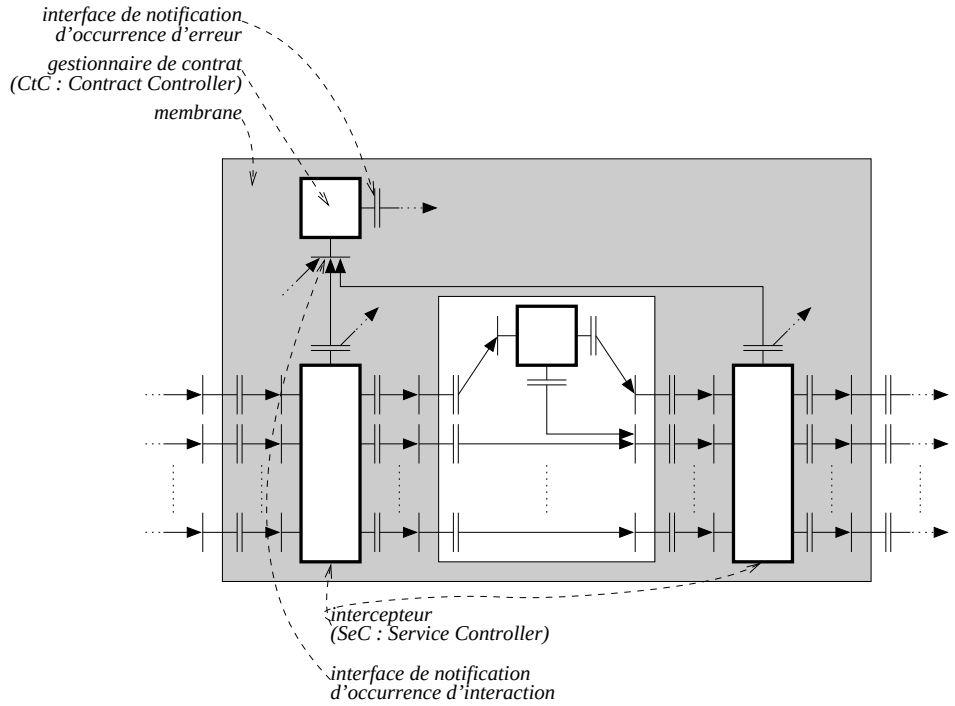


FIG. 4.2 : Eléments architecturaux de ConFract

violation, typiquement en levant une exception. Un CtC peut gérer plusieurs contrats. Les CtC et SeC dans la membrane d'un composant sont inactifs si aucun contrat ne concerne ce composant. L'implantation de SeC dans ConFract n'est pas spécifique à la vérification de contrats, et peut être réutilisée dans d'autres contextes.

Dans l'architecture de ConFract, de manière à rendre le gestionnaire de contrat indépendant des formalismes de spécification de contrats, l'évaluation des dispositions de contrats est déléguée à d'autres objets dans la membrane des composants : les *évaluateurs de dispositions de contrats*. Un CtC est lié à un évaluateur pour chaque formalisme. Cette architecture est illustrée dans la figure 4.3. Chaque évaluateur est spécifique à un formalisme de spécification. Pour l'instant, seul un évaluateur pour le formalisme CCL-J a été développé. CCL-J est présenté dans la section 4.3.3.

D'un point de vue synthétique, un SeC d'un composant est lié à chaque CtC qui gère un contrat auquel participe le composant, pour notifier à ces CtC les occurrences d'interactions vérifiées par ces contrats. Un CtC est lié au SeC de chaque participant à un des contrats gérés par ce CtC. Un CtC lié à un ou plusieurs évaluateurs de dispositions de contrats, un pour chaque formalisme utilisé pour la spécification des dispositions des contrats gérés par ce CtC.

TLO (*Temporal Logic Observer*) est un outil pour l'observation et la vérification de contrats sur des composants Fractal pendant leur exécution. TLO est développé dans le laboratoire DTL/ASR de France Telecom R&D. TLO s'appuie sur une technique de *model-checking*. L'architecture de TLO est conceptuellement identique à celle de ConFract, décrite ci-dessus, et comprend les mêmes éléments. La différence essentielle entre TLO et ConFract tient dans les formalismes de spécification supportés par ces outils, comme présenté dans la section 4.3.

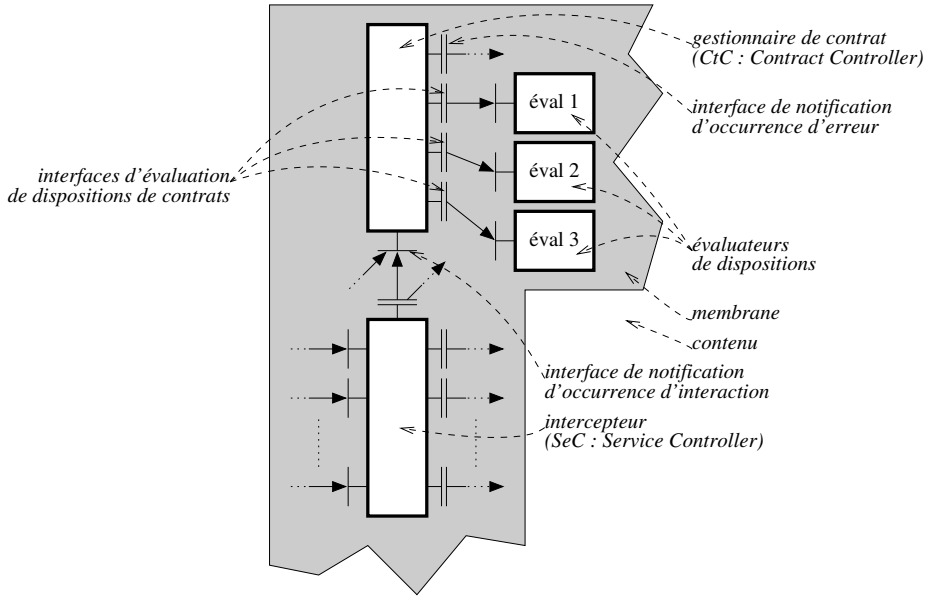


FIG. 4.3 : Evaluateurs de dispositions de contrats ConFract dans la membrane des composants

L'appellation TLO est dûe au fait que les spécifications formelles des comportements vérifiés par TLO sont exprimées dans le formalisme TLA (*Temporal Logic of Action*). Les différents formalismes que nous avons abordés sont décrits dans la section 4.3.

Cependant, l'implantation de TLO est moins évoluée. Notamment, lors de la détection d'une erreur, l'observateur de TLO peut seulement afficher un rapport d'erreur sur la console et lever une exception. Il ne peut pas être configuré pour notifier l'occurrence d'erreurs à d'autres parties du système, comme le gestionnaire de contrat de ConFract. De plus, pour l'instant l'architecture de TLO ne supporte que des contrats portant sur un seul composant, et ne permet de spécifier des contrats portant sur plusieurs composants comme dans ConFract. Cette contrainte n'est liée qu'à l'architecture de TLO : un intercepteur TLO ne peut communiquer qu'avec un seul observateur TLO. Nous avons donc choisi ConFract, et non TLO, comme base architecturale pour le traitement d'erreur dans Jabyce. Nous proposons de modifier l'implantation de TLO, pour devélopper dans l'architecture ConFract un évaluateur de dispositions de contrats spécifiées en TLA.

4.2.3 Vers un domaine de traitement d'erreur

Cette section présente notre proposition d'architecture pour le traitement d'erreur dans Jabyce, qui se base sur l'architecture de ConFract présentée ci-dessus. Les principales évolutions que nous proposons, par rapport à cette architecture, sont motivées par notre besoin de mettre en œuvre un traitement d'erreur complet, y compris un *recouvrement des erreurs* (cf. la section 4.1.3). En effet, les architectures d'observateurs, et notamment ConFract, n'abordent essentiellement que la détection et le diagnostic des erreurs. Par défaut, un CtC (gestionnaire de contrat ConFract) lève une exception détaillée lors de la détection d'une violation de contrat.

Il est cependant possible d'associer à un CtC un élément, dans la même membrane, qui implante un processus de contrôle de contrat arbitraire, c'est-à-dire qui réalise un recouvrement d'erreur arbitraire, comme illustré dans la figure 4.4. Pourtant, cette architecture pose deux problèmes dans notre contexte : 1) à cause de la différence entre la portée d'un contrat et la portée du recouvrement d'erreur associé, et 2) à cause du modèle d'implantation du gestionnaire de contrat et du récupérateur dans ConFract. L'architecture que nous proposons dans la suite, est une évolution de celle de ConFract pour résoudre ces problèmes.

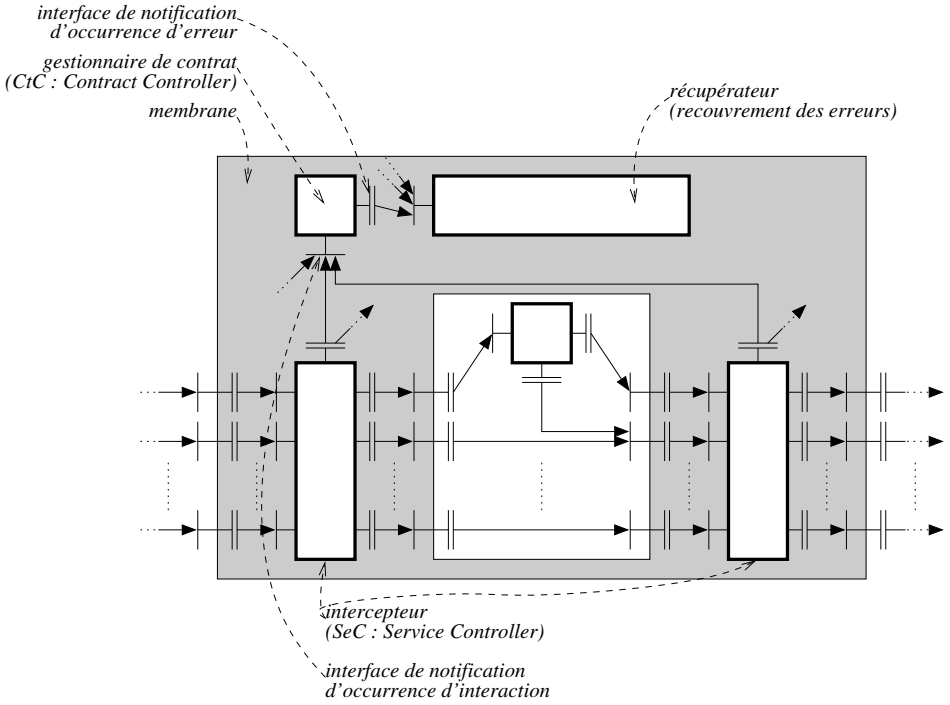


FIG. 4.4 : Intégration du recouvrement d'erreur dans ConFract

Portée du contrat vs. portée du recouvrement

Dans ConFract, lorsqu'un contrat concerne plusieurs composants, le gestionnaire du contrat est celui qui est localisé dans le composite qui englobe tous ces composants participants, qui forment la *portée du contrat*. Cependant, ce choix de la localisation de la gestion d'un contrat en fonction de sa portée est erronnée, si l'on considère le recouvrement d'erreur. En effet, la portée d'un contrat, c'est-à-dire l'ensemble des composants observés, ne correspond pas nécessairement à l'ensemble des composants concernés par le recouvrement d'erreur. La portée du recouvrement d'erreur peut être plus large, et concerner plus de composants. Par exemple, dans Jabyce, toutes les opérations de transformation d'une chaîne sont touchées par un recouvrement d'erreur, si une violation du contrat d'une seule opération est détectée, selon notre politique de recouvrement d'erreur présentée dans la section 4.1.3. De plus, d'autres composants, par exemple un composant de stockage des classes transformés, sont concernés par le recouvrement d'erreur (vidage du composant de stockage, etc.). Dans ce contexte, nous considérons qu'il n'est pas pertinent de placer l'ensemble des composants participants à un

contrat dans un seul composite qui le concrétise et limite sa portée. Nous proposons plutôt de définir explicitement la portée comme l'ensemble des liaisons avec le gestionnaire de contrat et le recouvreur, ceux-ci étant conçus comme des composants Fractal. Il sera donc nécessaire de faire évoluer ConFract pour sortir les gestionnaires de contrat et les recouvreurs des membranes des composants, et pour les concevoir comme des composants Fractal.

Implantation du gestionnaire de contrat et du recouvreur

Les différents éléments architecturaux introduits dans ConFract (SeC et CtC), notamment les éléments pour le recouvrement d'erreur, sont localisés dans la membrane des composants. Leurs implantations sont spécifiques aux implantations de Fractal. En l'occurrence, dans Fractal/Java, ces éléments sont des objets Java. Ces objets sont appelés *contrôleurs*, et sont développés selon les spécifications de Julia, l'implantation de référence de Fractal en Java. Ce ne sont pas des composants Fractal. En effet, la conception de contrôleurs comme des composants Fractal eux-mêmes n'est pas encore possible avec les implantations actuelles de Fractal. Il est pour l'instant nécessaire d'utiliser des modèles et des mécanismes spécifiques aux implantations de Fractal.

Dans un contrat concernant plusieurs composants, les objets SeC de tous les composants participants interagissent avec l'objet CtC qui gère le contrat, qui est localisé dans la membrane du composant composite commun à tous les participants. Or, ces interactions ne sont pas réalisées à travers des liaisons entre composants, mais par des appels de méthodes directs entre ces objets, qui forment donc des *liaisons cachées* qu'il est difficile de contrôler pendant l'exécution. De plus, l'inconvénient principal est que les éléments architecturaux de ConFract sont des objets Java développés spécifiquement pour Julia, est la difficulté de développement et de configuration de ces éléments architecturaux. Nous proposons donc de faire évoluer l'architecture de ConFract, pour concevoir ces éléments comme des composants Fractal.

Domaine de traitement d'erreur

L'architecture que nous proposons repose sur le concept de domaine dans RM-ODP (cf. l'annexe C). Nous introduisons ainsi le concept de *domaine de traitement d'erreur*. La fonction du *contrôleur* d'un tel domaine, que nous concevons comme un composant Fractal, est de mettre en œuvre les trois primitives du traitement d'erreur : détection, diagnostic et recouvrement d'erreur. Ainsi, non seulement un contrôleur est un observateur similaire à un gestionnaire de contrat ConFract, mais aussi il met en œuvre le recouvrement d'erreur.

Dans un domaine, la relation entre un contrôleur et les composants contrôlés est concrétisée par des liaisons entre ces composants. Les composants liés forment *la portée d'un domaine de traitement d'erreur*, qui est l'union de la portée du contrat et de la portée du recouvrement d'erreur. Ces liaisons permettent au contrôleur 1) d'observer les interactions sur les interfaces des composants contrôlés (liaisons de notification), et 2) d'interagir avec ces composants pour le recouvrement d'erreur (liaisons de réflexion). La figure 4.1 illustre l'architecture que nous proposons.

De même que dans ConFract, et en analogie avec l'approche des langages spécifiques aux domaines (DSL), nous séparons les mécanismes de traitement d'erreur (détection et diagnostic, et mécanismes pour le recouvrement), des politiques/stratégies de recouvrement à mettre en

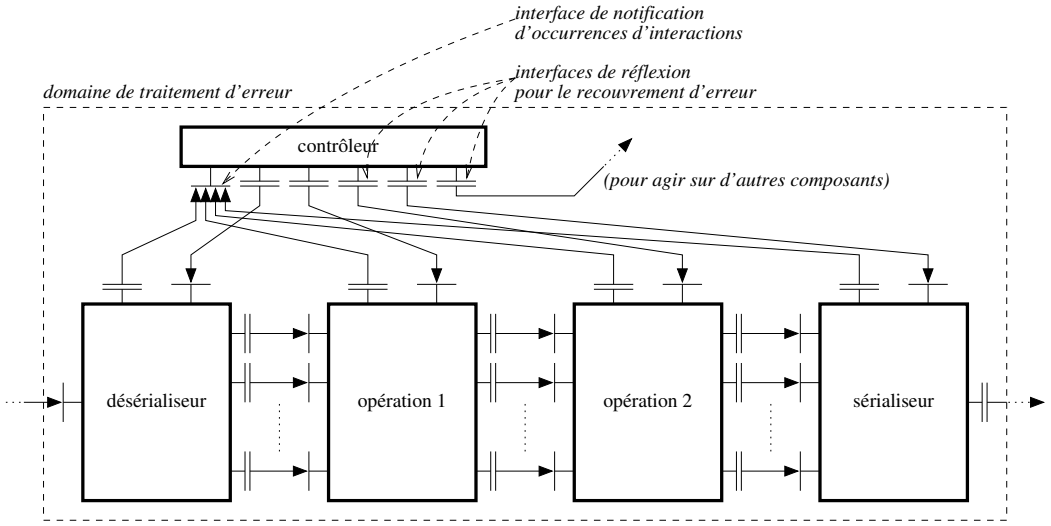


FIG. 4.5 : Architecture d'un domaine de traitement d'erreur

œuvre en fonction des erreurs détectées. Cela se traduit par la conception du contrôleur de traitement d'erreur comme un composant composite, contenant un sous-composant dédié à la détection et au diagnostic (*gestionnaire de contrat*), et un autre sous-composant mettant en œuvre une politique de recouvrement, comme illustré dans la figure 4.6. Les évaluateurs des dispositions de contrats sont également séparés dans des composants liés au gestionnaire de contrat. La séparation entre politiques et mécanismes de recouvrement d'erreur est naturelle avec Fractal, parce que les principaux mécanismes de recouvrement que nous considérons sont les contrôles fournis en standard par Fractal dans les membranes de chaque composant (contrôle du contenu, contrôle du cycle de vie, etc.), via les interfaces de contrôle (d'intercession).

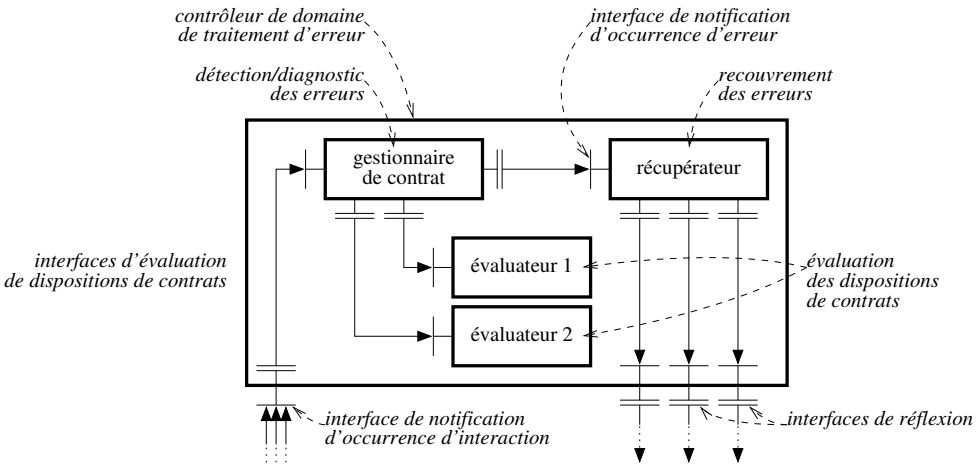


FIG. 4.6 : Architecture d'un contrôleur de domaine de traitement d'erreur

Cette architecture est très proche de celle de ConFract, et nous envisageons de réutiliser directement l'intercepteur de ConFract (SeC) dans les membranes des composants, comme illustré dans la figure 4.7, et de redévelopper le gestionnaire de ConFract comme un composant

Fractal.

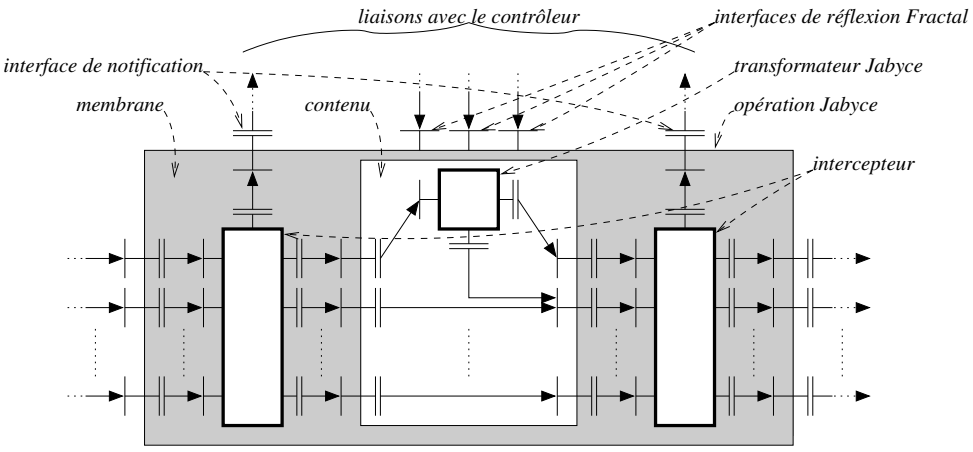


FIG. 4.7 : Intercepteurs dans la membrane d'une opération Jabyce

Comparaison avec Erlang/OTP

Le canevas logiciel Erlang/OTP, présenté dans la section 4.1.3, propose en standard une architecture de traitement d'erreur. Cette architecture se base également sur une l'architecture d'observateur, comme définie ci-dessus, mais où un observateur réalise également un recouvrement d'erreur lorsqu'une erreur est détectée. Ainsi, OTP définit deux sortes de processus : les processus *superviseurs* (*supervisor*), qui correspondent à des observateurs, et les processus *travailleurs* (*workers*). Les processus superviseurs démarrent les processus travailleurs, et traitent les terminaisons erronnées de ces processus. Un superviseur a pour fonction de détecter et de diagnostiquer les erreurs, et d'effectuer un recouvrement d'erreur soit 1) en redémarrant les processus, soit 2) en terminant tous les processus qu'il supervise et en se terminant lui-même pour propager l'erreur à son propre superviseur. Cette politique de recouvrement d'erreur est décrite dans la section 4.1.3. L'originalité de Erlang/OTP est de pouvoir construire des arbres de superviseurs, pour compartimenter le traitement des erreurs. Cette caractéristique n'a cependant pas d'intérêt dans notre contexte.

Dans notre approche, un contrôleur de domaine de traitement d'erreur est très similaire à un processus superviseur OTP, et applique une politique de recouvrement d'erreur similaire, comme expliqué dans la section 4.1.3. La principale différence se situe dans le mécanisme de notification des erreurs entre composant/processus travailleur et contrôleur/processus superviseur. Dans le cas d'Erlang, ce mécanisme est mis en œuvre par la machine virtuelle, comme décrit dans la section 10, alors que dans notre proposition nous devons enrichir la membrane des composants Fractal pour y introduire des intercepteurs. Une autre différence est que nous ne traitons pas des erreurs se manifestant par des exceptions, mais par des interactions qui sont identifiées comme violant des contrats. La détection de telles erreurs en Erlang imposerait d'intercepter les messages reçus et émis par les processus, ce qui ne peut pas être réalisé de manière transparente en Erlang. De plus, dans Erlang/OTP, la détection et le recouvrement d'erreur sont mélangés dans les processus superviseurs, ce qui rend leur évolution plus difficile. Notre approche est plus modulaire.

Resynchronisation de l'observateur

Lors du recouvrement d'une erreur, l'état du composant de détection d'erreur doit être synchronisé avec le nouvel état du système observé. Pour cela, [SS96] propose une architecture d'un observateur qui comprend deux composants : 1) un composant de détection d'erreur (*FDM - Failure Detection Mechanism*), et 2) un composant de resynchronisation d'état (*SRM - State Resynchronisation Mechanism*). A la détection d'une erreur, ces deux composants ne réalisent pas de recouvrement d'erreur, et le FDM émet seulement une notification. Le recouvrement est réalisé par un autre composant dans le système. Il est cependant nécessaire de resynchroniser le FDM avec le nouvel état des composants observés, pour qu'il puisse détecter des erreurs pendant la suite de l'exécution du système. Le nouvel état des composants observés est inconnu par le FDM. Le SRM introspecte cet état, et affecte un nouvel état au FDM en conséquence. Dans notre architecture de domaine de traitement d'erreur, le composant de détection d'erreur (gestionnaire de contrat) joue le rôle de FDM, et le composant de recouvrement joue le rôle de SRM. Nous avons introduit un composant de recouvrement d'erreur directement dans l'architecture, dont le rôle est justement de modifier l'état de tous les composants du domaine. Plus précisément, notre politique étant de réinitialiser tous les composants observés lors d'un recouvrement, le composant de détection d'erreur est aussi réinitialisé. Pour cela, le composant de recouvrement doit être lié à l'interface de gestion de cycle de vie du gestionnaire de contrat.

4.2.4 Conclusion

En conclusion, nous proposons une architecture de domaine de traitement d'erreur, qui est une évolution de l'architecture de ConFract, un système de gestion de contrats pour Fractal. Notre architecture se base sur les caractéristiques de Fractal : la gestion de contrat et le recouvrement sont implantés dans des composants Fractal, les portées des contrats et du recouvrement d'erreur sont définies par des liaisons entre composants Fractal, l'interception est implantée en enrichissant la membrane des composants, et le recouvrement d'erreur s'appuie sur les interfaces de contrôle standard de Fractal. Cette architecture est encore à l'état de proposition, et n'est pas encore implantée. Cependant, ConFract est déjà implanté et fonctionnel, et constitue une base essentielle de notre proposition.

4.3 Spécification de contrats

Dans l'architecture de domaine de traitement d'erreur que nous proposons (cf. la figure 4.6), un composant *gestionnaire de contrat* met en œuvre la détection des erreurs. Sa fonction est de vérifier que les interactions observées entre les composants respectent un *contrat*. Dans le contexte de Jabyce, les contrats concernent la *correction des éléments des programmes* représentées par ces interactions. ConFract définit un modèle de contrat et des concepts généraux, sur lesquels nous nous basons pour spécifier des contrats de correction de programmes. ConFract est donc à la fois un modèle général de contrat, et une architecture de gestion de contrat et une implantation de cette architecture, décrite dans la section 4.2 et que nous avons étendue. ConFract n'impose pas de formalisme de spécification de contrat, et a pour objectif d'accepter une grande variété de techniques de spécifications formelles ou semi-formelles.

Cette section présente des expérimentations que nous avons menées, pour la *spécification de contrats de correction de programmes* transformés. Le modèle de contrat ConFract est présenté dans la section 4.3.1. Dans le cadre donné par ce modèle de contrat, les sections suivantes présentent les différents formalismes que nous avons abordés : iContract, CCL-J et TLA. iContract est un formalisme de spécification de contrats sur des objets Java, et une implantation de gestionnaire de contrat. CCL-J est le seul formalisme actuellement défini pour ConFract, pour la spécification du comportement de composants Fractal/Java. TLA (*Temporal Logic of Action*) est le formalisme utilisé dans le gestionnaire de contrat TLO, présenté dans la section 4.2.2.

Le critère de comparaison de ces différents formalismes, est leur capacité à spécifier 1) des contrats de correction syntaxique des programmes, et 2) des contrats de correction sémantique, respectivement. Notamment, la spécification de contrats de correction sémantique impose de pouvoir raisonner sur un historique des interactions représentant des éléments de programmes, et de considérer simultanément plusieurs interfaces d'un même composant, ce que ne permettent pas tous les formalismes.

4.3.1 ConFract : un modèle de contrat pour Fractal

ConFract ne fait pas référence aux concepts du standard RM-ODP [ISO95a] (cf. l'annexe C), mais ils introduisent des concepts très similaires. Le modèle de contrat de ConFract est défini dans [CR03]. Le concept de contrat dans RM-ODP et dans ConFract doit être compris comme une métaphore du concept de contrat entre individus humains. Les individus humains qui implantent et assemblent les composants délèguent leurs responsabilités aux composants. Les composants sont des *mandataires* pour les individus humains. Ainsi, dans Jabyce, les développeurs de composants transformateurs et les assembleurs de composants opérations sont garants de la correction des représentations de programmes produites par ces composants, et sont bénéficiaires de la correction des représentations de programmes reçues par ces composants. La garantie impose que les développeurs ne génèrent pas de fautes de développement ou d'assemblage des composants (cf. notre modèle de faute dans la section 4.1.3) qui peuvent générer des erreur de correction des programmes. Le bénéfice est une simplification du développement des composants : il n'est plus nécessaire de se soucier de la correction des représentations de programmes reçus.

Le modèle de contrat ConFract

D'après RM-ODP, un *contrat* est un accord réglant une partie du comportement d'un ensemble d'objets, le *comportement* étant un ensemble d'actions, par exemple des interactions entre objets. Dans le cadre de Fractal / ConFract, ces interactions sont les appels émis et reçus à travers les interfaces des composants. Un contrat est un ensemble d'obligations, de permissions et d'interdictions sur un comportement. Une définition similaire est donnée dans ConFract, où un contrat est un ensemble de *dispositions* qui doivent être vérifiées par un comportement commun à un ensemble de composants. Dans RM-ODP, un contrat est mis en place explicitement par un *comportement d'établissement*, qui est dans ConFract une *négociation* entre les composants qui participeront au contrat. Une *spécification* est le support de la négociation d'un contrat entre des composants participants : le contrat est un consensus entre les spécifications fournies par les garants, et celles fournies par les bénéficiaires.

Dans ConFract, une *spécification* est un ensemble de *clauses*, qui ont chacune des *participants* identifiables du point de vue de leurs responsabilités : un garant, et un ou plusieurs bénéficiaires. Après négociation, les clauses d'une spécification sont traduites en dispositions dans un contrat, lorsque l'identité des composants participants est connue à l'exécution du système.

ConFract va plus loin que RM-ODP, en définissant non seulement les concepts décrits ci-dessus, mais aussi une architecture et des mécanismes pour mettre en œuvre les processus de gestion de contrats. Cette architecture est présentée dans la section 4.2.2.

Portée des contrats et du traitement d'erreur dans Jabyce

Dans le contexte de Jabyce, les clauses des spécifications de contrats que nous souhaitons vérifier sont des règles de correction syntaxique ou sémantique des programmes qui sont représentés par des séquences d'interactions. Ces clauses sont traduites à partir de la spécification de la JVM [LY99]. Ces clauses sont donc stables, et ne font pas l'objet d'une négociation dans Jabyce. Dans les contrats que nous considérons, nous définissons que les composants *garants* sont les composants désérialiseurs et opérations, qui garantissent qu'ils ne produisent que des représentations de programmes corrects. Les *bénéficiaires* sont les composants opérations et sérialiseurs, qui sont garantis qu'ils reçoivent seulement des représentations de programmes corrects. Les rôles des participants aux contrats sont illustrés dans la figure 4.8. Un composant opération Jabyce est donc à la fois bénéficiaire de la correction des représentations de programmes qu'il reçoit, et garant de la correction des représentations de programmes qu'il produit. Un contrat est vérifié sur l'ensemble des liaisons entre deux composants, parce que c'est sur un tel ensemble de liaisons qu'est représenté un programme à vérifier. De plus, vérifier les programmes représentés sur les liaisons entre *chaque paire* de composants, permet de localiser précisément les composants fautifs. Plusieurs contrats sont gérés dans un même domaine de traitement d'erreur, comme illustré dans la figure 4.5 pour la même configuration de composants, de manière à pouvoir redémarrer tous les composants si un seul contrat est violé, comme expliqué dans la section 4.1.3. Pour synthétiser, si une chaîne comprend un désérialiseur, un sérialiseur et N opérations, ces composants font partie d'un unique domaine de traitement d'erreur, et ce domaine gère $N+1$ contrats, c'est-à-dire un contrat pour chaque paire de composants liés.

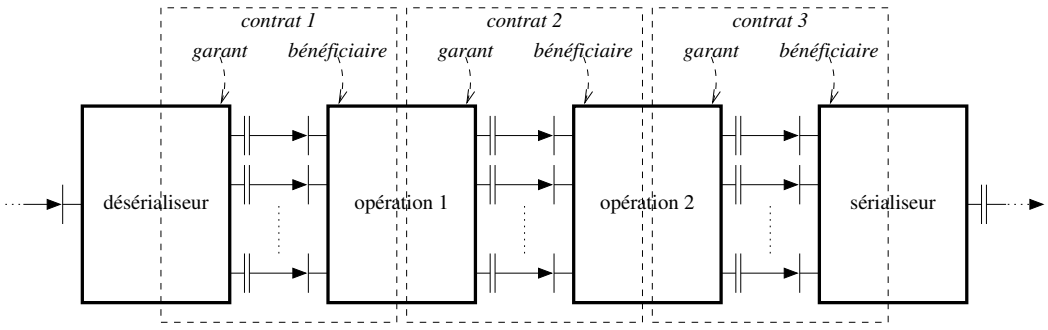


FIG. 4.8 : Composants participants aux contrats dans Jabyce

Vérification modulaire de correction de programmes dans Jabyce

Nous suivons une *approche modulaire* pour la spécification et la vérification des contrats dans Jabyce. C'est-à-dire que nous proposons de séparer en de nombreuses dispositions les règles de correction à vérifier, dans de nombreux contrats distincts, pour offrir une grande liberté dans le choix, par l'utilisateur, des règles de correction à vérifier. En conséquence, en pratique, un même composant opération est garant de plusieurs contrats, et bénéficiaire de plusieurs contrats. Le découpage en de nombreuses dispositions de contrats est lié à un découpage des règles de correction des programmes en de nombreuses spécifications distinctes. Ce découpage permet de mieux appréhender ces spécifications.

Les différents types de contrats dans ConFract et Jabyce

ConFract définit deux types de contrats, en fonction de la portée des contrats :

- *contrat d'interface* (ou *contrat de liaison*) : un contrat sur une interface d'un composant. La spécification d'un contrat d'interface est liée à la signature d'une interface, c'est-à-dire à une interface Java.
- *contrat de composition* : un contrat sur plusieurs interfaces d'un composant et/ou de ses sous-composants.

Nous considérons que la distinction entre ces deux types de contrats est similaire à la distinction que nous faisons entre les deux types d'erreurs (syntaxe et sémantique des programmes) que nous avons définis dans notre modèle d'erreur, cf. la section 4.1.2. Notre critère pour distinguer les deux types d'erreur est la nécessité, ou non, de prendre en compte plusieurs interactions pour détecter une erreur. La détection d'une erreur de correction syntaxique s'effectue en observant chaque interaction représentant un élément d'un programme, indépendamment des autres interactions : cela correspond à la vérification d'un contrat d'interface. En revanche, la détection d'une erreur de correction sémantique s'effectue en observant plusieurs interactions représentant plusieurs éléments d'un programme. Si les éléments observés sont de même type, ils sont représentés par des interactions sur la même interface : cela correspond à la vérification d'un contrat d'interface. Si les éléments observés sont de types différents, ils sont représentés par des interactions sur des interfaces différentes d'un même composant : cela correspond à la vérification d'un contrat de composition. Dans notre contexte, un critère de comparaison des formalismes de spécification est donc *la capacité à spécifier des contrats de composition.* Des exemples de spécifications de contrats d'interface et de composition sont donnés dans les sections suivantes.

L'état des contrats : un historique des interactions observées

Pour la détection d'une erreur de correction sémantique de programme, il est nécessaire de considérer plusieurs interactions qui se produisent dans une même séquence d'interactions. Il est donc nécessaire de pouvoir maintenir un *historique des interactions* qui se produisent, donc de maintenir un *état dans les contrats.* Dans ConFract, les dispositions des contrats peuvent avoir un état. Cependant, il est nécessaire pour un formalisme de spécification de raisonner sur un tel état : cela constitue un critère supplémentaire de comparaison des formalismes.

Par exemple, un contrat de correction sémantique peut garantir que toute représentation d'une instruction d'accès à un attribut (getfield, putfield, getstatic ou putstatic) référence un attribut qui est effectivement représenté. Plus précisément, si une interaction observée i1 représente une instruction getfield qui référence un attribut someField d'une autre classe SomeClass, une interaction i2 doit se produire, avant ou après i1, pour représenter l'attribut someField dans une séquence d'interactions représentant la classe SomeClass. Les interactions observées dans ce cas sont les appels à createFieldAccessInstruction sur l'interface de type FieldAccessInstructionFactory, et les appels à createField sur l'interface de type FieldFactory. Il est nécessaire de maintenir dans ce contrat un état qui est constitué 1) de la liste des noms des attributs représentés par les interactions observées, et 2) de la liste des noms des attributs référencés par les instructions représentés par les interactions observées. En ce qui concerne la spécification de ce contrat, il est nécessaire de pouvoir raisonner sur ces deux listes dans les clauses de la spécification. Un exemple de spécification de ce contrat est donné dans le formalisme CCL-J dans la figure 4.13.

Cet exemple illustre également le fait que l'état qui doit être maintenu dans les contrats est une *synthèse de l'historique des interactions observées* : toutes les interactions ne sont pas nécessairement conservées dans cet historique.

Synthèse sur nos critères de comparaison des formalismes de spécification de contrat

En conclusion, nous considérons les quatre critères suivants pour l'évaluation des formalismes de spécification de contrats dans le contexte de Jabyce :

- l'adéquation avec le modèle de composant Fractal ;
- l'adéquation avec l'architecture et le modèle de contrat de ConFract ;
- la possibilité de définir des contrats de composition, portant sur plusieurs interfaces d'un ou plusieurs composants ;
- la possibilité de raisonner sur un état propre à chaque contrat, pour maintenir un historique synthétique des interactions observées.

Nous utilisons ces critères pour évaluer trois formalismes dans les sections suivantes : iContract, CCL-J et TLA.

4.3.2 iContract : contrats sur objets Java

La première approche que nous avons étudiée est la conception d'objets par contrats (*Design by Contract*) [Mey97], en utilisant l'outil iContract [Kra98, Kra]. Dans les spécifications iContract, les clauses sont des pré- et post-conditions et des invariants spécifiés sous la forme d'assertions exécutables. Ces assertions sont écrites sous la forme de commentaires dans le code source d'interfaces ou de classes Java. La syntaxe est similaire à celle du langage de spécification OCL (*Object Constraint Language*) défini pour les modèles UML (*Unified Modelling Language*) [OMG03]. OCL est lui-même inspiré du langage de programmation Eiffel [Mey97]. Ces assertions sont automatiquement compilées par iContract dans le code source des classes Java. Les assertions sont exécutées lors des appels des méthodes de ces classes, et une exception est levée si une assertion n'est pas vérifiée (c'est-à-dire n'est pas évaluée à true). Un exemple de spécification iContract dans Jabyce est décrit dans la section 3.3.1.

Architecture de iContract

Comparée à l'architecture de ConFract présentée dans la section 4.2.2, l'architecture de iContract est rigide : l'interception des interactions et la gestion de contrats sont mélangés et non modulaires. L'un des avantages de mélanger ces aspects avec le code source des applications, est que les performances (en temps d'exécution) peuvent être améliorées automatiquement à la compilation, par une optimisation du code effectuée par le compilateur. Cependant, l'inconvénient est que la détection d'une violation de contrat ne peut déclencher que la levée d'une exception. Il n'est pas prévu de pouvoir réaliser un recouvrement d'erreur complexe lors d'une violation de contrat. Ainsi, avec iContract, il est impossible de mettre en œuvre notre architecture modulaire de domaine de traitement d'erreur décrite dans la section 4.2, où le recouvrement d'erreur est délégué à un autre composant.

D'autre part, comme iContract ne peut intercepter des appels de méthodes que dans le code des classes Java, iContract ne peut être utilisé pour la vérification de contrats que dans les composants Fractal primitifs. En effet, les composants Fractal composites n'ont pas d'implantation, contrairement aux composants primitifs. C'est une limitation importante par rapport à l'architecture de ConFract, qui peut intercepter les interactions dans les membranes de tout composant, primitif ou composite. Considérons l'architecture d'une opération Jabyce, décrite dans la section 3.2.6. Une opération est un composant composite dont une partie seulement des interfaces internes sont liées à un sous-composant transformateur. La plupart des interfaces client internes sont directement liées aux interfaces serveur internes correspondantes. En considérant qu'un composant transformateur est un composant primitif, le code source de ce composant peut être transformé par iContract pour intégrer des vérifications d'assertions. Ainsi, dans une opération, seules les interfaces des opérations qui sont liées à un sous-composant transformateur peuvent être soumises à des vérifications de contrats. Des contrats sur les interfaces internes directement liées ne peuvent pas être vérifiés. Le problème est alors qu'une erreur est détectée en vérifiant une assertion lors de l'occurrence d'une interaction sur une telle interface, il n'est ainsi pas possible de localiser le composant fautif.

Par exemple, la figure 4.9 illustre une chaîne de trois opérations Jabyce. Dans ces opérations, seuls les composants transformateurs sont primitifs et donc mettent en œuvre une vérification de contrats avec iContract. Si le transformateur dans l'opération op3 détecte une violation de contrat à l'occurrence d'une interaction sur l'interface b de l'opération, cette interaction fautive a pu être initiée par op1 ou op2. En effet, l'interface b de l'opération op2 n'est pas liée à un sous-composant transformateur, et donc ne peut pas vérifier de contrat. Une interaction fautive sur l'interface b de op2 peut donc traverser op2 sans vérification. Il n'est donc pas possible de localiser l'opération Jabyce fautive, c'est-à-dire celle qui a initié l'interaction. Le problème fondamental est que les approches de contrats sur les objets sont très liées au mécanisme d'héritage entre classes et interfaces. Il n'est pas possible de prendre en compte la composition hiérarchique des composants qui est offerte par les modèles à composants comme Fractal.

Formalisme de iContract

Nous utilisons le formalisme de iContract pour définir des assertions exécutables dans le code source des interfaces Java qui sont la signature des interfaces des composants, par exemple les interfaces Java FieldFactory, MethodFactory. Les contrats que nous spécifions avec iContract sont donc des *contrats d'interface*, dans la terminologie du modèle de contrat ConFract.

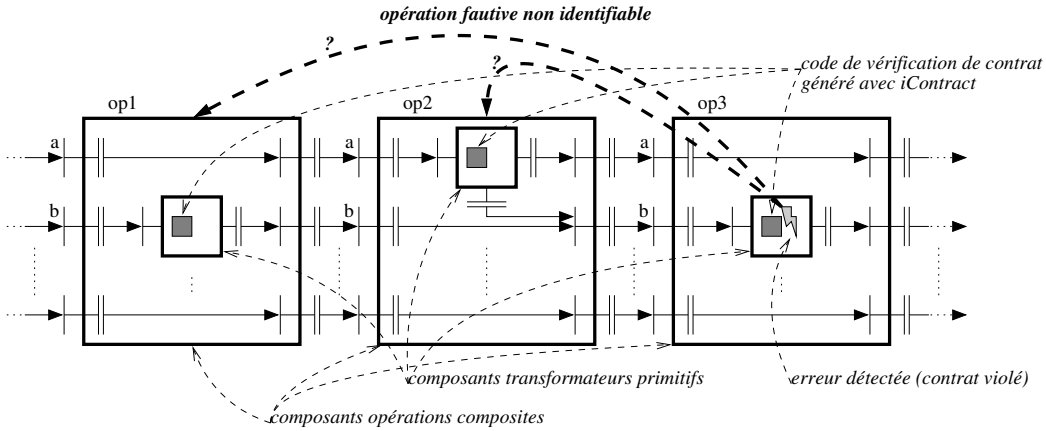


FIG. 4.9 : Impossibilité de localisation des fautes avec les contrats sur objets

Cependant, comme les spécifications iContract sont mélangées avec le code source des interfaces, il n'est pas possible qu'une spécification concerne plusieurs interfaces Java qui ne sont pas liées par héritage, ce qui est notamment le cas dans Jabyce. Il n'est donc pas possible de spécifier des *contrats de composition* avec iContract.

De plus, comme indiqué dans la section précédente, la vérification de contrats de correction sémantique des programmes impose de pouvoir raisonner sur un *historique des interactions*, pour détecter des erreurs liées à plusieurs interactions représentant des éléments d'un même programme. Or, il n'est pas possible avec iContract d'utiliser, dans les assertions, d'autres éléments que les attributs et méthodes définies dans la classe ou l'interface, et les paramètres des appels de méthodes. Il n'est pas possible de définir un état propre à un contrat iContract.

Dans notre contexte, ces contraintes limitent notre utilisation de iContract à la spécification de contrats de correction syntaxique des programmes représentés. Il n'est pas possible d'utiliser iContract pour la spécification et la vérification de contrats de correction sémantique.

Synthèse

iContract, et par extension la conception par contrats assertionnels sur des objets, est un formalisme simple et éprouvé, que nous utilisons pour la spécification et la vérification de contrats de correction syntaxique des programmes transformés. Nous avons généralisé l'utilisation de ce formalisme pour la spécification de contrats sur toutes les interfaces des composants dans Jabyce, notamment en générant automatiquement des spécifications iContract avec notre logiciel Jaidee (cf. la section 3.3.1).

Cependant, le formalisme iContract ne nous permet pas de spécifier des contrats de correction sémantique des programmes, car iContract ne permet pas de spécifier des contrats de composition, ni de raisonner sur un état propre à chaque contrat. De plus, l'architecture de iContract n'est pas adaptée aux caractéristiques du modèle Fractal (composants composites), ni à nos besoins en termes de modularité.

Les deux sections suivantes présentent des perspectives d'utilisation d'autres formalismes plus adaptés aux caractéristiques du modèle Fractal et à l'architecture de ConFract, et qui permettent la spécification de contrats de correction sémantique de programmes.

4.3.3 CCL-J : spécification par assertions exécutables

CCL-J (*Component Constraint Language - Java*) est un langage de spécification de contrats sur des composants, basé sur des assertions exécutables, et dérivé du langage OCL (*Object Constraint Language*) [OMG03] de spécification de contrats sur des objets. CCL-J a été défini par l'équipe OCL du laboratoire I3S, en collaboration avec le laboratoire DTL/ASR de France Telecom R&D, dans le cadre de ConFract. CCL-J est pour l'instant le seul formalisme de spécification de contrats utilisable avec ConFract.

CCL-J et iContract étant tous deux dérivés de OCL, la syntaxe des clauses dans les spécifications est très proche dans les deux langages. Cependant, CCL-J supporte à la fois la spécification de contrats d'interface et de composition de composants Fractal, alors que iContract supporte seulement la spécification de contrats d'interface.

La syntaxe de CCL-J est définie dans [CO03]. La terminologie utilisée est principalement celle du modèle de contrat ConFract, décrit dans la section 4.3.1. Dans CCL-J, toute spécification comprend un contexte, et des clauses (parties de spécifications). C'est le contexte d'une spécification qui définit le type de contrat spécifié (interface ou composition). Les spécifications de contrats d'interface se font dans le contexte d'une interface Java ou d'une méthode d'une interface Java. Les spécifications de contrats de composition se font dans le contexte d'un type de composant Fractal. Le contexte définit ce qui peut être référencé dans les clauses. Par exemple, dans le contexte d'une méthode, seuls les arguments et la valeur de retour peuvent être référencés. Chaque clause comprend un *type de spécification*, et une assertion exécutable. Dans CCL-J, les types généraux de spécification sont les pré-conditions (pre), les post-conditions (post) et les invariants (inv). Dans le contexte d'une interface Java, seuls des invariants peuvent être définis. Le langage des assertions est le langage des expressions booléennes Java, étendu avec des opérateurs spécifiques. Par exemple, l'opérateur forEach permet d'itérer sur des collections d'objets, et l'opérateur booléen implies est tiré des langages OCL et Eiffel.

Contrat d'interface en CCL-J

Un exemple de contrat d'interface, pour l'interface Java FieldAccessInstructionFactory, est illustré dans la figure 4.10. Cette spécification est identique à la spécification donnée dans le formalisme iContract dans le programme 3.2, et décrite dans les sections 3.2.5 et 3.3.1. Cette spécification spécifie un contrat de correction syntaxique des instructions d'accès aux attributs qui sont représentées en appelant cette méthode. La correction concerne ici la validité des paramètres passés dans l'appel.

Contrat de composition en CCL-J

Une spécification de contrat d'interface comme celle de la figure 4.10 n'est pas suffisante pour définir toutes les clauses sur la méthode createFieldAccessInstruction(...). La figure 4.11 donne une spécification de contrat de composition, qui porte sur toutes les interfaces serveur d'un composant opération ou sérialiseur (type OperationServerSide), qui est plus complète que la spécification de contrat d'interface donnée dans la figure 4.10. La dernière clause spécifie qu'on ne peut représenter une instruction que si la représentation de la méthode, identifiée par l'objet contexte passé en paramètre, n'est pas encore terminée. L'expression dans cette clause

```
context void FieldAccessInstructionFactory
    .createFieldAccessInstruction(
    MethodBuildingContext compositeMethodBuildingContext,
    ObjectType type, String fieldName,
    boolean fieldIsStatic, FieldDescriptor fieldType,
    boolean loadOrStore)
  pre compositeMethodBuildingContext != null;
  pre type != null;
  pre fieldName != null;
  pre fieldType != null;
```

FIG. 4.10 : Spécification de contrat CCL-J pour l'interface Java FieldAccessInstructionFactory

comprend un appel à la méthode isMethodBuildingFinished(...) sur l'interface method-factory : cet appel doit retourner false pour l'objet contexte en paramètre. C'est une spécification de contrat de composition, car elle référence plusieurs interfaces d'un même composant (create-field-acces-instruction-factory et method-factory). Une telle spécification n'est pas exprimable dans le formalisme iContract, présenté dans la section précédente.

```
on <OperationServerSide>
context void field-access-instruction-factory
    .createFieldAccessInstruction(
    MethodBuildingContext compositeMethodBuildingContext,
    ObjectType type, String fieldName,
    boolean fieldIsStatic, FieldDescriptor fieldType,
    boolean loadOrStore)
  pre compositeMethodBuildingContext != null;
  pre type != null;
  pre fieldName != null;
  pre fieldType != null;
  pre !method-factory.isMethodBuildingFinished(
    compositeMethodBuildingContext);
```

FIG. 4.11 : Spécification partielle de contrat de composition CCL-J pour les interfaces serveur d'une opération Jabyce

Historique des interactions avec les variables fantômes

Une caractéristique de CCL-J significative dans notre contexte est la possibilité de spécifier un état propre à chaque contrat. L'état d'un contrat est spécifié sous la forme de *variables* dans une spécification, qui ne sont manipulables que dans cette spécification. Ces variables n'étant pas manipulables hors de la spécification qui les définit, elles sont appellées *variables fantômes* (*ghost variables*) dans la littérature [CLSE03].[1] Les variables fantômes ont été introduites dans CCL-J, dans le cadre de la collaboration entre le laboratoire I3S et le laboratoire DTL/ASR de France Telecom R&D, pour pouvoir utiliser CCL-J dans Jabyce. Cette approche est classique dans les formalismes de spécification, et a été pour la première fois introduite dans le langage

[1] Dans la spécification de CCL-J, elles sont appelées improprement *variables de modèle*.

de spécification ANNA (*ANNotated Ada*) [LvHKBO84], et est offerte par les formalismes plus récents comme JML (*Java Modelling Language*) [CLSE03, LPC+04].

Comme expliqué dans [CLSE03] et [BP03], les concepts de variables de modèle (*model variables*) et de variables fantômes (*ghost variables*) ont été introduits pour modéliser l'état d'objets lorsqu'une classe n'offre pas suffisamment d'information sur l'état des objets. En effet, l'état des objets est accessible seulement à travers les méthodes et attributs accessibles hors de la classe (modificateur public). Typiquement, une classe définit explicitement des accesseurs (méthodes « get ») pour accéder directement à l'état des objets. Cependant, une classe n'offre pas toujours suffisamment d'information à travers les méthodes et attributs accessibles. Dans ce cas, les variables fantômes et les variables de modèle permettent de modéliser l'état des objets dans les spécifications, sans utiliser des méthodes et des attributs définis dans la classe pour accéder à l'état réel des objets. L'avantage principal est l'application du principe de masquage de l'information (*information hiding*). En effet, ce mécanisme limite la nécessité d'introduire dans les classes de nouveaux attributs et méthodes seulement pour les utiliser dans des spécifications de comportement, ce qui diminue la *complexité inutile* du code.

La différence entre variables de modèle et variables fantômes est le mécanisme offert pour la modification des valeurs des variables pendant l'évaluation des contrats [CLSE03]. Les variables fantômes sont modifiées explicitement par des fragments de code dans les spécifications, par exemple par du code source Java, qui sont exécutés avant ou après l'exécution de certaines méthodes. L'inconvénient est que les fragments de code qui spécifient une variable sont dispersés dans toute une spécification, et qu'il est difficile de raisonner sur ces fragments de code. En revanche, une variable de modèle est calculée comme le résultat de l'exécution d'une fonction pure, appelée *fonction d'abstraction* (*abstraction function*). Ainsi, la spécification d'une variable de modèle est plus compacte, plus simple et plus compréhensible qu'une spécification de variable fantôme. De plus, il est plus réaliste d'envisager des vérifications formelles des spécifications de variables de modèle, qu'avec des variables fantômes.

Pour l'instant, CCL-J offre seulement un mécanisme de variables fantômes, et l'une des perspectives de CCL-J est d'offrir un mécanisme de variables de modèle, plus puissant mais plus difficile à mettre en œuvre. Il est envisageable de s'inspirer des mécanismes de variables de modèles introduits dans JML [CLSE03, LPC+04].

Utilisation des variables fantômes pour Jabyce

La figure 4.12 illustre une spécification du comportement des interfaces de type MethodFactory. Le comportement spécifié est le protocole des appels à createMethod(...) et finishMethodBuilding(...), comme décrit dans la section 3.2.5. Un appel à createMethod(...) commence la représentation d'une méthode, et retourne un objet contexte qui doit être utilisé comme paramètre pour la représentation des éléments dans la méthode. La représentation de la méthode doit être explicitement terminée par un appel à finishMethodBuilding(...), avec cet objet contexte en paramètre.

La spécification de ce comportement se base sur la définition d'une variable fantôme qui est une collection des objets contextes des méthodes en cours de représentation. Une clause model permet de définir une variable fantôme dans le contexte d'une signature d'interface. Des clauses !!inv et inv!! permettent d'initialiser et de finaliser, respectivement, une ou plusieurs

variables fantômes. Une clause !!inv (respectivement inv!!) contient du code Java qui est exécuté avant (respectivement après) toute vérification d'invariant. Une variable fantôme peut être utilisée dans toute clause inv, pre ou post. De plus, des clauses body permettent de faire varier l'état de variables fantômes, dans le contexte d'une méthode. Une clause body contient du code Java qui est exécuté après que toutes les clauses pre aient été vérifiées sur la méthode, et avant que les clauses post soient vérifiées. Les méthodes qui n'ont pas de clause body ne modifient pas les variables fantômes.

```
context MethodFactory
  model Set methodSet;
  !!inv { methodSet = new HashSet(); };
  inv!!  { methodSet.clear(); methodSet = null; };
  inv methodSet != null;

context MethodBuildingContext MethodFactory.createMethod(
    ClassBuildingContext compositeClassBuildingContext, ...)
  body {
    if (return != null) {
      methodSet.add(return);
    }
  };
  post return != null;
  post methodSet.contains(return);

context void MethodFactory.finishMethodBuilding(
    MethodBuildingContext buildingContext)
  pre buildingContext != null;
  pre methodSet.contains(buildingContext);
  body {
    methodSet.remove(buildingContext);
  };
  post !  methodSet.contains(buildingContext);

context boolean MethodFactory.isMethodBuildingFinished(
    MethodBuildingContext buildingContext)
  pre buildingContext != null;
  post return == methodSet.contains(buildingContext);
```

FIG. 4.12 : Spécification de contrat CCL-J pour l'interface Java MethodFactory

Dans cet exemple, l'ensemble des objets contextes des méthodes en cours de représentation ne fait typiquement pas partie de l'état des composants qui offrent une interface de type MethodFactory. Cet état n'est donc pas accessible explicitement par l'interface MethodFactory. Cet exemple montre l'intérêt des variables fantômes, qui permettent de raisonner sur un tel état même s'il n'est pas offert par les composants.

Propriétés de vivacité pour Jabyce

Les exemples ci-dessus ont montré que CCL-J offre des abstractions (contrats de composition, et variables fantômes) indispensables pour son utilisation dans Jabyce. Cependant, ces ab-

stractions seules ne sont pas suffisantes pour spécifier des *propriétés de vivacité*. En effet, d'un point de vue général, CCL-J permet de spécifier des *propriétés de sûreté*, pour garantir que des « mauvaises » interactions ne peuvent pas se produire. En revanche, il n'est pas possible d'exprimer directement des *propriétés de vivacité*, pour garantir que des « bonnes » interactions vont se produire. Les propriétés de vivacité concernent principalement les contrats de correction sémantique, qui considèrent les dépendance entre les éléments d'un programme.

Par exemple, si dans un programme transformé une instruction accède à l'attribut someAttr de la classe SomeClass, il est nécessaire de vérifier que l'attribut someAttr est ou sera effectivement représenté dans le programme. Si sommeAttr n'a pas encore été représenté lorsque l'instruction d'accès est représentée, une propriété de vivacité est que cet attribut devra être représenté avant la fin de la représentation du programme. Cet exemple montre qu'il est nécessaire de pouvoir borner l'observation du système dans le temps pour pouvoir vérifier les propriétés de vivacité. Plus précisément, il est nécessaire de borner l'observation du système à la séquence des interactions qui représentent toutes les classes Java compilées d'un même programme dans son ensemble. Or, il n'y a pas de type d'élément à un niveau au-dessus des classes Java dans le modèle conceptuel de Jabyce, cf. la section 3.2.3. Il n'y a donc aucune notification explicite de la fin de la représentation d'un programme.

Nous avons donc défini l'interface Observer dans Jabyce, donnée dans le programme 4.1, qui permet de commencer et de terminer explicitement l'observation des interactions, respectivement avant et après la transformation de l'ensemble des classes compilées d'un programme. Cette interface devrait être offerte par un objet ou composant très simple dans la membrane de chaque composant observé, et interceptée par l'intercepteur de ConFract comme les autres interfaces fonctionnelles interceptées (cf. la figure 4.7). Ce mécanisme n'est pas encore implanté actuellement.

```
public interface Observer {
  void beginObservation();
  void endObservation();
  boolean isObserving();
}
```

PROG. 4.1 : Interface Observer

Grâce à l'introduction de cette interface, il est possible avec CCL-J d'exprimer des propriétés de vivacité sous la forme de clauses de pré-condition sur la méthode endObservation(). Ainsi, la figure 4.13 donne la spécification de contrat pour la propriété de vivacité décrite ci-dessus, concernant les dépendances entre attributs et instructions d'accès aux attributs. Cette spécification repose sur la définition de deux variables fantômes : l'ensemble des noms d'attributs déjà représentés (fieldSet), et l'ensemble des noms d'attributs accédés mais pas encore représentés (unresolvedFieldAccessSet). Une clause de pré-condition de endObservation() spécifie que ce deuxième ensemble doit être vide à l'occurrence d'un appel à endObservation().

Synthèse

Les exemples de spécification donnés dans cette section montrent que CCL-J offre des abstractions (contrats de composition et variables fantômes) qui permettent d'utiliser CCL-J dans

```
context FieldFactory
  model Set fieldSet;
  model Set unresolvedFieldAccessSet;
  !!inv {
    fieldSet = new HashSet();
    unresolvedFieldAccessSet = new HashSet();
  };
  inv!!  {
    fieldSet.clear(); fieldSet = null;
    unresolvedFieldAccessSet.clear();
    unresolvedFieldAccessSet = null;
  };
  inv fieldSet != null;
  inv unresolvedFieldAccessSet != null;

on <ObservedOperationServerSide>
context void field-access-instruction-factory
    .createFieldAccessInstruction(...,
    ObjectType type, String fieldName,
    ..., FieldDescriptor fieldType, ...)
  body {
    String fname = fieldType.toString()
      + type.toString() + fieldName;
    if (!field-factory.fieldSet.contains(fname) &&
        !field-factory.unresolvedFieldAccessSet
          .contains(fname)) {
      field-factory.unresolvedFieldAccessSet.add(fname);
    }
  };

on <ObservedOperationServerSide>
context FieldBuildingContext field-factory.createField(
    ClassBuildingContext compositeClassBuildingContext,
    String name, FieldDescriptor type, ...)
  body {
    String className = class-factory.getBuiltClassClassType(
      compositeClassBuildingContext).toString();
    String fname = type.toString() + className + name;
    if (!fieldSet.contains(fname)) {
      fieldSet.add(fname);
      unresolvedFieldAccessSet.remove(fname);
    }
  };

on <ObservedOperationServerSide>
context void observer.endObservation()
  pre field-factory.unresolvedFieldAccessSet.isEmpty();
  body {
    field-factory.fieldSet.clear();
  };
```

Fig. 4.13 : Spécification de propriété de vivacité en CCL-J pour les dépendances entre instructions et attributs

Jabyce pour la spécification de contrats de correction syntaxique et - surtout - de correction sémantique des programmes transformés. C'est d'ailleurs l'utilisation de CCL-J dans Jabyce qui a motivé l'introduction des variables fantômes dans CCL-J. De plus, CCL-J a été conçu simultanément avec ConFract, et est donc naturellement le formalisme le mieux adapté au modèle et à l'architecture de ConFract.

Cependant, l'exemple de spécification de la figure 4.13, qui utilise des variables fantômes, montre que ce type de variable introduit une grande complexité dans la spécification des contrats. Il est nécessaire d'écrire beaucoup de code dans des clauses `body` qui manipulent les variables fantômes, pour pouvoir spécifier des contrats intéressants. La multiplication des clauses `body` rend les spécifications difficilement lisibles. C'est pourquoi l'une des perspectives des travaux sur CCL-J est d'introduire des variables de modèle similaires à celles de JML, où les variables sont spécifiées par des fonctions d'abstraction et non plus par des fragments de code. C'est aussi pourquoi nous considérons pour l'instant CCL-J uniquement en remplacement de iContract pour la spécification de contrats de correction syntaxique, et pour la spécification de contrats simples de correction sémantique.

4.3.4 TLA : spécification en logique temporelle

Dans cette section, nous présentons le formalisme de modélisation de comportement utilisé dans le système de gestion de contrat TLO (*Temporal Logic Observer*), dont l'architecture est présentée dans la section 4.2.2. TLO est développé dans le laboratoire DTL/ASR de France Telecom R&D. Cette approche pour la détection d'erreur se base sur une comparaison du comportement observé d'un système avec un modèle formel de ce comportement pour lequel sont validées des propriétés temporelles. Cette section montre que le formalisme utilisé dans TLO, TLA (*the Temporal Logic of Action*) est plus adapté que CCL-J pour la spécification de contrats qui maintiennent un état complexe.

Approche générale

Cette approche est une application du domaine de la spécification formelle des systèmes communicants [Mon96]. Dans ce domaine, les propriétés vérifiées sont généralement liées au contrôle, c'est-à-dire à la synchronisation, à l'absence d'interblocage, etc. Un langage de modélisation de comportement est généralement basé sur le concept d'*automate communicant*, permettant de décrire pour chaque système un ensemble d'états, un ensemble d'actions, et pour chaque action une relation binaire sur l'ensemble des états. Les propriétés à vérifier sont exprimées dans un deuxième langage, qui est généralement une *logique temporelle*. La logique temporelle introduit notamment des opérateurs modaux, tels que l'opérateur « toujours » généralement noté $\Box$, qui signifie que la proposition à laquelle il est appliqué reste toujours vraie pendant l'exécution, et l'opérateur « fatalement » généralement noté $\Diamond$ qui signifie que la proposition sera vérifiée à un moment donné. L'opérateur « toujours » permet de spécifier des propriétés de *sûreté*, l'opérateur « fatalement » permet de spécifier des propriétés de *vivacité*.

En comparaison aux approches de spécifications par assertions exécutables, comme iContract et CCL-J présentés dans les sections 4.3.2 et 4.3.3, qui *mettent l'accent* sur la spécification des comportements illicites, **la logique temporelle *met l'accent* sur la spécification des**

comportements licites [Mon96]. En effet, dans les approches par assertions, si une spécification ne définit aucune assertion, tout comportement est considéré comme licite, et l'ajout d'assertions restreint l'ensemble des comportements licites. Au contraire, dans les approches par automates communicants, l'approche générale consiste d'abord à définir un automate constitué d'un seul état et d'aucune transition, pour lequel tout comportement est considéré comme illicite, puis à ajouter des états et des transitions dans le modèle, pour agrandir l'ensemble des comportements licites.

La *vérification par évaluation sur un modèle* (*model checking*) consiste à vérifier si un modèle formel vérifie une propriété qui est une formule de logique temporelle. Cette vérification est effectuée en parcourant le graphe des états et transitions possibles du modèle, et en vérifiant que chaque état possible vérifie la propriété [CES86]. L'intérêt est de pouvoir effectuer automatiquement et rapidement ces vérifications et d'obtenir une trace détaillée de tout comportement fautif accepté par un modèle.

L'*observation* d'un système, introduite de manière générale dans la section 4.2.2, est redéfinie plus précisément ici comme la comparaison du comportement observé du système, avec un modèle formel pour lequel ont été vérifiées des propriétés temporelles par *model checking*. Tout le comportement observable du système n'est pas nécessairement complétement validé. Le modèle peut ne concerner que certains aspects du comportement, notamment parce que le comportement est en général trop complexe pour être complétement observé (un trop grand nombre d'événements se produisent) [DJC94]. Nous avons ainsi suivi une approche *modulaire*, comme indiqué dans la section 4.3.1, en spécifiant plusieurs modèles pour des aspects différents de la correction des programmes. Ainsi, un utilisateur peut choisir facilement les aspects à vérifier sur une configuration d'opérations de transformation donnée.

Observation avec TLO

L'originalité de TLO est de se baser sur le langage TLA+, une extension du langage TLA (*Temporal Logic of Actions*) [Lam94, Lam99], dans le cadre de la validation de systèmes. L'avantage de TLA est d'offrir un formalisme unique pour spécifier à la fois le modèle d'un système, et les propriétés temporelles à vérifier, ce qui simplifie les spécifications. Une originalité de TLO est ainsi de permettre, à partir d'un même modèle formel en TLA, à la fois 1) une validation du modèle, statiquement, et 2) une validation des systèmes pendant leur exécution. Ces deux activités sont réalisées par *model-checking*. La description de TLO dans cette section est principalement tirée de [RC03].

L'élément du système qui détermine automatiquement si le comportement observé lors d'un test correspond à une spécification formelle, est appelé un *oracle de test* [RAO92]. Un oracle peut être généré automatiquement à partir d'une spécification, de même qu'un jeu de tests garantissant une couverture suffisante du système. Dans notre contexte, un jeu de tests serait un programme Java, sous la forme d'un ensemble de classes Java compilées, qui serait transformé par une chaîne d'opérations de transformation. Un tel jeu de tests pertinent est a priori difficile à générer. Nous abordons donc seulement la spécification d'oracles à partir de spécifications en TLA dans le cadre de TLO, et pas de jeux de tests.

En TLA, une spécification du comportement correct d'un système est généralement donnée sous la forme canonique donnée dans la spécification (4.1), où *Init* est un prédicat d'état qui caractérise les états initiaux du système, *Next* est une formule d'action qui caractérise les

transitions possibles entre états, x est le tuple des variables considérées qui modélisent l'état du système, et L est une formule de logique temporelle qui décrit des propriétés de vivacité. La formule $\Box[Next]_x$ permet en fait de spécifier des propriétés de sûreté : *Next* doit être toujours vraie.

$$Spec \triangleq Init \wedge \Box[Next]_x \wedge L \tag{4.1}$$

Dans TLO, le comportement effectivement observé du système est exprimé sous la forme d'une spécification TLA similaire, *Obs*, dont la forme canonique est donnée dans la spécification (4.2).

$$Obs \triangleq InitObs \wedge \Box[NextObs]_x \tag{4.2}$$

La spécification (4.3) de l'oracle correspondant est composée dans TLO du modèle formel du système *Spec*, et de l'observateur *Obs*. Ainsi, à la fois le comportement observé du système, le comportement attendu du système et l'oracle de test sont spécifiés dans le même formalisme, en l'occurrence TLA.

$$Oracle \triangleq (InitObs \wedge Init) \wedge \Box[(NextObs \wedge Next)]_x \tag{4.3}$$

Dans l'observateur TLO, la spécification de l'oracle est *évaluée* à l'aide du *model checker* TLC (*Temporal Logic Checker*) [Lam02]. TLC (*Temporal Logic Checker*) est un model-checker en Java pour un sous-ensemble de TLA+, qui est développé par l'équipe de L. Lamport. L'évaluation d'un modèle, par un *model-checker* comme TLC, consiste à explorer toutes les traces possibles acceptées par le modèle, et à verifier que chaque trace vérifie des propriétés temporelles. Dans le cas de TLO, l'évaluation de la spécification de l'oracle est lancée au démarrage des transformations. Dans l'évaluation de la spécification *Obs*, TLC *bloque* sur l'évaluation d'*opérateurs spéciaux*, qui permettent à TLO de communiquer avec TLC. Les notifications d'occurrences d'interactions observées par TLO débloquent l'évaluation de ces opérateurs spéciaux, ce qui provoque des actions (transitions) dans le comportement observé *Obs*. Dans la spécification de l'oracle, les prédicats d'états initiaux et les relations de transitions forment des conjonctions. Ainsi, si une action évaluée dans *Obs* n'est pas conforme à la spécification *Spec*, une conjonction est évaluée comme fausse, et TLC retourne un *rapport de diagnostic d'erreur* exhibant une trace du comportement qui conduit à cette violation. Pour synthétiser, l'algorithme d'évaluation de TLC est utilisé pour *comparer* le comportement observé (spécification *Obs*) et le modèle spécifiant les comportements corrects (spécification *Spec*), et pour fournir un diagnostic en cas d'erreur.

Une telle utilisation d'un *model-checker*, pour la validation en ligne, est l'originalité de TLO. En effet, non seulement TLC est utilisé classiquement pour vérifier statiquement des propriétés temporelles sur le modèle, mais aussi le même model-checker TLC est utilisé pour vérifier que les traces d'exécution observées sont conformes à ce modèle. Dans les deux cas, la même approche (le *model-checking*) est utilisée, c'est-à-dire que TLC est utilisé pour évaluer (« exécuter ») un modèle sous la forme d'un automate.

Application à Jabyce

Notre démarche générale de spécification consiste à définir l'état du modèle de système comme *un graphe* correspondant à une *synthèse de la représentation interne du programme* représenté par des interactions entre deux composants. Ce graphe est une représentation synthétique d'un programme, parce qu'il contient seulement les informations nécessaires à la validation de l'aspect de correction considéré. Contrairement à CCL-J, présentés dans la section précédente, il est facile avec TLA de construire un tel historique synthétique des interactions observées. Comme nous proposons de généraliser cette démarche de modélisation, nous avons factorisé dans des modules TLA réutilisables des spécifications utiles pour la construction d'arbres, basés sur le module Graph non présenté ici. Ces modules TreeItf et Tree sont donnés dans la figure 4.14. Ces modules permettent de construire des arbres, où les nœuds modélisent des éléments de programmes, qui peuvent être composites (p.ex. une Class) ou non (p.ex. une FieldAccessInstruction), et les arcs modélisent des relations de composition entre éléments. Ce sont les mêmes types d'éléments que ceux définis dans le modèle conceptuel de Jabyce, cf. la section 3.2.3. Les actions licites dans les spécifications d'actions *CreatePrimitive*, *CreateComposite* et *FinishComposite* correspondent aux contraintes sur les interactions représentant des éléments composites décrites dans la section 3.2.5, pour garantir que les séquences d'interactions correspondent bien à un parcours d'arbre.

Pour chaque aspect de correction est défini un module TLA qui *raffine* ces modules. Par exemple, les modules InnerEnt et Inner, donnés dans la figure 4.15, modélisent de manière synthétique les relations de *classes internes* (*inner classes*) entre les classes Java compilées qui sont représentées par les interactions observées. Ce modèle spécifie l'une des contraintes sur les relations de classes internes définies dans la spécification de la JVM [LY99] : l'absence de cycles dans ces relations. C'est-à-dire que nous souhaitons vérifier qu'une classe Java ne peut pas être interne à une classe Java qui lui est interne. C'est une spécification de contrat de correction sémantique des programmes (cf. la section 4.1.2).

Dans les arbres construits par les actions, seuls deux types de nœuds sont considérés : *class*, qui correspond au type d'élément Class du modèle conceptuel de Jabyce, et *inner*, qui correspond au type d'élément InnerClassDeclaration. C'est donc un modèle synthétique des programmes transformés : ne sont pas modélisés par des nœuds des arbres les autres éléments de type Method, Field, etc. Les arcs modélisent les relations de composition entre les éléments de type Class et InnerClassDeclaration (spécification *Struct*) : il existe un arc si une déclaration de classe interne est représentée dans une classe.

Les noms des classes (*id*) sont conservés dans chaque nœud *class* et *inner*. La valeur de *id* dans un nœud *inner* est le nom d'une classe interne. Une relation de classe interne existe si un nœud *class* et un nœud *inner* ont la même valeur *id* (spécification *GraphById*). La spécification *NoCycle* est un invariant structurel qui vérifie qu'il n'existe pas de cycle dans les relations de composition et de classe interne, en se basant sur une comparaison des valeurs de *id*. Cet invariant correspond, dans la terminologie de ConFract, à une clause de spécification de contrat.

Le module InnerOracle, donné dans la figure 4.16, spécifie le comportement observé et l'oracle de test. La spécification du comportement observé repose sur l'utilisation de l'opérateur *ioStr*, qui est implanté par l'observateur de TLO directement en Java. A chaque occurrence d'un appel de méthode Jabyce createClass, finishClassBuilding ou createInnerClassDeclaration, qui représente

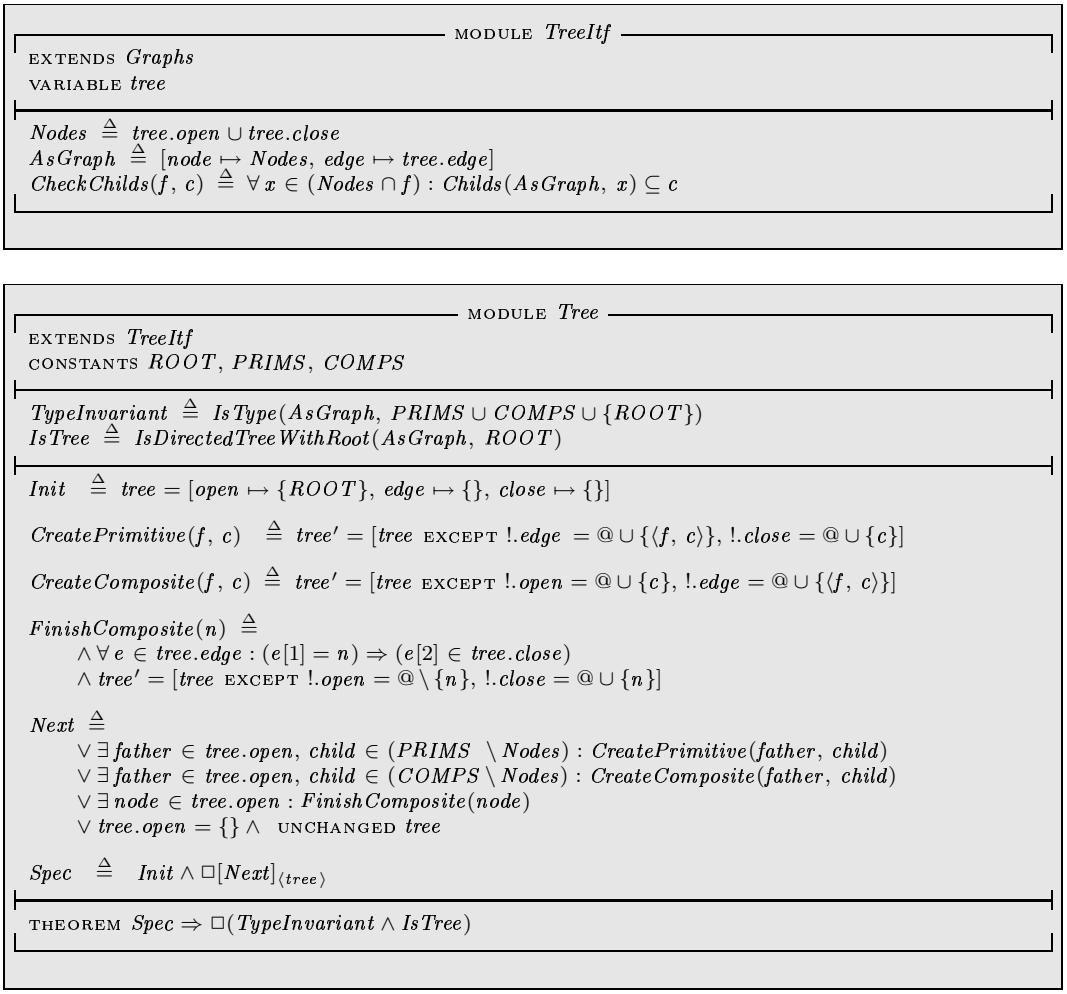


FIG. 4.14 : Les modules TLA TreeItf et Tree

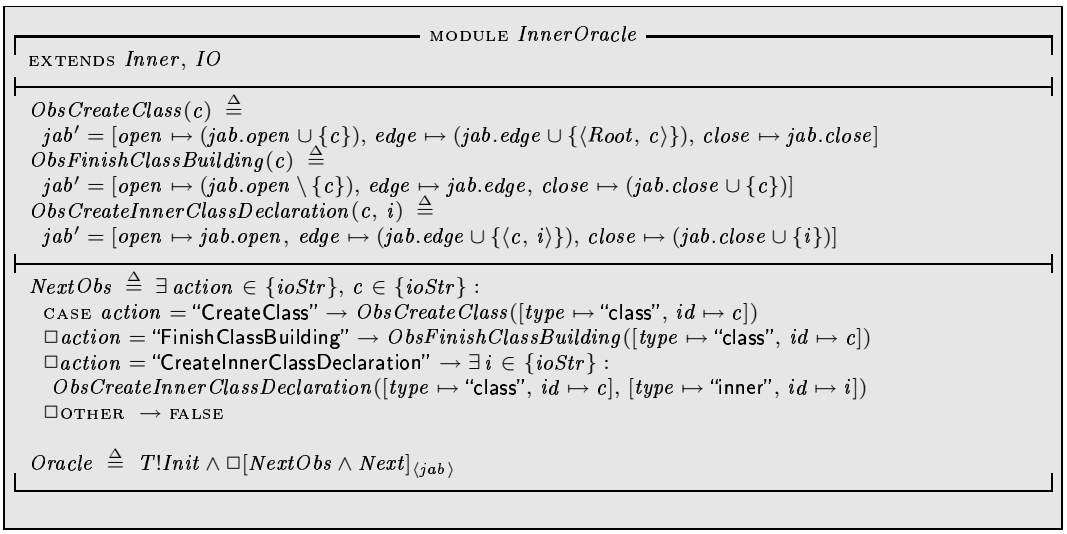
```
MODULE InnerEnt
EXTENDS TreeItf
CONSTANTS CLASSES_ID
----------
Root ≜ [type ↦ "root", id ↦ "root"]
Classes ≜ [type : {"class"}, id : CLASSES_ID]
Inners ≜ [type : {"inner"}, id : CLASSES_ID]
----------
GraphById ≜ [node ↦ {x.id : x ∈ Nodes}, edge ↦ {⟨y[1].id, y[2].id⟩ : y ∈ tree.edge}]
Struct ≜ CheckChilds({Root}, Classes) ∧ CheckChilds(Classes, Inners)
NoCycle ≜ ∀ i ∈ Nodes : ¬IsConnected(GraphById, i.id, i.id)
Closure ≜ (tree.open = {}) ⇒ (Classes ⊆ tree.close)
```

```
MODULE Inner
EXTENDS InnerEnt
----------
T ≜ INSTANCE Tree WITH ROOT ← Root, COMPS ← Classes, PRIMS ← Inners
----------
Next ≜
    ∧ T!Next
    ∧ Struct'
    ∧ NoCycle'
    ∧ Closure'

Spec ≜ T!Init ∧ □[Next]⟨tree⟩
----------
THEOREM Spec ⇒ □(T!TypeInvariant ∧ T!IsTree)
THEOREM Spec ⇒ T!Spec
```

FIG. 4.15 : Les modules TLA InnerEnt et Inner

un élément de type Class ou InnerClassDeclaration, l'observateur de TLO communique à TLC, via *ioStr*, le nom de la méthode Jabyce (variable *action*) et le nom de classe passé en paramètre (variable c), au fur et à mesure de l'évaluation du modèle. La spécification *Oracle* vérifie que ce comportement observé correspond à une instance du module de spécification Inner, conformément à la forme canonique donnée dans la formule 4.3.



MODULE *InnerOracle*

EXTENDS *Inner*, *IO*

$ObsCreateClass(c) \triangleq$
$jab' = [open \mapsto (jab.open \cup \{c\}),\ edge \mapsto (jab.edge \cup \{\langle Root,\ c\rangle\}),\ close \mapsto jab.close]$
$ObsFinishClassBuilding(c) \triangleq$
$jab' = [open \mapsto (jab.open \setminus \{c\}),\ edge \mapsto jab.edge,\ close \mapsto (jab.close \cup \{c\})]$
$ObsCreateInnerClassDeclaration(c,\ i) \triangleq$
$jab' = [open \mapsto jab.open,\ edge \mapsto (jab.edge \cup \{\langle c,\ i\rangle\}),\ close \mapsto (jab.close \cup \{i\})]$

$NextObs \triangleq \exists\, action \in \{ioStr\},\ c \in \{ioStr\}$:
CASE $action =$ "CreateClass" $\rightarrow ObsCreateClass([type \mapsto$ "class", $id \mapsto c])$
$\square action =$ "FinishClassBuilding" $\rightarrow ObsFinishClassBuilding([type \mapsto$ "class", $id \mapsto c])$
$\square action =$ "CreateInnerClassDeclaration" $\rightarrow \exists\, i \in \{ioStr\}$:
$ObsCreateInnerClassDeclaration([type \mapsto$ "class", $id \mapsto c],\ [type \mapsto$ "inner", $id \mapsto i])$
$\square$OTHER $\rightarrow$ FALSE

$Oracle \triangleq T!Init \wedge \square[NextObs \wedge Next]_{\langle jab\rangle}$

FIG. 4.16 : Le module TLA InnerOracle

Synthèse

L'exemple de spécification TLA ci-dessus montre que la spécification de contrats qui maintiennent un état complexe (un historique complexe des interactions), est plus concise et expressive en TLA qu'en CCL-J. Cela peut être vérifié en comparant les spécifications dans les figures 4.15 et 4.13 qui concernent des propriétés similaires, en l'occurrence des dépendances sémantiques entre des éléments de programmes. Notre approche de modélisation en TLA pour Jabyce est également adaptée à la spécification de contrats de composition, qui considèrent plusieurs interfaces.

Un inconvénient est que TLA n'est pas un formalisme adapté à l'expression de propriétés de correction des programmes au niveau de la syntaxe. En effet, les types de données offerts par TLA sont trop basiques pour pouvoir exprimer simplement des spécifications telles que celle donnée dans la figure 4.10 en CCL-J.

En conclusion, nous proposons d'utiliser TLA uniquement pour la spécification de contrats de correction sémantique qui nécessitent de maintenir un état complexe dans les contrats.

4.3.5 Comparaison avec les travaux existants

Un problème pour la vérification de programmes compilés Java, est que la spécification du processus de vérification est intégralement décrite en langue naturelle (en anglais), de manière semi-formelle. Il existe de nombreux travaux de formalisation concernant le code compilé Java,

dont un grand nombre sont recensés dans [Ler03]. L'objectif commun à tous ces travaux est de pouvoir garantir que toute exécution de programme sur une JVM respecte une spécification formelle de la JVM. Nous distinguons ces travaux par les points de vue adoptés : 1) les travaux mettant l'accent sur la correction des implantations de JVM, et 2) les travaux mettant l'accent sur la vérification des programmes Java compilés.

Parmi les travaux de formalisation de la JVM, un des travaux les plus significatifs est la dJVM (*defensive JVM*) [Coh97], une spécification formelle exécutable d'une implantation d'un sous-ensemble de la spécification de la JVM, dans le langage ACL2 (*A Computational Logic for Applicative Common Lisp*). ACL2 est une version de Common Lisp dotée d'un langage de programmation fonctionnel, d'une logique mathématique du premier ordre, et d'un prouveur automatique de théorème. Ainsi, des propriétés de sûreté de typage (*type safety*) peuvent être prouvées pour cette implantation, pour garantir que seulement des programmes respectant la spécification peuvent être exécutés.

Les travaux sur la vérification de bytecode Java sont des formalisations de l'algorithme de vérification décrit dans la spécification de la JVM [LY99]. La plupart de ces travaux sont basés sur des techniques d'analyse de flot de données (*data flow*), et ont pour objectif de valider formellement les implantations d'algorithmes de vérification [Ler03]. Cet algorithme est classiquement itératif, bien que certains travaux proposent des versions non itératives plus performantes [MH02b]. Cependant, l'utilisation d'un *algorithme* pour la vérification n'est pas appropriée dans le contexte de la validation dans Jabyce, parce que notre objectif est de valider chaque interaction représentant un élément d'un programme, au moment de l'occurrence de cette interaction, pour diagnostiquer précisément les erreurs. Il est très difficile de modifier une telle spécification d'algorithme pour l'adapter à cette caractéristique. En revanche, d'autres travaux récents de formalisation ne se basent pas sur l'analyse de flot de données. Notamment, [BFG03] propose une formalisation en logique temporelle de la sémantique statique du code compilé Java dans les implantations de méthodes, et une technique basée sur la validation sur modèle (*model checking*) pour la vérification du code. Notre proposition se rapproche de cette approche, en ce qui concerne le formalisme (automates et logique temporelle). Cependant, dans l'approche de [BFG03], chaque implantation de méthode est formalisée sous la forme d'un automate, où chaque instruction est un état, et les relations de branchement entre instructions sont les transitions. Les propriétés de correction des implantations de méthodes sont exprimées comme des propriétés temporelles, et sont vérifiées par *model-checking*. Par exemple, il est possible de vérifier la propriété que la pile d'exécution à l'exécution d'une instruction doit contenir les mêmes types de données quelque soit le chemin d'exécution vers cette instruction. Or, dans Jabyce / TLA, l'état du modèle correspond à un historique des éléments de programme représentés, et les actions (transitions) correspondent à l'occurrence d'une interaction représentant un élément de programme, et ne représentent pas des branchements entre instructions comme dans [BFG03]. Les deux approches de modélisation sont donc trop différentes pour nous puissions rapprocher ces travaux. Il existe aussi des approches utilisant le *model-checking* pour vérifier des propriétés de plus haut niveau que la correction des programmes, pour vérifier par exemple que deux *applets* ne peuvent échanger que des données autorisées [BCM+01].

Les travaux de formalisation de la JVM portent généralement sur la spécification formelle de la *sémantique dynamique* des programmes, c'est-à-dire de l'effet de l'exécution des programmes, i.e. des instructions. [ZX04] est une étude générale des formalismes de spécification de sémantique dynamique, non spécifique à la JVM. Pour le bytecode Java, la sémantique

dynamique considère par exemple les *valeurs* manipulées par les exécutions d'instructions de bytecode, alors que la *sémantique statique* nécessite seulement de considérer les *types* des données manipulées par les instructions. Une spécification de sémantique dynamique pourrait servir seulement à vérifier, voire prouver, qu'une implantation de JVM exécute correctement les instructions de bytecode. Par exemple, la sémantique dynamique de l'exécution d'une instruction iadd, est que pour deux valeurs de type int a et b au sommet de la pile, le résultat de type int empilé est $a + b$, et pas $a + b + 1$ ou une autre valeur. La sémantique statique de cette instruction est seulement que deux valeurs de type int doivent être au sommet de la pile avant l'exécution, et qu'une valeur de type int est empilée.

Pour la vérification des programmes, comme par exemple dans notre approche, il n'est pas nécessaire d'aborder la sémantique dynamique, mais seulement la syntaxe et la sémantique statique. La spécification du vérificateur de la JVM ne traite d'ailleurs explicitement que de la syntaxe et de la sémantique statique [LY99]. Dans ce chapitre, nous avons donc utilisé le terme de « sémantique » comme synonyme de « sémantique statique », et nous n'avons pas considéré la sémantique dynamique. Les caractéristiques de nos travaux, par rapport aux autres travaux de formalisation pour la vérification de bytecode Java, sont 1) la considération de la syntaxe en plus de la sémantique statique, 2) l'application à la validation pour le test des transformations. Les autres travaux ne sont pas adaptés à ces caractéristiques.

4.4 Conclusion et perspectives

Dans ce chapitre, nous avons exploré l'intersection entre le domaine de la sûreté de fonctionnement et celui de la transformation de programme, pour aborder la problématique de la validation de la correction des programmes dans leur représentation interne, pendant leur transformation. Cette problématique est nouvelle, et nous ne connaissons pas d'autres travaux qui l'abordent. Nous avons appliqué notre approche et expérimenté dans le cadre de Jabyce.

Architecture

L'approche proposée tire parti de deux travaux existants autour du modèle de composant Fractal, pour la validation de comportement de composants : ConFract et TLO. Nous proposons une architecture générale d'un *domaine de traitement d'erreur*, qui généralise et étend ces travaux. Pour l'instant, nous avons abordé et expérimenté seulement les primitives de la détection et du diagnostic des erreurs. Une première perspective de ce travail est d'implanter et d'expérimenter un composant de recouvrement d'erreur. A priori, un tel composant est facile à implanter dans le cadre de ConFract, d'autant plus que nous proposons une politique de recouvrement d'erreur simple. En ce qui concerne l'architecture de ConFract, comme détaillé dans la section 4.3.1, il faudra modifier l'architecture de ConFract pour sortir les gestionnaires de contrats (CtC) et les évaluateurs de dispositions de contrats, hors des membranes des composants Fractal, pour les concevoir comme des composants Fractal à part entière. Cela permettra de configurer par des liaisons entre composants les relations entre composants observés, composants gestionnaires de contrats, composants de recouvrement d'erreurs et évaluateurs de dispositions de contrats.

Formalismes de spécification

Nous avons abordé trois formalismes de spécification de contrats : 1) iContract, un langage de spécification de contrats sur objets, 2) ConFract/CCL-J, pour la spécification de contrats sur composants, et 3) TLO/TLA, pour la spécification de comportement en logique temporelle. Le tableau 4.1 synthétise l'évaluation de ces formalismes en fonction des quatre critères que nous avons définis (cf. la section 4.3.1).

	iContract	CCL-J	TLA
adéquation avec Fractal		√	√
adéquation avec ConFract		√	√
types de contrats (interface et/ou composition)	interface	interface et composition	composition
contrats avec état		√ (mais difficiles à spécifier)	√

TAB. 4.1 : Synthèse de l'évaluation des formalismes de spécification de contrats pour la correction des programmes transformés

Nous avons déjà généralisé l'utilisation de iContract dans Jabyce, pour la spécification de contrats de correction syntaxique des programmes transformés. iContract est le seul formalisme que nous avons utilisé pour spécifier de nombreux contrats. Cependant, parmi ces trois formalismes, seuls CCL-J et TLA permettent d'exprimer des propriétés de correction des programmes au niveau de leur sémantique, sous la forme de contrats de composition et/ou de contrats qui maintiennent un état propre. Cependant, le formalisme TLA est plus concis et plus expressif que CCL-J pour de telles spécifications. Le problème dans CCL-J est que le mécanisme de variables fantômes actuellement offert, rend difficile la spécification de contrats qui nécessitent de maintenir un état complexe dans les contrats. L'une des perspectives des travaux sur CCL-J, discutée plus en détails à la fin de la section 4.3.3, est d'introduire des variables de modèles avec des fonctions d'abstraction, comme dans JML par exemple.

Il apparaît donc nécessaire de pouvoir combiner CCL-J et TLA, basés respectivement sur des assertions exécutables, et sur des modèles formels et des propriétés en logique temporelle, pour obtenir la meilleure expressivité possible. Nous utiliserons cette combinaison en remplacement de iContract. Nous considérons que cette problématique ne concerne que l'architecture logicielle de ConFract et de TLO. Notamment, un gestionnaire de contrat ConFract (CtC) évalue les dispositions des contrats qu'il gère en délégant cette évaluation à un évaluateur spécifique à chaque formalisme de spécification. Ainsi, nous proposons comme perspective de modifier TLO pour l'intégrer dans notre architecture de domaine de traitement d'erreur, sous la forme d'un évaluateur qui utilise le *model checker* TLC pour évaluer des spécifications TLA. L'utilisation simultanée de plusieurs formalismes de spécification est originale par rapport aux autres travaux de spécification formelle du bytecode Java.

Dans TLO, les spécifications en TLA des oracles et des comportements observés sont pour l'instant écrites à la main. L'une des perspectives de TLO est de pouvoir générer automatiquement

ces spécifications, à partir des spécifications des comportements attendus, comme cela est réalisé pour d'autres formalismes existants pour la validation de comportement [RAO92, DR96].

Spécification de programmes Java compilés

Pour l'instant, seules quelques spécifications en CCL-J et TLA ont été écrites, pour valider notre approche, qui ne concernent que certaines sous-parties de la spécification de la JVM [LY99]. Une des perspectives de ce travail est donc de compléter ces spécifications, pour aboutir à un ensemble de spécifications couvrant complémentement la spécification de la JVM pour les vérifications. La plupart des autres travaux existants de spécification formelle, tels que [FM98] pour la spécification formelle de la sémantique des séquences d'instructions pour la création d'objets, ne concernent que des sous-parties de la spécification. Notre approche modulaire pour la spécification et la validation facilitera la réutilisation de telles spécifications, indépendamment les unes des autres.

L'approche proposée n'est pas limitée aux vérifications réalisées par le vérificateur de la JVM. Il est notamment envisageable de vérifier que les programmes transformés respectent certaines conventions de programmation, afin de détecter certaines erreurs liées à des dépendances contextuelles, dans les transformateurs spécifiques à ces conventions de programmation (cf. la section 4.1.2).

Consommation de ressources

Notre objectif principal est l'intégrité, c'est-à-dire la garantie que des opérations de transformation ne puissent pas représenter des programmes incorrects, quelque soit le coût de la vérification de cette intégrité pendant l'exécution des transformations. Cependant ce coût, exprimé en terme de consommation de mémoire et de temps d'exécution, peut être très élevé et un objectif secondaire est de le minimiser. Le coût de la détection et du diagnostic des erreurs doit particulièrement être minimisé, parce qu'un grand nombre d'interactions représentant des éléments de programmes doivent être observées. En revanche, un recouvrement d'erreur peut être coûteux, parce que cela se produit exceptionnellement.

En pratique, nous avons mesuré le coût en temps d'exécution de la détection d'erreur pour tous les contrats d'interfaces spécifiés avec iContract dans Jabyce, pour la transformation d'un grand programme (toutes les classes des APIs Java version 1.2). La mise en œuvre de cette détection augmente d'un rapport 10 le temps d'exécution des transformations. En revanche, le coût en mémoire est négligeable. Nous n'avons pas mesuré le coût de la détection avec ConFract/CCL-J et TLO/TLA, mais nous prédisons que ce coût est supérieur à celui de iContract. En effet, l'introduction d'intercepteurs dans les membranes des opérations de transformation composites, comme illustré dans les figures 4.15 et 4.13, rendent impossible les optimisations des liaisons effectuées par Julia décrites dans la section 3.2.6. De plus, les spécifications de propriétés de correction du niveau de la sémantique des programmes, qui peuvent être spécifiées avec CCL-J et TLA et pas avec iContract, imposent une consommation de mémoire supplémentaire pour conserver l'état des contrats (historique synthétique des séquences d'interactions), même si nous prédisons que cette consommation n'est pas très significative.

Cette consommation de ressources supplémentaires prohibe une mise en œuvre généralisée de ce traitement d'erreur. En pratique, nous allons utiliser notre approche pour la validation de

la correction des programmes dans deux cas : 1) pour le test des transformations, et 2) dans l'environnement Jivaro. Jivaro est un compilateur bytecode Java → C basé sur Jabyce et un environnement d'exécution Java, présentés dans le chapitre 9. Cet environnement n'intègre pas de processus de vérification, contrairement à ce qui est spécifié dans la JVM [LY99] (cf. la description des passes 2 et 3 du vérificateur, dans la section 4.1.2). Il est envisageable de mettre en œuvre un tel processus de vérification, en appliquant notre approche de traitement d'erreur, pendant la compilation par la chaîne d'opérations Jabyce et le traducteur qui forment Jivaro. La complétion des spécifications en CCL-J et TLA, permettra de se rapprocher d'un vérificateur de programme complet, respectant complétement les spécifications de la JVM pour les vérifications, qui serait utilisable dans Jivaro.

Généralisation de l'approche de traitement d'erreur

Dans la perspective d'un rapprochement entre ConFract et TLO, et d'une utilisation combinée des formalismes CCL-J et TLA, une autre perspective est de définir un nouveau formalisme spécifique, pour l'expression de propriétés de correction de représentation internes de programmes (ou autres). Cette démarche s'inscrit dans l'évolution du logiciel Jaidee, notre générateur de canevas logiciel de transformation décrit dans la section 3.3. Ce nouveau formalisme devra permettre d'exprimer, de manière cohérente et dans le même formalisme, un modèle conceptuel (i.e. l'ensemble des types d'éléments de programmes représentables), et des propriétés de correction sur ce modèle, comme dans le langage actuellement proposé (cf. la figure 3.12), mais en permettant en plus d'exprimer des propriétés de correction au niveau de la sémantique des programmes. Jaidee devra pouvoir générer automatiquement des spécifications CCL-J ou TLA à partir de spécifications dans ce nouveau formalisme.

Ainsi, l'approche proposée dans ce chapitre n'est pas limitée à la vérification de la correction des programmes Java transformés par des opérations de transformation Jabyce. Cette approche est facilement généralisable à tout système de transformation généré par Jaidee, comme décrit dans la section 3.3. Il est par exemple envisageable de mettre en œuvre une vérification des transformations de documents LaTeX, dans un système de transformation similaire à Jabyce. La seule différence est le modèle conceptuel, qui définit les éléments manipulés par les transformateurs ; en revanche, les contrats à vérifier sont de même nature.

Chapitre 5

Comparaison avec les systèmes existants

Dans ce chapitre nous comparons Jabyce, présenté dans le chapitre 3, à d'autres systèmes de transformation de classes Java compilées. Nous ne considérons pas les sytèmes de transformation de programmes source Java. L'objectif de ce chapitre est de montrer l'avantage qu'apportent les deux caractéristiques originales de Jabyce pour la construction de programmes de transformation complexes : 1) l'utilisation d'un modèle de composant (Fractal) pour le développement et la composition de transformateurs, et 2) la représentation des programmes par des séquences d'interactions. La comparaison des systèmes se rapporte à nos objectifs, c'est-à-dire qu'elle se base sur une évaluation du degré d'application dans chaque système de nos trois principes de conception définis au début du chapitre 3 : Généralité du Système de Transformation, Encapsulation et Composabilité des Transformations, Composition Efficace des Transformations.

5.1 Critères de comparaison

En matière de *critères de comparaison* de ces systèmes de transformation, nous utilisons les *dimensions* des systèmes de transformation de programme introduites par Eelco Visser dans [Vis01]. Cet article introduit une catégorisation des systèmes de transformation de programme, qui comprend trois dimensions : *portée* (*scope*), *modèle de représentation de programmes* (*program representation*) et *paradigme de transformation* (*transformation paradigm*). La portée concerne les utilisations possibles d'un système de transformation, c'est-à-dire l'ensemble des transformations qu'il est possible d'implanter avec un système. La dimension du modèle de représentation de programmes considère la modélisation des éléments formant un programme. Cette dimension est composée de deux sous-dimensions : 1) les modèles en arbres ou en graphes (pour représenter ou non les relations entre déclarations et utilisations, en plus des relations de composition entre p.ex. classes et méthodes), et 2) les graphes d'analyse syntaxique conservant des informations de formattage (*parse trees*) ou abstraits (*AST*). La dimension du paradigme de transformation considère essentiellement le langage offert pour exprimer des règles de transformation, comme par exemple le langage déclaratif Stratego [Vis01]. Une révision de cette catégorisation [Vis02, Vis04] introduit une autre dimension : le modèle de représentation de

fragments de programmes dans les transformations. Cette représentation peut être *abstraite* (*abstract syntax*) si ces fragments sont représentés par des abstractions du langage de transformation (p.ex. des `struct` si le langage de transformation est C), ou *concrète* (*concrete syntax*) si ces fragments sont représentés directement dans la syntaxe du langage des programmes transformés (p.ex. des constructions Java, si les programmes transformés sont en Java, même si le transformateur n'est pas écrit en Java). Les fragments sont utilisés pour spécifier dans les implantations de transformateurs des ensembles d'éléments à rechercher (*pattern matching*), ou à de nouveaux éléments à représenter.

Cependant, cette catégorisation considère seulement les systèmes à base de règles déclaratives, et pas les systèmes à base de règles procédurales, comme Jabyce. Plus généralement, de même que dans l'approche des langages spécifiques aux domaines (DSL, cf. section 2.1), cette catégorisation aborde les problèmes d'expression des transformations dans des langages de haut niveau, sans prendre en compte les aspects d'architecture et d'ingénierie logicielle des systèmes de transformation. Ces aspects sont pourtant essentiels lorsqu'on considère des configurations complexes de transformateurs, et la composition de tels logiciels avec des logiciels d'autres natures (p.ex. une base de données).

De plus, cette limite ne permet pas de prendre en compte une caractéristique originale de Jabyce : la représentation interne de programmes par des séquences d'interactions. C'est pour pouvoir prendre en compte cette caractéristique que nous séparons la dimension de la représentation de programme en deux dimensions orthogonales : la dimension du modèle conceptuel, et la dimension du mécanisme de représentation interne, comme expliqué dans les sections 3.2.3 et 3.2.4. Un modèle conceptuel définit les types d'éléments qui forment le grain d'une représentation interne d'un programme, et les relations entre les éléments. Par exemple, un modèle conceptuel d'une classe Java compilée peut comprendre entre autres les types d'élément « classe » et « méthode », et des relations de composition, par exemple entre une « classe » et une « méthode ». En ce qui concerne le mécanisme de représentation interne, seuls deux mécanismes existent pour représenter les éléments de programmes : par des objets, ou par des interactions. Ces deux dimensions sont orthogonales. Par exemple, les systèmes de transformation Serp et Jabyce ont des modèles conceptuels très proches (les mêmes types d'éléments de programmes sont définis), mais Serp représente les éléments par des objets Java, alors que Jabyce représente les éléments par des interactions (appels de méthodes). Autre exemple, les systèmes Serp et BCEL représentent tous deux les éléments par des objets Java, mais ont des modèles conceptuels différents : BCEL offre un modèle conceptuel à grain plus fin que celui de Serp, c'est-à-dire avec un plus grand nombre de types d'éléments.

Nous avons également abordé la problématique nouvelle de la *vérification des transformations*, c'est-à-dire les garanties qu'offre un système de transformation sur la correction des programmes représentés, comme décrit en détails dans le chapitre 4 pour Jabyce.

Nous proposons donc une nouvelle catégorisation des systèmes de transformation de programme, qui étend celle de E. Visser, et qui comprend sept dimensions :

- *portée* (la dimension *scope* de E. Visser) : utilisations possibles du système, transformations implantables, comme par exemple la compilation et l'ingénierie inverse.
- *mécanisme de représentation interne* : les abstractions du langage de transformation (p.ex. Java) utilisées pour la représentation interne d'éléments de programmes. Nous avons identifié deux catégories : par des graphes d'objets, ou par des interactions (c'est-à-dire des appels de méthodes) comme dans Jabyce.

- *modèle conceptuel* (la dimension *program representation* de E. Visser) : les types des éléments manipulables dans la représentation interne des programmes. Cette dimension concerne le choix (le « découpage ») des types des éléments de programmes représentables (p.ex. « classe », « méthode », « attribut », etc.), et des relations de composition et de dépendance entre ces types d'éléments.
- *modèle de représentation de fragments de programmes* : correspond à la distinction entre *concrete syntax* et *abstract syntax* de E. Visser, pour la représentation de fragments de programmes dans les implantations de transformateurs.
- *paradigme de transformation* (la dimension *transformation paradigm* de E. Visser) : le modèle d'implantation / de définition des transformations. Cette dimensions concerne essentiellement le langage de transformation, par exemple un langage déclaratif comme Stratego, ou un langage impératif comme C ou Java. Nous considérons que cette dimension concerne également les aspects architecturaux, comme les modèles de composants, etc.
- *stratégie de transformation* (introduit par E. Visser) : le moyen de définition de compositions de transformations.
- *vérification des transformations* : les mécanismes permettant de garantir un comportement correct des transformations.

Dans le cadre de notre démarche générale (cf. la section 2.4), nous avons étudié les dimensions (boutons) du domaine de la transformation de programme données dans l'état de l'art, et nous avons introduit de nouvelles dimensions. Ces nouvelles dimensions nous permettent de décrire les particularités de nos travaux comme des variations dans ce domaine, et de décrire les intersections avec d'autres domaines, comme décrit par exemple dans le chapitre 4 pour le domaine de la sûreté de fonctionnement.

La suite de ce chapitre suit un plan thématique, avec une section comparant les systèmes existants avec Jabyce selon chacune des dimensions que nous avons définies. Dans chaque dimension, nous avons introduit plusieurs critères pour évaluer précisément le degré d'application dans chaque système de nos principes de conception 4, 5 et 6 définis dans le chapitre 3. La dimension de portée des systèmes est la plus importante, car elle nous permet de sélectionner les systèmes qui sont évalués dans les sections suivantes. Les dimensions du mécanisme de représentation interne de programmes et de la vérification des transformations, ne sont abordées brièvement dans ce chapitre que pour comparer Jabyce avec les autres systèmes, et sont introduits plus en détails dans les chapitres 3 et 4. La section 5.9 donne une synthèse des comparaisons.

5.2 Portée et sélection des systèmes comparés

Les différentes formes de la transformation de programme définies dans [Vis01, Vis] sont illustrées dans la figure 5.1. Il existe deux grandes catégories d'utilisations de la transformation de programme : la *réexpression* (*rephrasing*) et la *traduction* (*translation*).

Les systèmes de *réexpression* de programme transforment des programmes en d'autres programmes dans le même modèle conceptuel. Plus précisément, cette catégorie de transformations comprend les transformations de normalisation, d'optimisation, de remaniement, et de rénovation. La *normalisation* (*normalization*) est la réduction d'un programme à un programme dans un sous-langage, pour diminuer sa complexité syntaxique. Notamment, la *purification* (*desugaring*) consiste en le remplacement de constructions syntaxiques complexes en

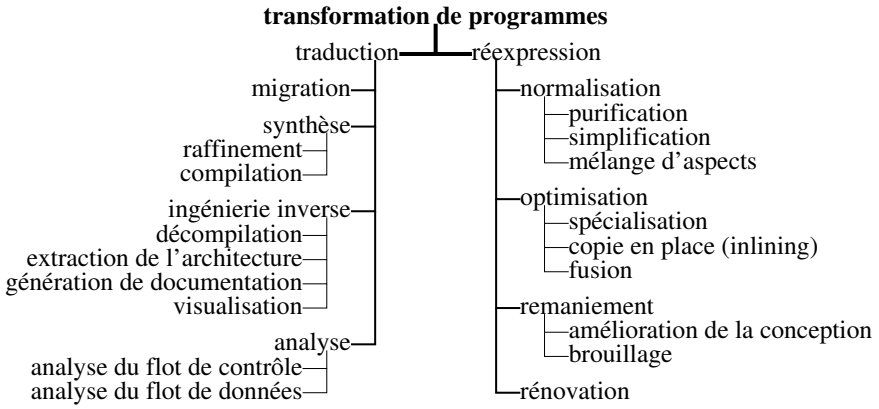


FIG. 5.1 : Formes de transformations de programmes

des constructions plus fondamentales, la *simplification* est une transformation plus générale qui consiste en la réduction d'un programme à une forme normale. Le mélange d'aspects (AOP, cf. la section 2.2), est considéré également comme une forme de normalisation. Une *optimisation* (*optimization*) est une réexpression d'un programme pour améliorer ses performances notamment en termes de temps d'éxecution et de consommation de mémoire. Le *remaniement* (*refactoring*) est une transformation de la structure d'un programme, pour le rendre plus facile à comprendre et à modifier, sans changer sa sémantique. La *rénovation* (*renovation*), ou *ré-ingénierie*, est un processus comprenant une ingénierie inverse (*reverse-engineering*), une restructuration du programme dans une forme d'un niveau d'abstraction supérieur, et une ingénierie en avant (*forward engineering*) pour reproduire un programme dans la forme originale. Contrairement à un remaniement, une rénovation considère un programme dans une forme à un niveau d'abstraction supérieur à sa forme originale. On peut également citer le *brouillage* (*obfuscation*), qui a pour objectif d'empêcher la compréhension d'un programme par un humain, typiquement en remplaçant tous les identifiants par des chaînes aléatoires.

Les systèmes de *traduction* de programme transforment des programmes dans un modèle conceptuel source, en des programmes dans d'autres modèles conceptuels. Une *migration de programme* (*program migration*) adapte un programme à une autre version de langage de programmation. Une *synthèse de programme* (*program synthesis*) est la dérivation d'un programme à partir d'une spécification. L'*ingénierie inverse* (*reverse engineering*) d'un programme est son examination dans une représentation à un haut niveau d'abstraction, pour identifier les parties du programme et leur relations. Par exemple, l'ingénierie inverse d'un programme Java peut consister en la génération automatique d'un modèle de classes UML à partir du code source. Une *analyse de programme* (*program analysis*) est l'inspection d'un programme, p.ex. du flot de contrôle ou du flot des données, pour inférer des propriétés. L'analyse de programme est une classe de transformations très courantes, souvent utilisées pour implanter un grand nombre d'autres transformations.

Nous avons motivé dans l'introduction du chapitre 3, que nous considérons principalement des systèmes de réexpression de programmes Java compilées dans le cadre de notre problématique de la mise en place d'un lien entre code fonctionnel et code technique. Ainsi, nous n'étudions pas les systèmes de traduction dans de chapitre. En revanche, le chapitre 9 présente Jivaro, un compilateur de code compilé Java vers C basé sur Jabyce. De même, nous ne considérons pas

les systèmes de remaniement ou de rénovation, car ils nécessitent souvent une interaction avec des utilisateurs humains, alors que nous souhaitons automatiser le plus possible les transformations. Les systèmes que nous considérons doivent cependant permettre d'implanter toute autre forme de transformation de réexpression, en application de notre principe 4 de Généralité du Système de Transformation. La suite de cette section identifie les systèmes de réexpression de programmes Java compilés, et sélectionne les systèmes qui respectent ces critères.

5.2.1 Systèmes de transformation généralistes

COMPOST [LH00] est un canevas logiciel (framework) et un programme interactif, développés en Java, pour la maintenance de grands programmes Java. Ce projet académique est développé depuis 1998 par des membres de l'Université de Karlsruhe. Ce système est généraliste, et permet d'implanter n'importe quelles transformations de réexpression de programmes Java, appelées méta-programmes. La portée de ce système est essentiellement, mais pas exclusivement, le remaniement de programmes (*refactoring*). Une bibliothèque de transformateurs directement utilisables est fournie avec le système. Le cœur de ce système de transformation, appelé RECODER, est publié en logiciel libre depuis 2001. C'est l'un des systèmes de transformation de programmes Java les plus complets. Cependant, nous n'étudions pas ce système en détails dans la suite, car il transforme exclusivement des fichiers source Java.

ASM [BLC02] est développé dans le laboratoire DTL/ASR de France Telecom R&D, et publié sous la forme de logiciel libre depuis 2002 dans le cadre du consortium Objectweb [Objb]. C'est un canevas logiciel développé en Java, pour l'implantation efficace de transformations de réexpression de classes Java compilées. ASM représente les programmes comme des séquences d'interactions, comme dans Jabyce, mais sans aborder complètement la problématique de la composition des transformations. JOIE (*Java Object Instrumentation Environment*) [CCK98], JMangler [KCA01] et Javassist [Chi00] sont des systèmes de transformation de classes Java compilées, qui offrent des abstractions pour l'implantation de transformations. Ces trois systèmes sont similaires, et sont issus du milieux académique : JOIE est issu de Duke University aux USA, JMangler est issu de l'Université de Bonn, et Javassist est issu du Tokyo Institute of Technology. Il est à noter que JMangler est limité aux transformations qui conservent la *compatibilité binaire* [DWE98] des classes transformées, c'est-à-dire aux transformations de classes qui ne nécessitent pas également la transformation des classes référençant les classes transformées. Serp et JOIE sont limités à la transformation de classes Java compilées existantes, et ne permettent pas de créer de nouvelles classes. Les premiers travaux universitaires sur la transformation de code compilé Java étaient BCA (*Binary Component Adaptation*) [KH98] développé à l'Université de Californie, et BIT (*Bytecode Instrumentation Toolkit*) [LZ97] développé à l'Université du Colorado. Cependant, ces projets ne sont plus maintenus. JOIE, JMangler et Javassist offrent des fonctionnalités similaires. Nous n'étudions donc pas BCA et BIT dans la suite de ce chapitre.

jclasslib [KK], Serp [Whi] et BCEL [DMM+] (*ByteCode Engineering Library*) sont des bibliothèques de classes Java exclusivement destinées à représenter les classes Java compilées sous forme de graphes d'objets Java. Contrairement aux autres systèmes cités ci-dessus, ces bibliothèques n'offrent pas d'abstractions pour l'implantation de transformations, et ne sont donc pas par définition des systèmes de transformation. Mais elles sont utilisables pour implanter un système de transformation. Notamment, BCEL est utilisé pour implanter JMangler. Nous

n'étudions BCEL et Serp que pour les modèles conceptuels et les mécanismes de représentation interne de programmes qu'ils offrent, dans la section 5.4. jclasslib offre un modèle conceptuel de classes Java compilées trop basique pour être utilisable en pratique dans des transformations complexes. En effet, trop peu de détails du format sérialisé des classes compilées sont abstraits au programmeur de transformation, ce qui implique une duplication inutile du traitement de ces détails dans les implantations de transformateurs. Ceci est en contradiction avec notre principe 6 de Composition Efficace des Transformations. Ce principe implique d'éviter de dupliquer les traitements redondants, notamment ceux qui ont trait aux détails du format des classes Java compilées, afin d'améliorer l'efficacité globale d'une chaîne de transformateurs. Nous n'allons donc pas étudier jclasslib dans la suite de ce chapitre.

5.2.2 Logiciels de transformation spécialisés

Outre les systèmes de transformation généralistes cités ci-dessus, il existe de nombreux logiciels de transformation de programmes Java compilés, spécialisés dans certaines transformations, et qui montrent l'intérêt de transformer des programmes Java compilés.

Java Class File Editor [MGM] est un éditeur graphique de classes compilées. Dava est un décompilateur de code Java compilées vers du code source Java [MH02a]. Cglib [BNRB] est une librairie, basée sur ASM, de génération de classes Java compilées, utile pour créer des objets mandataires (*proxy*) pendant l'exécution.

Nanning [LLLT] permet de mélanger des aspects (AOP) lors de la génération de classes d'objets mandataires, et est basé sur Cglib. JMunger [BHvdMW] est un système de programmation par aspects (AOP), basé sur JMangler, et qui permet d'insérer, au chargement des classes Java compilées, du code exprimé sous forme de langage source Java. PCESjava [Cytb] permet d'instrumenter des classes compilées en insérant du code spécifié par des classes Java avec des conventions de nommage spécifiques.

jContractor [AK02, KHB98, AKWN] est un outil de conception par contrats sur des objets, qui se base sur des transformations de classes Java compilées pour introduire dans les implantations de méthodes des vérifications d'assertions exécutables. jContractor est basé sur la librairie BCEL.

Jarg [Ohu] est un optimiseur basique du code de classes. GenJar [Sto] et jclassinfo [Pan] permettent d'analyser les dépendances entre classes. jclassinfo est développé en C. Jode [FBH+] est un optimiseur contrôlable avec un langage de script. JAX [TLSS99, LMS+] et Jamit [Gro] sont des optimiseurs qui se basent sur une optimisation des modificateurs d'accès des classes, des méthodes et attributs suivant leurs utilisations effectives, et sur la suppression des classes et membres inutilisés. BLOAT [Nys98, NW] est un optimiseur des implantations de méthodes. jDFA [Moh02] est un système pour implanter des analyses statiques de flot de données de bytecode Java. JoGa [Hou] est un optimiseur qui peut mélanger des classes et remplacer les appels de méthodes par l'implantation des méthodes appelées (*inlining*). JavaGO [Kni] est un optimiseur qui dévirtualise les appels de méthodes, et qui peut remplacer les appels de méthodes par leurs implantations. Clazzer [Cyta] et JCMP [Zha] sont des optimiseurs d'implantations de méthodes, et des brouilleurs qui réduisent la longueur des identificateurs. Soot [VRHS+99] est un système de transformation de classes compilées général et extensible pour l'implantation d'optimisations de programmes en Java, basé sur trois modèles de représentations internes qui simplifient les optimisations à différents niveaux d'abstraction. Soot est un

logiciel issu du milieux universitaire (McGill University, Montreal), très complet et puissant. ProGuard [Laf] est un logiciel de brouillage et d'optimisation paramétrable. JavaGuard [Hei] est un logiciel de brouillage.

[SSY00] décrit un système qui transforme les programmes Java compilés pour permettre une migration transparente forte des activités Java. JavaGO [YL] est un système similaire. Barat [BS98, BSD] est un système permettant de réaliser des analyses statiques sur des représentations, en graphes d'objets non mutables, du code source ou compilé de programmes Java.

La bibliothèque Tea Trove Class File [OJY+] est similaire à BCEL, mais a pour limite de ne pas permettre d'analyser le code de méthodes de classes compilées existantes, c'est-à-dire qu'elle n'offre pas de désérialisation des implantations de méthodes.

JSpec [SLCM98, Proa] est un évaluateur partiel de programmes Java sous la forme de code source ou de classes Java compilées, qui se base sur un traducteur Java → C, sur l'évaluateur partiel de programmes C Tempo [CHL+98, Prob], et sur un traducteur C → Java.

Ces logiciels montrent l'intérêt de la transformation de programmes Java compilés, et les variations possibles dans ce domaine. Cependant, pour la plupart, ces logiciels sont en pratique difficilement réutilisables et composables, car ils ne s'appuient pas sur un système de transformation général et commun. Cette limitation importante motive la conception de systèmes de transformations qui facilitent la composition de transformateurs, comme Jabyce, en application de notre principe 5 d'Encapsulation et de Composabilité des Transformations.

5.3 Mécanisme de représentation interne (objets *vs.* interactions)

Cette dimension concerne le choix d'un modèle pour la représentation interne des éléments des programmes transformés, dans les implantations de transformateurs. Les systèmes JOIE, BCEL, Javassist et Serp offrent classiquement une représentation interne des programmes par des *graphes d'objets* Java, dont les classes et interfaces sont définies dans ces systèmes. Seuls ASM et Jabyce offrent une représentation interne par des *séquences d'interactions* entre transformations. Cette approche nouvelle est introduite dans la section 3.2.4. A priori, nous considérons que ce sont les deux seules formes de représentation envisageables pour un élément de programme : soit par quelque chose *« qui est »*, telle qu'un objet, soit par quelque chose *« qui se produit »*, telle qu'une interaction (voir la définition d'une *action* dans RM-ODP, dans l'annexe C).

La représentation par séquences d'interactions est plus efficace qu'une représentation en graphes d'objets, en termes de consommation de mémoire et de temps d'exécution, comme montré dans le chapitre 6 des mesures de temps d'exécution. Cependant, elle rend plus difficile l'implantation de transformations nécessitant de raisonner sur les relations entre les éléments d'un programme, par exemple les transformations qui réalisent des analyses statiques du flot de contrôle. Cette différence de complexité tient essentiellement dans la manière dont sont représentées les relations entre éléments de programmes, avec ces deux mécanismes. Cette section met donc l'accent sur *une comparaison des représentations internes des relations entre éléments*, avec Serp, ASM et Jabyce. Serp est représentatif de tous les systèmes qui utilisent une représentation interne des programmes par graphes d'objets.

Nous introduisons deux critères pour la comparaison dans cette section : 1) *l'homogénéité* des représentations de relations entre éléments, et 2) *la facilité d'analyse* de ces relations, c'est-à-dire la facilité d'implantation de transformateurs raisonnant sur ces relations. Nous avons identifié deux autres critères concernant les relations entre éléments, mais qui sont dans d'autres dimensions : 3) *la complétude* (le nombre) des relations explicitement identifiées dans le modèle conceptuel (cf. la section 5.4), et 4) *l'encapsulation* systématique des transformations d'éléments de programmes sémantiquement dépendants (cf. la section 5.5). Nous n'abordons pas ces deux derniers critères dans cette section. Un autre critère de comparaison est la facilité de modification, de création et de suppression des représentations d'éléments de programmes et des relations entre éléments.

Considérons par exemple la classe Java UneClasse dont le code source est donné dans le programme 5.1. Le code compilé de cette classe est partiellement donné dans le programme 5.2, tel qu'il est donné par un désassembleur[1]. Nous nous intéressons ici à la manière dont sont représentées certaines relations dans cette classe Java, respectivement avec Serp, ASM et Jabyce :

- les relations entre éléments d'une même classe (*intra-classes*) :
  - les relations de *composition*, par exemple entre la classe UneClasse et la méthode uneMethode() ;
  - les relations de *dépendance sémantique* entre éléments d'une même classe, par exemple entre l'instruction de branchement if_icmpge et l'instruction return ;
- les relations de *dépendance sémantique* entre éléments de différentes classes (*inter-classes*), comme les relations de classes internes (*inner classes*) et d'héritage, ou par exemple entre les instructions d'accès à l'attribut unAttribut (getfield et putfield) et l'attribut lui-même.

```
public class UneClasse {
  private int unAttribut;
  public void uneMethode() {
    for (int i = 1; i < 3; i++) {
      unAttribut += i;
    }
  }
}
```

PROG. 5.1 : Classe UneClasse

Représentation des relations intra-classes

Avec Serp La figure 5.2 donne le diagramme d'objets UML d'une partie de la représentation interne de la classe compilée UneClasse, avec Serp. Chaque élément est représenté par un objet Java. Par exemple, la classe est représentée par un objet BCClass et un objet ObjectState. Comme décrit dans la section 5.4, toutes les valeurs constantes sont identifiées par leur index dans la table des constantes. Par exemple, un objet BCField contient l'index du nom de l'attribut dans l'attribut _nameIndex. Dans une instruction getfield (objet GetFieldInstruction), l'index est celui d'une constante complexe, qui comprend le nom de l'attribut accédé, le nom

[1] Les étiquettes à gauche des instructions sont les décalages en octets depuis le début de la méthode.

```
public class UneClasse extends java.lang.Object{
private int unAttribut;
public void uneMethode();
  Code:
   0:   iconst_1
   1:   istore_1
   2:   iload_1
   3:   iconst_3
   4:   if_icmpge       23
   7:   aload_0
   8:   dup
   9:   getfield        #2; //Field unAttribut:I
   12:  iload_1
   13:  iadd
   14:  putfield        #2; //Field unAttribut:I
   17:  iinc    1, 1
   20:  goto    2
   23:  return
}
```

Prog. 5.2 : Code compilé de la classe UneClasse

de la classe qui le définit, et la signature de l'attribut. Des méthodes utilitaires sont offertes pour lire et modifier ces constantes. Par exemple, BCField offre une méthode getName(), qui retourne la chaîne contenant le nom stockée dans la table. Le descripteur (*descriptor*) d'un attribut ou d'une méthode est une chaîne qui encode le type d'un attribut, ou la signature d'un méthode.

Les relations de composition sont concrétisées par des *liaisons entre les objets*, implantées par les attributs _fields, _methods, _opcodes et _owner des objets. Les objets et les liaisons forment un arbre. L'attribut _opcodes d'un objet BCMethod est une liste des objets Instruction représentant les instructions de la méthode. La relation de dépendance entre l'instruction de saut (if_icmpge) et l'instruction à laquelle continue l'exécution (return) est également représentée par une liaison, entre les objets IfInstruction et ReturnInstruction correspondants. Cette liaison est utilisée pour calculer automatiquement le décalage du saut, en octets, lors de la sérialisation de cette représentation interne du code.

Avec ASM Dans ASM, chaque élément d'un programme est représenté par un appel de méthode Java. Cette approche est décrite de manière générale dans la section 3.2.4. Le programme 5.3 donne un code Java imaginaire qui initie la même séquence d'appels de méthodes qui représentent la classe UneClasse donnée ci-dessus, que celle qui serait initiée par un objet désérialiseur ASM (classe ClassReader). Pour la représentation de chaque classe compilée, il faut créer un objet ClassVisitor. Un appel à la méthode visit(...) permet de donner le nom de la classe, celui de sa superclasse, etc. Chaque appel à une méthode visit*X*(...) permet de représenter un élément de programme. Par exemple, un appel à visitField(...) permet de représenter un attribut dans la classe. Un appel à visitMethod(...) permet de représenter une méthode. visitMethod(...) retourne un objet CodeVisitor, qui permet de représenter l'implantation de la méthode, en offrant des méthodes visit*X*(...), notamment pour les instructions. Par exemple,

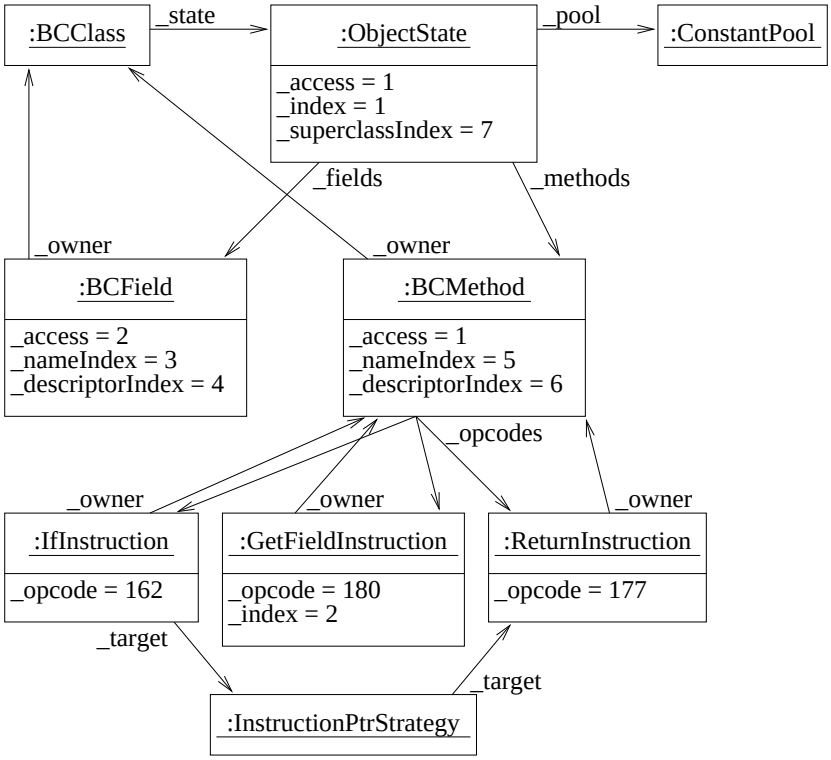


FIG. 5.2 : Diagrammes d'objets UML partiel de la représentation interne de la classe compilée UneClasse avec Serp

la méthode visitFieldInsn(...) permet de représenter des instructions getfield, putfield, getstatic et putstatic. Les *opcodes* sont donnés explicitement dans les représentations d'instructions. L'interface Constants de ASM définit des constantes entières contenant les *opcodes* (identifiants d'instructions), par exemple la constante Constants.IF_ICMPGE contenant l'opcode pour les instructions if_icmpge.

```
ClassVisitor classVisitor = ...;
classVisitor.visit(1, "UneClasse", "java/lang/Object",
    null, null);
...
classVisitor.visitField(2, "unAttribut", "I", null, null);
...
CodeVisitor codeVisitor = classVisitor.visitMethod(
    1, "uneMethode", "V()", null, null);
...
Label jumpLabel = new Label();
codeVisitor.visitJumpInsn(Constants.IF_ICMPGE, jumpLabel);
...
codeVisitor.visitFieldInsn(Constants.GETFIELD, "UneClasse",
    "unAttribut", "I");
...
codeVisitor.visitLabel(jumpLabel); // cible du saut
codeVisitor.visitInsn(Constants.RETURN);
...
codeVisitor.visitMaxs(...);
...
classVisitor.visitEnd();
```

PROG. 5.3 : Code pour la génération de la classe UneClasse avec ASM

La relation de composition entre la classe UneClasse et l'attribut unAttribut est représentée lors de l'appel à visitField(...) qui représente l'attribut : la cible de cet appel est l'objet ClassVisitor qui représente la classe qui contient l'attribut. Il en est de même pour la méthode uneMethode, lors de l'appel à visitMethod(...) sur l'objet ClassVisitor. Le même mécanisme est également appliqué dans ASM pour les relations de composition entre les méthodes et les instructions. Les appels à visit*X*Insn(...) sont effectués sur l'objet CodeVisitor qui représente la méthode qui contient les instructions représentées par ces appels.

La relation de dépendance entre l'instruction if_icmpge et l'instruction return, où s'effectue le saut, est représentée dans ASM en introduisant un objet « étiquette » de type Label juste avant l'instruction return, par un appel à visitLabel(...). Ce même objet est passé en paramètre lors de la représentation de l'instruction if_icmpge par un appel à visitJumpInsn(...). La relation entre l'instruction if_icmpge et l'instruction return est identifiée en fonction de l'identité de cet objet Label : si l'étiquette est la même (comparaison avec == en Java) dans l'instruction if_icmpge et avant l'instruction return, alors il existe une relation de dépendance / de branchement entre ces instructions.

Avec Jabyce Dans Jabyce, de même que dans ASM, chaque élément de programme est représenté par une interaction, comme décrit dans la section 3.2.5. Le programme 5.4 donne le

code Java, dans l'implantation d'un composant Fractal, qui initie la même séquence d'appels de méthodes vers un composant opération Jabyce, pour représenter la classe UneClasse donnée ci-dessus, que celle qui serait initiée par un composant désérialiseur.

```
ClassBuildingContext classContext =
  classFactory.createClass(
    new ObjectType("UneClasse", false),
    new ObjectType("java.lang.Object", false),
    null, true /*ispublic*/, ...);
...
FieldBuildingContext fieldContext =
  fieldFactory.createField(classContext, "unAttribut",
    PrimitiveType.INT, false, true /*isprivate*/, ...);
fieldFactory.finishFieldBuilding(fieldContext);
...
MethodBuildingContext methodContext =
  methodFactory.createMethod(classContext, "uneMethode",
    VoidType.VOID, true /*ispublic*/, ...);
...
InstructionLabel jumpLabel = new BasicInstructionLabel();
integerConditionalBranchInstructionFactory
  .createIntegerConditionalBranchInstruction(methodContext,
    IntegerConditionalBranchInstructions.GE, jumpLabel);
...
fieldAccessInstructionFactory.createFieldAccessInstruction(
  methodContext, new ObjectType("UneClasse", false),
  "unAttribut", false /*fieldIsStatic*/, PrimitiveType.INT,
  true /*loadOrStore*/);
...
instructionLabelFactory.createInstructionLabel(
  methodContext, jumpLabel);
returnInstructionFactory.createReturnInstruction(
  methodContext);
...
methodFactory.finishMethodBuilding(methodContext);
...
classFactory.finishClassBuilding(classContext);
```

Prog. 5.4 : Code pour la génération de la classe UneClasse avec Jabyce

Cette séquence d'interactions est très similaire à la séquence d'appels pour représenter la même classe avec ASM, donnée dans le programme 5.3. Les relations de dépendance sont représentées de la même manière que dans ASM. Par exemple, dans Jabyce les relations de saut entre instructions sont identifiées par l'égalité (selon l'opérateur ==) des objets InstructionLabel passés en paramètres, similaires aux objets Label de ASM.

Une différence entre ASM et Jabyce tient dans la manière de représenter les relations de composition. Dans Jabyce, toutes les interactions représentant des éléments de programme sont effectuées sur des interfaces d'un composant opération ou sérialiseur (référencées dans l'exemple par des variables classFactory, fieldFactory, etc.). Chaque interface est utilisée pour les appels représentant tous les éléments d'un type donné. Par exemple, l'interface de type

MethodCallInstructionFactory d'un composant opération est utilisée comme « cible » pour tous les appels représentant des instructions d'appels de méthodes. Les relations de composition sont identifiées par l'égalité des objets *X*BuildingContext qui identifient les représentations de classes, méthodes et attributs, et qui sont passés en paramètres pour la représentation des éléments dans ces éléments composites. Ce mécanisme est décrit plus en détails dans la section 3.2.5. Dans Jabyce, ce même mécanisme de l'égalité des paramètres est utilisé pour représenter à la fois les relations de composition et les relations de dépendance entre éléments.

Au contraire, dans ASM, les relations de composition sont déterminées par la « cible » des appels représentant des éléments de programmes : la « cible » identifie l'élément composite dans lequel est représenté chaque élément. Ainsi, la relation de composition entre l'attribut unAttribut et la classe UneClasse est concrétisée par le fait que la cible de l'appel à visitField(...) représentant l'attribut unAttribut est effectué avec pour cible l'objet ClassVisitor représentant la classe UneClasse. Ce mécanisme est différent de celui utilisé dans ASM et Jabyce pour réprésenter les relations de dépendance. Les relations ne sont donc pas représentées de manière homogène dans ASM.

Relations inter-classes

Les relations qui peuvent exister entre éléments de classes différentes sont des relations de dépendance entre les *déclarations* de classes, de méthodes et d'attributs, et les *utilisations*, c'est-à-dire les signatures de méthodes, les instructions de création d'objets, etc. Tous les systèmes que nous considérons permettent d'effectuer des transformations des classes Java compilées pendant leur chargement par une JVM. Les classes étant chargées une par une par la JVM, aucun système n'offre de moyen de raisonner sur un programme entier composé d'un ensemble de classes. Aucun système considéré ne permet donc de raisonner sur des relations explicites entre des représentations d'éléments qui sont potentiellement dans des classes différentes, comme les déclarations de méthodes et les appels de méthodes.

Ainsi, dans Serp, la relation de dépendance entre l'instruction d'accès à l'attribut unAttribut représentée par l'objet GetFieldInstruction et l'attribut lui-même représenté par l'objet BCField n'est pas explicitement représentée comme une liaison entre ces objets. Cette relation doit être *calculée* dans les transformateurs, en comparant les noms d'attributs, les noms des classes qui les définissent, et les signatures stockés dans l'objet GetFieldInstruction et dans l'objet BCField. Dans ASM et Jabyce, cette relation doit être calculée de la même manière, en comparant les chaînes contenant les noms d'attributs et de classes et les descripteurs, passées en paramètres dans les appels de méthodes.

Représentations modifiables

Tous les systèmes que nous considérons offrent généralement le moyen de transformer complétement, de créer et de supprimer des représentations d'éléments de programmes. Cependant, une limitation forte de JOIE est de permettre presque exclusivement de transformer des représentations d'éléments de programmes existants. En effet, la plupart des constructeurs des classes ne sont pas publics. Par exemple, il n'est pas possible de créer un objet ClassInfo qui représente une classe, sans désérialiser une classe existante. De plus, JOIE limite fortement les

modifications possibles des représentations d'éléments. Par exemple, il est impossible de modifier une représentation de méthode privée (objet Method) pour la rendre publique, ou encore de transformer une méthode abstraite en méthode concrète en lui ajoutant une implantation. Ces limitations sont en contradiction avec notre principe 4 de Généralité du Système de Transformation. Dans tous les autres systèmes, les représentations d'éléments de programmes sont complétement modifiables, et peuvent être librement créées et supprimées.

Synthèse

Les systèmes de transformations, comme JOIE, BCEL, Javassist et Serp, représentent classiquement les éléments de programmes par des objets Java. Les relations de composition et de dépendance sémantique entre les éléments sont représentées par des liaisons entre objets, pour former des graphes. Une approche nouvelle est utilisée dans ASM et Jabyce : une représentation de chaque élément par une interaction (un appel de méthode). Dans ASM et Jabyce, les relations entre éléments sont typiquement représentées par des objets identifiants les relations, passés en paramètres des appels représentant les éléments. La forme de représentation choisie dans JOIE, BCEL, Javassist et Serp, par des liaisons entre objets, est celle qui facilite le plus l'implantation d'analyses des relations, car il est facile de naviguer dans ces liaisons avec un algorithme itératif ou récursif.

Seul ASM n'offre pas une représentation homogène des relations de composition et de dépendance entre éléments de programmes. Cette caractéristique rend plus difficile une analyse homogène des relations entre éléments, notamment pour la vérification de la correction des programmes (cf. le chapitre 4). Tous les autres systèmes offrent une seule forme de représentation. Une limitation forte spécifique à JOIE est d'offrir une représentation des éléments par des objets peu modifiables.

5.4 Modèle conceptuel

La dimension du *modèle conceptuel* concerne le choix des types des éléments formant la représentation interne d'un programme transformé, et des relations entre ces types d'éléments, comme expliqué dans les sections 3.2.3 et 3.3. La section précédente, qui présente la dimension du mécanisme de représentation interne, montre que ces éléments sont représentés soit par des objets Java, soit par des interactions, et donne un aperçu des modèles conceptuels de Serp, ASM et Jabyce. Par exemple, Serp offre des classes (BCClass, BCField, etc.) pour représenter les éléments des classes Java. Ces modèles conceptuels sont détaillés dans cette section.

Dans un système qui représente les éléments par des objets Java, comme Serp, JOIE, BCEL et Javassist, le modèle conceptuel se traduit par la bibliothèque des classes Java offertes par ce système qui implantent ces objets et leurs relations. Pour ASM et Jabyce, qui représentent les éléments par des interactions, le modèle conceptuel se traduit directement par l'ensemble des méthodes offertes pour représenter les éléments, et les signatures de ces méthodes. Il est à noter que les dimensions du mécanisme de représentation interne et du modèle conceptuel sont *orthogonales*, c'est-à-dire que ce sont des « boutons » que l'on peut tourner indépendamment. Ainsi, comme montré dans cette section, les modèles conceptuels de Serp et de Jabyce sont proches, bien que ces deux systèmes représentent les éléments de programmes respectivement par des objets et par interactions.

D'un point de vue général, un modèle conceptuel est un graphe, dont les nœuds sont des types d'éléments, et les arcs des relations de composition et éventuellement de dépendance sémantique (relations déclaration - utilisation) [Vis01]. Nous avons identifié les aspects suivants à prendre en compte lors de la définition d'un modèle conceptuel : 1) la prise en compte ou pas d'éléments d'analyse syntaxique, 2) une modélisation sous forme d'arbre ou de graphe connexe quelconque, et 3) la granularité des types d'éléments dans le modèle. Ces différents aspects sont abordés dans cette section.

Conservation d'information d'analyse syntaxique

D'après [Vis01] (dimension *program representation*), un modèle conceptuel peut être soit 1) un graphe (*parse tree*) dont les nœuds modélisent des éléments conservant des informations d'analyse syntaxique (comme la mise en forme, les espaces, l'indentation, les lignes vides), soit 2) un graphe abstrait (*abstract syntax tree - AST*) qui ne conserve aucune information de formattage. La conservation d'éléments syntaxiques dans la représentation d'un programme n'est utile que pour des programmes manipulés par des humains sous la forme de fichiers source, pour pouvoir produire des programmes le plus proche possible des programmes source après transformation, en termes d'indentation, etc. Les fichiers de classes Java compilées n'étant jamais modifiés directement par un humain, tous les systèmes de transformation de classes Java compilées représentent les programmes comme des graphes abstraits, sans éléments syntaxiques. Par exemple, l'ordre de définition des méthodes est sans importance, et une transformation peut inverser l'ordre des définitions de deux méthodes dans une classe. Parmi les systèmes considérés, seul COMPOST offre des graphes contenant des informations syntaxiques, car il manipule des fichiers source, et est surtout utilisé pour les transformations de remaniement de programme (*refactoring*).

Arbre ou graphe plus complexe

Un modèle conceptuel peut être un arbre ou un graphe plus complexe. Les arcs dans un arbre représentent les relations de composition entre les éléments, par exemple la relation de composition entre un nœud « classe » et un nœud « méthode ». L'avantage de représenter les programmes en graphes quelconques au lieu d'arbres, est de pouvoir en plus représenter des liens entre déclarations et utilisations [Vis01]. Ainsi, tous les systèmes étudiés utilisent un modèle des programmes en graphes, définissant non seulement des arcs de relations de composition formant un arbre, mais aussi des arcs de dépendance sémantique entre éléments. Par exemple, dans BCEL et Serp le modèle conceptuel des classes Java compilées est un graphe, parce qu'une instruction de saut dans une méthode peut référence directement l'instruction à laquelle s'effectue le saut (voir par exemple la figure 5.2). Il est à noter que ASM et Jabyce ont un modèle en graphe, comme les autres systèmes de transformation, mais ces graphes sont « encodés » pour pouvoir être représentés par des séquences d'interactions, comme décrit dans la section 5.3.

Granularité et niveau d'abstraction du modèle

Le critère principal de comparaison dans cette dimension est le choix des *granules de transformation*, c'est-à-dire des types d'éléments formant le modèle conceptuel de programme, et

des relations entre ces types d'éléments.

Traditionnellement, dans les systèmes de transformation de code source, le modèle conceptuel correspond directement au format sérialisé des programmes, c'est-à-dire à la syntaxe / la grammaire du langage des programmes transformés. En revanche, le code compilé Java a un format sérialisé trop complexe pour être utilisé directement comme modèle conceptuel de la représentation interne des programmes. Notamment, comme décrit dans l'annexe A, chaque classe compilée a une table de constantes, et les utilisations de constantes sont concrétisées par des index dans cette table. Il est nécessaire d'abstraire de tels détails, donc d'offrir un modèle conceptuel de plus haut niveau que le format sérialisé. Ainsi, tous les systèmes que nous considérons font référence à la même spécification de forme sérialisée de programme Java compilé (la spécification de la JVM [LY99]), mais tous ces systèmes ont des modèles conceptuels à un niveau d'abstraction supérieur, pour faciliter l'implantation de transformateurs. Les modèles conceptuels offerts par ces systèmes sont différents. En effet, comme expliqué dans la section 3.2.3, le choix d'un modèle conceptuel, et sa granularité, correspondent à des *objectifs* propres à chaque système de transformation. Un critère pour comparer ces modèles conceptuels est donc *le niveau de détails* du modèle. Ainsi, dans la suite de cette section, nous considérons la manière dont sont modélisés certains éléments de la spécification des classes Java compilées [LY99], dans les différents systèmes considérés. L'objectif est d'analyser sur ces points précis, si les systèmes appliquent notre principe 4 de Généralité des Systèmes de Transformation, c'est-à-dire permettent de manipuler tous les éléments formant une classe Java compilée, tout en appliquant notre principe 6 de Composition Efficace des Transformations, c'est-à-dire en automatisant et en factorisant le traitement des détails difficiles de la spécification.

Tous les systèmes de transformation de bytecode permettent de transformer les classes au chargement par la JVM, comme expliqué dans la section 3.2.3. Une classe Java compilée est donc le plus gros granule de transformation, dans tous les systèmes. Nous considérons donc ici le « découpage » d'un modèle d'une classe Java compilée en les types d'éléments qui la composent.

Systèmes à modèles multiples Javassist a un modèle conceptuel à *gros grain*. En effet, Javassist a été conçu pour offrir une interface de programmation proche de celle d'un protocole à méta-objets [Chi00, Tat02] (cf. la section 2.1). Le modèle de classe Java compilées qu'il offre (paquetage javassist) est donc à gros grain, et n'offre qu'un ensemble d'abstractions limité : attributs, méthodes, constructeurs, relations d'héritage, modificateurs d'accès, arguments formel, instructions de création d'objets, et instructions d'accès aux attributs. Les transformateurs Javassist ne manipulent que des abstractions du langage Java. Tous les détails des classes compilées sont cachés. Javassist offre une autre interface (paquetage javassist.bytecode), de plus bas niveau d'abstraction (classes Bytecode, CodeAttribute, CodeIterator, etc.), qui est cependant trop basique pour être utilisable dans des transformations complexes car les instructions dans les méthodes ne sont même pas décodées. Nous ne considérons donc que le modèle conceptuel de haut niveau de Javassist.

BCEL offre également deux librairies de classes Java pour représenter les éléments des classes Java compilées : une librairie pour des objets modifiables (ClassGen, FieldGen, MethodGen, etc.) dans le paquetage org.apache.bcel.classfile, et une autre pour des objets non modifiables (JavaClass, Field, Method, etc.) dans le paquetage org.apache.bcel.generic. Les objets modifiables servent à générer des objets non modifiables, par exemple un objet ClassGen sert à générer

un objet JavaClass. Les objets non modifiables ne servent qu'à l'analyse de programmes, mais sont moins coûteux en mémoire et en temps de création que les objets modifiables. Nous n'étudions dans ce chapitre que la librairie d'objets modifiables de BCEL, parce que nous nous intéressons aux capacités offertes par ce système pour la *transformation* de programmes, et parce que la librairie de classes d'objets modifiables est beaucoup plus complète que la librairie de classes d'objets non modifiables. Notamment, les instructions dans les méthodes sont manipulables seulement dans les objets MethodGen, et pas dans les objets Method.

Tables de constantes La table de constantes (*constant pool*), dans une classe compilée, contient et indexe toutes les constantes utilisées dans la classe : les chaînes constantes, les noms des classes, méthodes et attributs déclarés et accédés, les constantes numériques, etc. Par exemple, dans une instruction qui accède à un attribut (getfield), le nom et le type de l'attribut, et le nom de la classe qui définit l'attribut, sont indiqués par un index dans le code de l'instruction, qui référence des chaînes constantes contenant ces noms. Ainsi, renommer un attribut nécessite d'ajouter une constante dans la table, pour le nouveau nom, dans la classe qui définit l'attribut et dans chaque classe accédant à l'attribut, et de modifier la définition de l'attribut, et toutes les instructions qui accèdent à cet attribut, pour prendre l'index de la nouvelle chaîne constante. **JOIE**, **BCEL** et **Serp** exposent explicitement les index de constantes. Cependant, ces systèmes automatisent la factorisation des constantes en fournissant des méthodes de recherche et d'insertion de constantes. Ainsi, si deux constantes sont identiques dans une même classe, elle ne sont pas dupliquées et ne correspondent qu'à une seule entrée dans la table de constantes. Dans **Javassist**, **ASM** et **Jabyce**, les index de constantes sont complétement cachés et les constantes sont manipulées directement sous forme de chaînes ou de valeurs numériques. La factorisation des constantes est automatique, comme dans JOIE, BCEL et Serp.

Adresses de branchement Une méthode peut contenir des instructions de transfert de contrôle à d'autres instructions dans la même méthode. Les instructions auxquelles le contrôle est transféré sont spécifiées dans les instructions de saut par des décalages relatifs, en nombre d'octets. Chaque instruction a une adresse unique, qui est le nombre d'octets depuis le début de la méthode. Ainsi, insérer ou enlever une instruction dans une méthode peut éventuellement nécessiter de modifier les décalages dans les instructions de saut. Dans **Javassist**, les adresses de saut ne sont pas manipulées, car les transformations ne manipulent que des abstractions du langage Java. Dans **JOIE**, **BCEL** et **Serp**, les adresses de saut sont représentées par des références directes vers les objets représentant les instructions pointées, comme illustré dans la section 5.3 pour Serp. Dans **ASM** et **Jabyce**, les adresses de saut sont représentées par des objets Java abstraits qui jouent le rôle d'identifiants (InstructionLabel dans Jabyce, Label dans ASM), et qui sont représentés dans les méthodes avant les instructions pointées. Dans tous les systèmes, les décalages pour les branchements sont calculés automatiquement.

Variables locales Dans une implantation de méthode, chaque variable locale est identifiée par son index, et sa taille (32 ou 64 bits) qui est fonction du type de donnée qu'elle contient. L'index est spécifié dans chaque instruction qui lit ou écrit dans une variable locale. La référence vers l'objet « this » et les valeurs d'arguments sont passées comme les valeurs initiales de variables locales, dont les index sont 0 pour « this », et 1 et plus pour les arguments,

dans l'ordre donné dans la signature de la méthode. Les variables locales sans valeur initiale ont les index suivants. Ainsi, la transformation de la signature d'une méthode pour ajouter un argument formel nécessite de modifier les index des variables locales sans valeur initiale, dans toutes les instructions qui y accèdent. Dans l'interface de haut niveau de **Javassist**, il n'est pas possible de modifier les signatures de méthodes, ni de manipuler les variables locales. Dans **JOIE**, **BCEL**, **ASM** et **Serp**, les index et tailles de variables locales sont manipulées explicitement par les transformateurs. Une particularité de **Jabyce** est de rendre abstraits les index de variables locales en identifiant de manière unique chaque variable locale par un objet implantant l'interface FrameSlotIdentifier définie dans Jabyce. Un tel objet est passé lors de la représentation d'une instruction qui accède à une variable locale. Lors de la représentation d'une variable locale sans valeur initiale, un objet FrameSlotIdentifier est passé pour l'identifier dans les instructions qui accèdent à cette variable, et le prochain index inutilisé est affecté à cette variable. Lors de la représentation d'un argument formel dans la signature d'une méthode (type d'élément MethodFormalArgument), un objet FrameSlotIdentifier est passé en argument pour identifier la variable locale dont la valeur initiale est la valeur de l'argument, l'index de cette variable est automatiquement calculé, et les index des variables déjà représentées sont automatiquement modifiés dans toutes les instructions déjà représentées. Les transformateurs Jabyce ne manipulent donc jamais directement d'index de variables locales. Il est ainsi très facile avec Jabyce d'implanter des transformations qui ajoutent ou suppriment des arguments formels et des variables locales, dans les méthodes. Il n'existe pas à notre connaissance d'autres systèmes qui automatisent ce traitement des index de variables locales. Le choix de factoriser ces traitements est une application de notre principe 6 de Composition Efficace des Transformations.

Homogénéité des groupements de types d'instruction Plusieurs instructions du bytecode Java sont des variantes les unes des autres et ont une sémantique similaire. Ces *familles d'instructions* requièrent d'être transformées de manière similaire. Par exemple, les instructions getstatic, putstatic, getfield et putfield sont des instructions d'accès aux attributs de classes et d'instances. Lors de la transformation d'attributs, il est nécessaire de transformer les instructions de ces quatre types. En application du principe 8 de Fermeture Commune, un système de transformation doit fournir des informations pour facilement identifier les types d'éléments qui nécessitent des transformations similaires, de manière à pouvoir encapsuler ensemble ces transformations. Ainsi, **BCEL** définit une classe Java pour la représentation de chaque type d'instruction, et des classes abstraites héritées par ces classes, pour identifier les instructions similaires. Par exemple, la classe abstraite FieldInstruction est héritée par les classes GETSTATIC, PUTSTATIC, GETFIELD et PUTFIELD. Au total, BCEL définit 180 classes pour représenter les instructions de bytecode, ce qui rend cette bibliothèque complexe. **Serp** a une approche similaire, en définissant une classe Instruction, héritée par 31 classes pour représenter les instructions, y compris des classes abstraites comme dans BCEL pour regrouper les instructions avec des sémantiques similaires. **JOIE** définit une classe Instruction, héritée par 17 classes pour représenter les instructions, p.ex. Getfield, Putfield servent à représenter les instructions d'accès aux attributs. Le caractère statique ou non d'un attribut accédé est un attribut de ces classes. Cependant, tous les types d'instructions ne bénéficient pas d'une classe dédiée pour les représenter, et certaines instructions, comme monitorenter et monitorexit, sont représentées par de simples objets Instruction. Le modèle n'est donc *pas homogène*, ce qui rend plus difficile la transformation de certaines instructions, et notamment

impose de comparer les *opcodes* des instructions pour différencier certains types d'instructions. Or, d'après notre principe 4 de Généralité des Systèmes de Transformation, le modèle conceptuel doit être homogène pour permettre l'implantation de n'importe quels transformateurs de manière homogène. Dans **ASM**, l'interface CodeVisitor définit 12 méthodes pour des interactions représentant des instructions, qui correspondent au groupement de types d'instructions ayant la même *structure*. Dans certains cas, ce groupement structurel correspond aussi à un groupement sémantique. Par exemple, la méthode visitFieldInsn permet de représenter les instructions d'accès aux attributs. Ces instructions ont une sémantique similaire, mais aussi la même structure : un identifiant de type d'instruction (*opcode*), le nom de l'attribut et de la classe qui le définit, et le type de l'attribut. Dans d'autres cas, une même méthode permet de représenter des instructions ayant la même structure mais des *sémantiques différentes*. Par exemple, la méthode visitInsn permet de représenter les instructions dont la structure ne contient qu'un *opcode*, comme pop qui enlève les valeurs au sommet de la pile, et i2l qui convertit une valeur entière 32 bits en valeur entière 64 bits. Les transformations ASM, qui implantent cette interface, doivent donc manipuler des instructions sémantiquement différentes, et analyser leurs *opcodes* pour les différencier. Dans **Jabyce**, nous avons suivi une démarche similaire à BCEL et Serp, en regroupant les instructions sémantiquement similaires dans des types d'éléments. Contrairement à ASM, dans Jabyce un type d'élément regroupe seulement des éléments sémantiquement similaires. Contrairement à JOIE, le modèle est homogène, c'est-à-dire il n'existe pas de type d'élément « non typé » pour représenter une partie des instructions. Par exemple, l'interface FieldAccessInstructionFactory permet de représenter toutes les instructions qui accèdent à des attributs, l'interface StackInstruction permet de représenter toutes les instructions qui empilent ou dépilent des valeurs comme pop, l'interface ArithmeticTypeConversionInstructions permet de représenter les instructions de conversion comme i2l. Jabyce définit 30 interfaces pour la représentation des instructions, avec un modèle similaire à celui de Serp. Dans BCEL, Serp et Jabyce, les *opcodes* des instructions sont cachés, c'est-à-dire que le décodage des *opcodes* est factorisé dans les désérialiseurs et sérialiseurs, ce qui facilite l'implantation des transformations. Dans **Javassist**, seuls certains types d'instructions sont manipulables : la classe CodeConverter ne définit des méthodes que pour manipuler les instructions d'accès aux attributs (redirectFieldAccess, replaceFieldRead, replaceFieldWrite), les instructions d'appel de méthode (redirectMethodCall) et les instructions de création d'objets (replaceNew). Les autres instructions ne sont pas manipulables.

Formes d'instructions optimisées Certaines instructions ont des formes optimisées. Par exemple, une instruction iload charge une valeur à partir d'une variable locale, dont l'index est spécifié dans l'instruction après son *opcode*. Une instruction iload est donc codée sur deux octets. Il existe cependant quatre formes optimisées de cette instruction, avec des *opcodes* distincts et codées sur un seul octet, pour accéder aux variables d'index 0, 1, 2 et 3 : iload_0, iload_1, iload_2 et iload_3. Tous les systèmes étudiés, y compris Jabyce, ne définissent pas de types d'éléments spécifiques pour ces formes optimisées d'instructions, et sérialisent automatiquement les instructions dans leur forme la plus optimisée.

Complétude des définitions de relations de composition et de dépendance

Il existe des *relations entre les types d'éléments*. Comme introduit dans la section 5.3, nous distinguons les relations de *composition* et les relations de *dépendance sémantique*. Par exemple,

une relation de composition existe entre les classes et les méthodes.

En analogie avec la distinction entre objets et classes dans la programmation par objets, nous considérons que les relations qui peuvent être *concrétisées* entre les représentations d'éléments de programmes sont définies au niveau de *l'implantation* du système de transformation, comme décrit dans la section 5.3. Serp, JOIE, BCEL et Javassist offrent des classes Java entre lesquelles sont définies des *associations*, qui permettent de concrétiser les relations de composition et de dépendance par des *liaisons* entre les objets qui représentent les éléments de programmes. Dans ASM et Jabyce, les relations qui peuvent être concrétisées (par exemple par l'égalité des paramètres) sont définies par l'ensemble des signatures des méthodes dont les appels représentent des éléments. Cependant, Jabyce est le seul système dans lequel la définition des relations entre éléments est réalisée à un niveau d'abstraction supérieur à l'implantation du système de transformation. En effet, comme décrit dans la section 3.3, un outil automatique (Jaidee) est utilisé pour générer le code source des interfaces de Jabyce, et donc les signatures des méthodes dont les appels représentent des éléments de programmes. Cette génération est paramétrée par une définition du modèle conceptuel sous la forme d'un fichier XML. Ce modèle définit des dépendances entre les types d'éléments, dont *seulement certaines se traduisent dans les signatures des méthodes* générées. Par exemple, dans Jabyce, une relation de dépendance est définie entre le type d'élément Field et le type d'élément FieldAccessInstruction, et une telle relation n'est pas concrétisée explicitement entre les représentations d'attributs et d'instructions, comme expliqué à la fin de la section 5.3. Ces relations définies mais non concrétisées sont essentiellement les relations inter-classes, comme les relations d'héritage, de classes internes et de déclaration - utilisation d'attributs et de méthodes. Cette caractéristique de Jabyce est motivée par un point de vue différent sur l'utilité des définitions de relations entre types d'éléments. En effet, les relations sont définies dans Jabyce avec pour objectif premier de déterminer *les dépendances sémantiques entre les transformations* de types d'éléments. Notre objectif est de pouvoir encapsuler les implantations de transformations dépendantes, selon le principe 8 de Fermeture Commune. Cette approche est détaillée dans les sections 3.2.6 et 3.3. Ainsi, une originalité de notre proposition est de considérer les relations entre types d'éléments, non seulement 1) pour permettre aux transformateurs d'analyser les programmes représentées en s'appuyant sur les relations concrétisées, mais aussi 2) pour garantir une bonne encapsulation et réutilisation des transformateurs, en garantissant une bonne encapsulation des traitements interdépendants. Dans tous les autres systèmes, la problématique de l'encapsulation des transformations dépendantes n'est pas abordée.

Plus généralement, la problématique des relations entre éléments de programmes concerne à la fois les dimensions 1) du modèle conceptuel pour la *complétude* des relations identifiées, 2) du mécanisme de représentation interne (objets *vs.* interactions) pour *l'homogéneité* et *la facilité d'analyse* des représentations de relations (cf. la section 5.3), et 3) du modèle d'implantation des transformateurs (cf. la section 5.5) pour *l'encapsulation* des transformations sémantiquement dépendantes. La complétude des relations identifiées est un critère important, parce qu'elle a un impact positif sur la facilité d'analyse des relations, et sur la facilité d'encapsulation des transformations dépendantes.

En considérant seulement les relations concrétisées entre des représentations d'éléments, Jabyce offre la représentation d'un plus grand nombre de relations de dépendance entre éléments, que dans les autres systèmes. Notamment, comme décrit ci-dessus, Jabyce est le seul système qui représente les relations entre déclarations et utilisations d'arguments formels (représentation des index sous forme d'objets en paramètres). De plus, en considérant plus généralement les

relations définies entre les *types* d'éléments, Jabyce est le seul système qui définit des relations inter-classes. Jabyce est donc le système le plus complet en ce qui concerne les relations définies entre les types d'éléments de programmes.

Synthèse

Jabyce offre le modèle conceptuel de classes Java compilées le plus *abstrait* après celui de Javassist, c'est-à-dire celui qui abstrait le plus les détails du format sérialisé des classes (p.ex. les index de variables locales), afin de factoriser au maximum les traitements associés à ces détails dans les (dé)sérialiseurs, en application de notre principe 6 de Composition Efficace des Transformations. Jabyce permet toujours cependant de transformer tous les éléments formant une classe, comme les autres systèmes sauf Javassist, en application de notre principe 4 de Généralité du Système de Transformation. Jabyce offre également l'un des modèles les plus *homogènes*, c'est-à-dire que le découpage des types d'éléments ne favorise pas certaines catégories d'instructions en termes de facilité d'implantation, contrairement à JOIE par exemple. La complexité du modèle conceptuel de Jabyce, c'est-à-dire le nombre de types d'élément définis, est moyenne comparée à celle des autres systèmes. Le modèle conceptuel de Serp est le plus proche de celui de Jabyce. Le modèle conceptuel de Jabyce est celui qui identifie le plus grand nombre de relations (de composition et de dépendance) entre les types d'éléments de programmes.

5.5 Paradigme de transformation

Cette dimension concerne les modèles d'architecture et de traitement pour l'implantation des transformateurs. Nous considérons que ces modèles doivent apporter des réponses aux questions suivantes.

Comment exprimer des transformations ?

Tous les systèmes considérés, y compris Jabyce, sont développés en Java et offrent des mécanismes pour implanter des transformateurs en Java (*règles procédurales*).

Comment empaqueter et déployer des transformations ?

Java étant un langage de programmation par objets, il offre naturellement des mécanismes permettant d'encapsuler et de réutiliser des implantations de transformateurs. Cependant, afin de considérer de manière homogène les transformateurs et de systématiser leur réutilisation et leur composition, en application de notre principe 5 d'Encapsulation et de Composabilité des Transformations, la plupart des systèmes définissent des abstractions, p.ex. des interfaces Java qui doivent être implantées par les classes Java implantant des transformateurs. Ces abstractions diffèrent suivant les systèmes. Seuls **BCEL** et **Serp** n'offrent pas d'abstractions pour implanter des transformateurs. **JMangler** définit une interface CodeTransformerComponent pour l'implantation de transformateurs d'implantations de méthodes, et une interface InterfaceTransformerComponent pour l'implantation de transformateurs des

« éléments externes » des interfaces et des classes Java (attributs et méthodes non privés, relations d'héritage, etc.). JMangler limite les transformations implantables pour garantir la *compatibilité binaire* [DWE98] des classes Java transformées : n'importe quelle transformation d'implantation de méthode est implantable, mais les transformations des « éléments externes » se limitent à *l'ajout* de membres, de relations d'héritages, etc. Il n'est par exemple pas possible de renommer des méthodes avec JMangler. **JOIE** définit une interface ClassTransformer, qui définit seulement une méthode transform avec comme argument un objet ClassInfo racine du graphe d'objets représentant une classe à transformer. Il à noter qu'il n'est pas possible de représenter de nouvelles classes avec JOIE : il est possible seulement de transformer des classes existantes. **Javassist** définit une interface Translator similaire à l'interface ClassTransformer de JOIE. **ASM** définit deux interfaces, ClassVisitor et CodeVisitor, que doivent implanter les transformateurs, qui sont conçus comme des décorateurs d'autres objets de type ClassVisitor et CodeVisitor, de manière similaire à Jabyce (cf. la section 3.2.5). **Jabyce** définit les transformateurs comme des composants logiciels Fractal, et définit les interfaces des composants, permettant de représenter des éléments de programmes, comme décrit dans la section 3.2.6.

Comment contrôler l'environnement des transformateurs ?

Dans Jabyce, l'utilisation du modèle Fractal offre la meilleure maîtrise de la conception de configurations de transformateurs complexes, y compris de transformateurs liés à d'autres composants non transformateurs, c'est-à-dire des *règles procédurales paramétrées*, et de composants contrôlant la configuration de transformateurs. En effet, les modèles offerts par les autres systèmes ne couvrent que la modélisation des transformateurs, mais pas d'un programme de transformation dans son ensemble.

Comment encapsuler des implantations de transformations causalement dépendantes ?

Lors de l'implantation d'un transformateur, il est nécessaire de prendre en compte les relations de dépendance entre les éléments de programme, p.ex. les relations déclaration - utilisation d'attributs, afin d'éviter de violer des relations et donc d'introduire des erreurs de sémantique dans les programmes représentés. Pour éviter de telles fautes de développement, il est nécessaire de bien encapsuler dans une même implantation de transformateur les transformations d'éléments dépendants, c'est-à-dire les *transformations causalement dépendantes*. Tous les systèmes étudiés permettent de bien encapsuler des transformateurs, notamment grâce à l'utilisation d'un langage de programmation par objets comme Java. Cependant, aucun système n'impose d'encapsuler les implantations de transformations causalement dépendantes. Par exemple, aucun système n'impose d'encapsuler dans un même transformateur des implantations de transformations d'attributs et d'instructions d'accès aux attributs. En revanche, Jabyce *guide* cette encapsulation lors de l'implantation de transformateurs, bien qu'il ne l'impose pas. En effet, Jabyce définit des classes Java abstraites pour l'implantation des composants transformateurs, dont les héritages d'interfaces correspondent aux dépendances causales entre les transformations à effectuer. Ces relations d'héritage imposent aux programmeurs de transformateurs de prendre en compte les relations de dépendance, ce qui permet a priori de réduire le risque de faute de développement. Cette problématique nouvelle est détaillée dans la section 3.2.6, et à la fin de la section 5.4.

5.6 Stratégie de transformation

L'exécution d'un ensemble de transformations implique une *relation de réécriture* entre des programmes [Vis01]. Cette relation est soit *confluente* si tout ordre d'exécution des transformations aboutit à un même programme à partir d'un programme donné, soit *non-confluente* si l'ordre d'exécution des transformations a un impact sur les programmes résultants. En pratique, un ensemble de transformations implique généralement une relation non-confluente. Il est donc nécessaire de contrôler l'ordre d'exécution des transformations, c'est-à-dire la *composition* des transformations, pour choisir un chemin dans la relation de réécriture entre programmes. Ce choix est d'autant plus important qu'une faute de composition peut provoquer des erreurs, par exemple rendre les programmes transformés incorrects, comme expliqué dans notre modèle de faute dans la section 4.1.2. [Vis01] définit une *stratégie de transformation* comme un algorithme pour choisir un tel chemin. Suivant les systèmes, une telle stratégie peut être spécifiée de différentes manières :

- stratégie unique imposée par le système de transformation ;
- stratégie déterminée par analyse automatique des relations entre transformations ;
- stratégie choisie par un utilisateur dans un ensemble restreint de stratégies proposées par le système ;
- stratégie complétement programmable par l'utilisateur, p.ex. en utilisant un langage spécifique (DSL) comme dans Stratego [Vis01].

Flexibilité pour la définition de stratégies de transformation

Nous considérons que l'application d'une stratégie de transformation est un *contrôle* particulier appliqué à une configuration de transformateurs. Dans tous les cas, quelque soit la manière de spécifier une stratégie, il est nécessaire de séparer ce contrôle de l'implantation des transformateurs, en application du principe 2 d'Inversion de Contrôle. De cette manière, des stratégies arbitraires doivent pouvoir être applicables à tout ensemble de transformateurs, sans devoir modifier leur implantation. Cette inversion de contrôle est mise en œuvre dans tous les systèmes de transformation considérés, y compris Jabyce. Dans **JMangler**, tout ensemble de transformations des « éléments externes » d'interfaces et de classes Java forme une relation confluente sur les programmes transformés, la stratégie étant fixée par JMangler : les transformateurs sont tous exécutés dans un ordre non déterministe, itérativement, jusqu'à ce qu'un point fixe soit atteint et que le programme ne soit plus transformé par aucun transformateur. En revanche, les transformateurs JMangler d'implantations de méthodes sont éxécutés dans un ordre séquentiel spécifié par l'utilisateur dans un fichier de configuration, donc dans un langage spécifique de spécification de stratégie de transformation. De la même manière, dans **JOIE**, les transformateurs sont exécutés dans un ordre séquentiel spécifié par l'utilisateur, l'ordre étant déterminé par la « priorité » donnée à chaque transformateur lors de leur enregistrement auprès du chargeur de classes implanté dans JOIE. Cependant, aucun langage spécifique n'est offert pour spécifier de telles stratégies : cela est réalisé par programmation. De plus, la classe implantant le chargeur de classes n'est plus disponible dans les dernières versions de JOIE, c'est-à-dire que JOIE ne fournit plus aucun mécanisme de mise en œuvre de stratégies de transformations. **Javassist** ne fournit pas explicitement de mécanisme pour la spécification de stratégies de transformation. Il est pourtant possible de lier par programmation des objets de type ClassPool, semblables à des « conteneurs » de classes

compilées, pour former une chaîne. Lors du chargement d'une classe par le chargeur de classes de Javassist, la classe est chargée en passant à la suite dans les conteneurs dans la chaîne, qui déclenchent chacun l'exécution d'un transformateur pour transformer la classe. Cependant, les classes sont passées sous une forme sérialisée (tableau d'octets) entre les objets conteneurs, ce qui impose que la classe soit désérialisée et sérialisée pour chaque transformation, ce qui implique une répétition inutile de ces traitements. Dans **ASM**, les transformateurs forment une chaîne, c'est-à-dire qu'ils sont composés séquentiellement, par programmation, en spécifiant à la création de chaque transformateur, c'est-à-dire de chaque objet héritant de ClassAdapter (qui implante l'interface ClassVisitor), la référence du prochain transformateur dans la chaîne. Le problème est qu'une chaîne d'objets décorateurs de type ClassAdapter doit être créée pour la transformation de chaque classe Java compilée. En effet, comme illustré dans la section 5.3, l'interface ClassVisitor ne permet de représenter qu'une seule classe. L'algorithme qui construit une chaîne de décorateurs, donc qui implante la stratégie de tranformation, doit être exécuté pour chaque classe. Dans **Jabyce**, les opérations de transformation sont composées en utilisant des liaisons entre composants Fractal spécifiées par l'utilisateur. Ces configurations forment généralement des chaînes, comme dans ASM. Cependant, contrairement à ASM, les opérations Jabyce permettent de représenter / transformer plusieurs classes, et l'algorithme de composition des opérations n'est exécuté qu'une seule fois.

Dans tous les systèmes considérés, une stratégie spécifie généralement explicitement une séquence fixe de transformations. Cependant, il est envisageable de pouvoir spécifier des stratégies plus complexes, en implantant des transformateurs spéciaux, que l'on peut qualifier de *commutateurs*, dont le rôle est uniquement d'activer ou pas des ensembles d'autres transformateurs. La configuration des commutateurs serait une partie intégrante d'une stratégie. Dans JMangler, JOIE, Javassist, ASM et Jabyce, un transformateur peut ainsi déléguer les traitements à d'autres transformateurs, qui portent les mêmes interfaces spécifiées par le système de transformation, en fonction d'un algorithme qu'il implante.

Paramétrage et configuration de l'environnement

Le paramétrage des transformateurs fait également partie d'une stratégie de transformation. Les valeurs d'attributs de transformateurs peuvent ainsi avoir une influence importante sur le comportement global d'une configuration de transformateurs. Ce paramétrage peut également être mis en œuvre par des configurations d'objets ou de composants, liés aux transformateurs, comme par exemple dans le patron de conception Stratégie [GHJV94] (cf. l'annexe B.3). Grâce à l'utilisation du modèle de composants Fractal, Jabyce est le seul système considéré qui offre des mécanismes et un langage (l'ADL Fractal) standards pour spécifier à la fois la composition des transformateurs, les attributs de transformateurs, et les configurations de composants liés aux transformateurs et nécessaires à leur exécution (formant leur *environnement*).

Stratégie de balayage, et redondance du balayage

Une partie importante d'une strategie de transformation est la *stratégie de balayage* (*traversal strategy*), qui est l'algorithme qui *balaie* les éléments formant un programme pour déclencher les transformations, indépendamment de l'ordre de déclenchement des transformations [vdBKV03, Vis01]. Plus précisément, dans une stratégie de transformation, la stratégie

de balayage détermine dans quel ordre les éléments sont transformés, et le reste de la stratégie détermine dans quel ordre les transformations sont exécutées sur chaque élément. Une stratégie de balayage est soit générale, soit spécifique à un modèle de programme, p.ex. à un langage de programmation. Une stratégie peut être par exemple *bottom-up* en transformant d'abord les éléments feuilles comme les instructions puis les éléments composites de plus haut niveau comme les classes, ou *top-down* en transformant d'abord les éléments composites puis les éléments de plus bas niveau. Dans tous les systèmes de transformation considérés, les classes Java compilées formant un programme sont transformées dans un ordre non déterministe, parce que tous les systèmes permettent de transformer les classes lors de leur chargement par la machine virtuelle, et que les classes sont chargées dans un ordre non déterministe. En revanche, les stratégies de balayage diffèrent en ce qui concerne les éléments à l'intérieur de chaque classe. Nous définissons deux critères de comparaison en ce qui concerne les stratégies de balayage : 1) la flexibilité pour la définition des stratégies de balayage, et 2) la redondance ou non du balayage d'une classe lors de sa transformation par une chaîne de transformateurs.

Dans **JMangler, JOIE** et **Javassist**, la stratégie de balayage des classes Java transformées est *programmée explicitement* dans chaque transformateur, et lui est spécifique. L'avantage est que le balayage est a priori optimal pour chaque transformateur, en termes de coût et de simplicité. Cependant, dans une chaîne de transformateurs, une classe transformée est balayée par chaque transformateur, ce qui implique une *redondance* de ces traitements, en contradiction avec notre principe 6 de Composition Efficace des Transformations.

Dans **ASM** et **Jabyce**, en revanche, une classe est *balayée une seule fois* pour toutes les transformations réalisées par une configuration de transformateurs. Une stratégie de balayage, dans ASM et Jabyce, détermine un ordre dans lequel sont initiées les interactions dans une séquence représentant une classe Java compilée, et détermine donc implicitement un ordre d'exécution des transformations. En effet, la représentation d'un élément par une interaction, et sa transformation, sont simultanées. Comme décrit dans la section 3.2.5 pour Jabyce, aucun ordre n'est généralement imposé pour les interactions. Une stratégie de balayage est *implicite* : elle n'est pas explicitement spécifiée par un algorithme, et « émerge » implicitement de la composition des sérialiseurs et transformateurs dans une chaîne. Par exemple, les séquences d'interactions représentant des classes différentes peuvent être entrelacées arbitrairement, parce qu'un transformateur peut représenter une nouvelle classe pendant qu'elle transforme un élément dans une autre classe. Ainsi, il n'est pas possible de contrôler explicitement le balayage d'une classe, en implantant un algorithme comme dans JMangler, JOIE et Javassist, ce qui rend plus difficile l'implantation de transformations qui manipulent plusieurs éléments de programmes et donc dépendent de la stratégie de balayage.

ASM et Jabyce définissent cependant des propriétés temporelles pour les séquences d'interactions possibles représentant des programmes, et donc limitent l'ensemble des stratégies de balayage possibles. Par exemple, dans ASM les méthodes de l'interface ClassVisitor doivent être appelées dans un ordre spécifié dans la documentation de l'interface sous la forme de l'expression rationnelle suivante : visit (visitField | visitMethod | visitInnerClass | visitAttribute)* visitEnd. Cet ordre des appels est illustré dans le programme d'exemple 5.3 de la section 5.3. Autre exemple, dans Jabyce les variables locales doivent être balayées avant les instructions qui utilisent ces variables locales. De plus, dans Jabyce le balayage des éléments composites et des éléments qu'ils contiennent récursivement, doit respecter un ordre bien défini, cf. la section 3.2.5. Dans ASM peu de vérifications sont réalisées à l'exécution pour garantir de telles propriétés temporelles. En revanche, les travaux que nous avons menés autour de Jabyce,

décrits dans le chapitre 4, sur l'utilisation de modèles et de langages de spécifications de contrats entre composants, et la vérification de ces contrats pendant l'exécution, permettent entre autres de *vérifier des propriétés temporelles d'une stratégie de balayage* pendant l'exécution des transformateurs. Cette approche permet de diminuer la complexité des implantations de transformateurs, bien qu'elle reste plus élevée que dans JMangler, JOIE et Javassist.

Stratégies de balayage aboutissant à un point fixe

L'une des bonnes propriétés d'une stratégie de balayage est qu'elle aboutisse à un *point fixe*, c'est-à-dire que l'ensemble des transformations pour une classe Java se terminent. Cette propriété se décline en deux propriétés : 1) il ne doit pas y avoir de cycles dans la composition des transformateurs, 2) chaque exécution d'un transformateur doit se terminer. **JOIE**, **Javassist**, **ASM** et **Jabyce** ne permettent pas facilement de configurer des transformateurs en boucle, ni d'exécuter itérativement des transformateurs pour un même programme transformé. Ces systèmes offrent donc généralement des stratégies qui aboutissent à des points fixes, à condition que chaque exécution d'un transformateur aboutisse à un point fixe. C'est également le cas pour les transformateurs de code dans **JMangler**, mais pas pour les transformateurs des « éléments externes » d'interfaces et classes Java. En effet, ces derniers transformateurs sont exécutées itérativement par JMangler jusqu'à un point fixe caractérisé par un programme qui n'est plus transformé par les transformateurs. Une telle itération peut ne jamais se terminer.

Synthèse

Tous les systèmes offrent un moyen de spécifier des stratégies de transformation, c'est-à-dire des compositions de transformateurs, indépendamment des implantations de transformateurs. Seul JMangler impose une stratégie, pour les transformations d'interfaces externes de classes et interfaces compilées. Jabyce offre le mécanisme de spécification de stratégies le plus général et le plus flexible, en s'appuyant sur les mécanismes de liaisons entre composants et le langage ADL de Fractal, qui permettent également de paramétrer et de configurer les transformateurs et les composants formant leur environnement de manière homogène.

Seuls les systèmes offrant une représentation des classes Java par des graphes d'objets (JMangler, JOIE et Javassist) permettent de spécifier la stratégie de balayage des classes transformées, en implantant un algorithme dans chaque transformateur. Cependant, le balayage est effectué dans chaque transformateur d'une classe, ce qui implique une redondance inutile de ces traitements. En revanche, dans ASM et Jabyce, chaque classe transformée n'est balayée qu'une seule fois pour une chaîne de transformateurs, ce qui optimise les traitements, au prix d'une plus grande complexité dans l'implantation des transformateurs. En effet, dans ASM et Jabyce, le balayage d'une classe est non déterministe, bien qu'un balayage doive respecter certaines propriétés temporelles, ce qui complexifie les implantations de transformateurs.

5.7 Vérification des transformations

Nous avons introduit dans le chapitre 4 la problématique nouvelle de la vérification du comportement de transformateurs de programmes, à l'intersection des domaines de la transformation de programme et de la sûreté de fonctionnement. Notre objectif est de vérifier, pendant

l'exécution des transformateurs, que chaque transformateur (chaque composant opération, dans Jabyce), produit seulement des représentations internes de programmes corrects.

Comme décrit dans la section 4.1.2, nous avons défini deux types d'erreur (correction syntaxique et correction sémantique), qui correspondent aux différentes passes de l'algorithme de vérification de la correction des classes compilées, dans une JVM. Un premier critère de comparaison des systèmes est *l'ensemble des erreurs détectées* par les différents systèmes, c'est-à-dire notamment les niveaux de correction considérés. Nos travaux se placent dans le contexte du test des transformateurs de programmes. Dans ce contexte, notre deuxième critère est la *précision du diagnostic* des erreurs de correction, c'est-à-dire notamment la précision de la localisation des fautes de dévéloppement dans les transformateurs fautifs.

Programmation défensive

Seuls **JOIE** et **Javassist** implantent une vérification de la correction des représentations d'éléments par *programmation défensive*, c'est-à-dire par des tests explicites des paramètres en début des méthodes, dans les classes d'objets représentant des éléments de programme. Ces tests ont été codés à la main, par les développeurs de JOIE et Javassist. La programmation défensive est utile pour vérifier une certaine intégrité des programmes représentés. Par exemple, dans **Serp**, il est possible de ne pas renseigner les informations sur l'attribut accédé par une instruction getfield, c'est-à-dire donner des valeurs null pour le nom de l'attribut, le nom de la classe qui le définit et son type dans un objet GetFieldInstruction qui la représente, ce qui ne provoque aucune erreur. Cependant, cela se traduit lors de la sérialisation par la génération d'une instruction qui contient l'index de constante 0, ce qui est une erreur dans la majorité des cas car la constante à cet index ne sera pas une entrée du type « référence d'attribut ». Le même problème apparaît avec **BCEL** (et donc dans **JMangler**) : à la création d'un objet GETFIELD, il est possible de passer n'importe quelle valeur d'index de constante en paramètre, et il n'est pas vérifié que cet index est celui d'une constante de type « référence d'attribut ». Dans **ASM** également, il n'est pas vérifié dans le sérialiseur (classe CodeWriter) si les paramètres à un appel à visitFieldInsn(...) ne sont pas null, pour représenter une instruction getfield, ce qui provoque la levée d'exceptions peu explicites. En revanche, **JOIE** est plus défensif que les autres systèmes, grâce à des tests explicites dans les constructeurs et les méthodes, et par la restriction de l'utilisation de la plupart des méthodes de modification d'état par le modificateur protected. Il n'est par exemple pas possible de représenter une instruction getfield (par un objet Getfield) avec un index d'une constante d'un autre type que « référence d'attribut ». L'inconvénient de cette approche dans JOIE, est de limiter la généralité du système de transformation, comme expliqué à la fin de la section 5.3. Dans **Javassist** également, des vérifications de validité des paramètres sont explicitement codées dans le code des méthodes, mais sans limiter les transformations possibles.

Le seul apport du code de programmation défensive, est de rendre plus explicite les messages dans les exceptions, et de pouvoir détecter les erreurs plus tôt, qu'en absence de détection d'erreur. Cependant, avec une représentation des programmes par des graphes d'objets, la programmation défensive est limitée pour *diagnostiquer et localiser* précisément les fautes de développement qui provoquent ces erreurs. En effet, un graphe d'objets représentant des éléments de programme est manipulé par plusieurs fragments de code, *dispersés* dans une implantation de transformateur. Ainsi, le code qui provoque une erreur détectée par programmation défensive dans le code des objets du graphe, n'est pas nécessairement le seul code

fautif dans l'implantation du transformateur. Il est donc difficile de localiser les fautes de développement avec la programmation défensive et une représentation en graphes d'objets.

En revanche, avec une représentation en séquences d'interactions, comme dans ASM et Jabyce, la représentation d'un élément de programme (par un appel de méthode) et sa transformation sont simultanées. L'interaction représentant un élément est déclenchée par un fragment précis de l'implantation du transformateur. Dans ce contexte, la programmation défensive permet de localiser plus précisément le code fautif.

Cependant, le problème fondamental avec la programmation défensive, est le *mélange* entre le code de vérification et le code fonctionnel. Ce mélange complexifie inutilement les implantations des classes de graphes d'objets dans JOIE, ou le code des transformateurs avec une représentation par interactions. Le code de détection d'erreur est dispersé dans tout le code fonctionnel, ce qui empêche d'avoir une idée précise des erreurs qui sont effectivement détectées, et limite donc la confiance que l'on peut avoir dans le système. De plus, la programmation défensive se limite généralement à la détection d'erreurs de syntaxe des représentations d'éléments (vérification d'un seul élément à la fois), et n'est pas utilisable pour la détection d'erreurs de sémantique (vérification de plusieurs éléments). Par exemple, dans JOIE, malgré l'utilisation de programmation défensive, il n'est fait aucune vérification que tout chemin d'exécution dans une implantation de méthode aboutit à une levée d'exception où à une instruction xreturn.

Domaines de traitement d'erreur

Il est donc souhaitable de ne pas utiliser la programmation défensive pour la vérification de la correction des représentations de programmes. Dans **Jabyce**, comme détaillé dans la section 4.2.1, nous avons appliqué notre principe 10 d'Inversion du Traitement d'Erreur, en séparant complétement code fonctionnel et code de traitement d'erreur, et en proposant un mécanisme à la fois plus général, plus homogène et plus puissant que la programmation défensive, pour traiter à la fois les erreurs de syntaxe et de sémantique. Notre approche, que nous appelons *domaines de traitement d'erreur* d'un point de vue architectural, est basée sur l'interception transparente des interactions représentant des éléments de programmes, et sur la vérification de *contrats*, spécifiés dans des formalismes de haut niveau. Cette approche originale est détaillée dans le chapitre 4.

Algorithme de vérification

Cependant, une limite de notre approche, est que les spécifications de contrats que nous avons produites ne couvrent pas pour l'instant toute la spécification de la JVM en ce qui concerne la vérification des programmes [LY99]. Seul **BCEL** permet d'effectuer une vérification complète des programmes, en offrant une implantation littérale et complète de *l'algorithme de vérification* de la JVM (dans le paquetage org.apache.bcel.verifier). Ainsi, comme Jabyce, BCEL permet de vérifier à la fois la syntaxe et la sémantique des classes Java compilées. Cependant, l'inconvénient de l'utilisation d'un algorithme de vérification est que la localisation des fautes de développement qui ont produit les erreurs détectées sont difficilement localisables, encore plus difficilement qu'avec la programmation défensive, car une classe ne peut être vérifiée par l'algorithme qu'après qu'elle aie été complétement transformée. De plus, avec l'algorithme de vérification, la vérification n'est pas modulaire. Si l'on souhaite vérifier seulement certains

aspects lors du test de transformateurs, il est nécessaire d'exécuter l'algorithme complétement. En revanche, notre approche dans Jabyce, comme indiqué ci-dessus, offre la meilleure précision pour la localisation des fautes, et nous avons une approche modulaire pour la spécification et la vérification des contrats de correction, en permettant également de combiner plusieurs formalismes de spécification de contrats.

Synthèse

ASM et Serp n'offrent aucun mécanisme de détection d'erreur de la correction des programmes dans leur représentation interne. Javassist et JOIE implantent une programmation défensive, qui se limite à la syntaxe des éléments de programmes représentés. L'approche de vérification de la correction des programmes la plus complète est celle de BCEL, qui implante complétement l'algorithme de vérification de la JVM. Cependant, Jabyce offre l'approche la plus générale, la plus modulaire et la plus précise pour la localisation des fautes de développement dans les transformateurs qui provoquent des erreurs de corrections des programmes représentés.

5.8 Représentation de fragments de programmes

Les implantations de transformateurs comprenent des fragments de programmes pour deux fonctions [Vis02] : 1) pour la détection de patrons (*pattern matching*) d'ensembles d'éléments de programmes à transformer, et 2) pour la production d'éléments de programmes. La syntaxe pour spécifier ces fragments de programmes peut être *abstraite* (*abstract syntax*) si le langage d'implantation des transformateurs est directement utilisé, ou *concrète* (*concrete syntax*) si le langage des programmes transformés est utilisé.

JMangler et JOIE offrent une syntaxe abstraite : les fragments sont spécifiés sous la forme de code Java (le langage d'implantation des transformateurs) qui crée, transforme ou supprime des objets Java représentant des éléments de programmes. ASM et Jabyce également offrent une syntaxe abstraite : les fragments sont spécifiés sous la forme de code Java qui recoivent ou émettent des appels de méthodes Java qui représentent des éléments de programmes, comme illustré dans la section 5.3 dans les programmes 5.3 et 5.4.

En revanche, la grande originalité de Javassist est d'offrir une syntaxe concrète pour la représentation de fragments de code compilé. Ce mécanisme se limite à la production de nouveaux blocs d'instructions de bytecode, et n'est pas utilisable pour le *pattern-matching* d'éléments à transformer. La deuxième originalité de Javassist est que ces fragments de code compilé Java sont spécifiés sous la forme de *code source Java*. Par exemple, le programme 5.5 donne partiellement l'implantation d'un transformateur Javassist qui insère des instructions de bytecode avant chaque instruction *x*return dans une méthode, spécifié sous la forme de code source Java. Le code inséré affiche sur la console un message de trace d'exécution.

Comme illustré dans cet exemple, les fragments en syntaxe concrète sont spécifiés sous la forme de chaînes contenant du code source Java, qui sont passées en paramètres lors d'appels de méthodes des objets représentant des méthodes ou des instructions. Ainsi, par exemple la classe CtMethod fournit la méthode insertBefore(...) pour ajouter du code au début de la méthode, la méthode addCatch(...) pour ajouter du code de traitement d'exception, la

```
CtClass c = ...;
CtMethod m = c.getDeclaredMethod("uneMethode");
String msg = "retour de " + m.getName() +
             " dans " + c.getName();
m.insertAfter("{ System.out.println(\"" + msg + "\"); }");
```

PROG. 5.5 : Code d'un transformateur Javassist utilisant un fragment de programme

méthode insertAfter(...) pour ajouter du code avant chaque instruction *x*return, et la méthode setBody(...) pour remplacer complétement le code de la méthode. Les classes Cast, FieldAccess, InstanceOf, MethodCall, et NewExpr, qui sont les seules classes définies dans Javassist pour représenter les instructions dans les méthodes, offrent une méthode replace(...), permettant de remplacer l'instruction par du code passé en paramètre sous la forme d'une chaîne. La classe Handler, dont les instances représentent des blocs de traitement d'exception, offre également des méthodes insertBefore(...) et replace(...). La syntaxe concrète de Javassist n'est utilisable qu'en paramètre de cet ensemble restreint de méthodes. Il n'est pas possible par exemple de spécifier de nouvelles relations d'héritage des classes transformées dans la syntaxe concrète, ou de remplacer des instructions d'autres types que les 5 modélisés dans Javassist (p.ex. les instructions d'accès aux variables locales).

Malgré cette limitation de la portée de l'utilisation de la syntaxe concrète, l'introduction d'une syntaxe concrète dans Javassist, et surtout l'utilisation de la syntaxe du langage Java, font de Javassist le système de transformation de bytecode Java le plus facile à utiliser parmi tous les systèmes considérés, et le seul qui ne nécessite pas de connaître les caractéristiques du format des classes compilées Java.

5.9 Synthèse

Ce chapitre offre une comparaison qualitative des systèmes généralistes de transformation de code compilé Java : JOIE, JMangler, Javassist, ASM, Jabyce, Serp et BCEL. Les tableaux 5.1*a* à 5.1*e* synthétisent nos différents critères de comparaison dans toutes les dimensions de notre catégorisation donnée dans la section 5.1. Seule la dimension de la portée (*scope*) n'est pas synthétisée ici, car elle nous a essentiellement servi comme premier filtre de sélection des systèmes à comparer (cf. la section 5.2). Nos objectifs sont spécifiés sous la forme de trois principes de conception, dans la section 3.1, et nous regroupons nos différents critères de comparaison par rapport à ces trois principes. Ainsi, l'évaluation de chaque système selon ces critères nous permet d'évaluer le degré d'application de nos trois principes, et donc l'atteinte de nos objectifs :

- Principe 4 de Généralité du Système de transformation : critères (1.1) et (1.2) pour évaluer les manipulations possibles des représentations d'éléments de programmes, et critère (2.2) car le modèle conceptuel doit être à un grain suffisamment fin pour permettre de transformer tous les éléments formant un programme ;
- Principe 5 d'Encapsulation et de Composabilité des Transformations : critères (3.1), (2.3) et (3.2) pour évaluer les abstractions et les mécanismes offerts pour encapsuler et réutiliser facilement les transformateurs, et notamment pour encapsuler des implantations de transformations causalement dépendantes (principe 8 de Fermeture Commune), et critère (4.1)

pour les mécanismes et langages offerts pour composer les transformateurs sans modifier leur code ;
- Principe 6 de Composition Efficace des Transformations : critère (2.1) pour la factorisation des traitements redondants des détails du format sérialisé des classes compilées dans les désérialiseurs et sérialiseurs, et critères (4.1) et (4.3) pour les optimisations et factorisations automatiques pouvant être réalisées lors de la composition des transformateurs, notamment la factorisation du balayage des classes.

Les critères (5.1) et (5.2), concernant la vérification de la correction des transformations, sont indirectement liés à la réutilisation des transformateurs : un transformateur n'est réutilisable que lorsque les utilisateurs ont une certaine confiance dans sa correction. Ces deux critères concernent donc notre principe 5 d'Encapsulation et de Composabilité des Transformations.

En conclusion, *Jabyce atteint nos trois objectifs*, en appliquant effectivement nos trois principes de conception. C'est aussi le seul système de transformation évalué qui atteint simultanément ces trois objectifs. Notamment, *grâce à l'utilisation du modèle de composant Fractal* pour la conception et la composition des transformateurs, c'est le système de transformation évalué qui applique le mieux notre principe 5 d'Encapsulation et de Composabilité des Transformations.

Cependant, nous ne nous sommes pas fixé d'objectifs en termes de simplicité d'implantation des transformateurs, qui sont évalués par les critères (1.2), (1.3), (2.1), (2.2), (3.1), (4.2) et (6.1). Il apparaît que Jabyce est, avec ASM, l'un des systèmes avec lequel il est le plus difficile d'implanter des transformateurs. Notamment, Jabyce et ASM représentent les éléments de programmes par des séquences d'interactions, au lieu de graphes d'objets Java, ce qui implique une difficulté pour analyser les relations entre éléments (critère (1.3)), et offrent un balayage non déterministe des classes qui rend difficile l'implantation de transformateurs (critère (4.2)). En revanche, Javassist apparaît comme le système le plus facile à utiliser, parce qu'il offre le modèle le plus abstrait (critère (2.1)), et surtout parce qu'il est le seul système qui offre une syntaxe concrète pour la spécification de fragments de programmes (critère (6.1)). Javassist est cependant l'un des systèmes les moins généralistes.

En revanche, bien qu'elle rende plus difficile l'implantation de transformateurs, la représentation des programmes par des séquences d'interactions, comme dans Jabyce et ASM, est une représentation bien plus performante qu'une représentation en graphes d'objets, en termes de mémoire et de temps d'exécution. Ce point est montré dans le chapitre 6 des mesures de performances de transformation, qui complète les critères de comparaison qualitatifs de ce chapitre par des critères quantitatifs. Les bonnes performances de la représentation des programmes par séquences d'interactions permettent d'atteindre notre objectif / principe 6 de Composition Efficace des Transformations.

Ainsi, les deux caractéristiques principales de Jabyce, à savoir 1) la représentation de programmes par interactions, et 2) l'utilisation d'un modèle de composant général (Fractal), nous permettent d'atteindre complétement nos objectifs.

	Mécanisme de représentation interne		
	(1.1) capacité de modification des représentations d'éléments	(1.2) homogénéité des représentations de relations de composition et de dépendance entre éléments	(1.3) facilité d'implantation d'analyses de relations entre éléments
JOIE[1]	(−) représentations par objets Java peu modifiables, impossible de créer de nouvelles classes	**(+) homogènes : liaisons entre objets Java**	**(+) liaisons entre objets Java facilement navigables**
JMangler[1]	(−) seulement modifications qui conservent la compatibilité binaire	*(cf. BCEL)*	
Javassist[1]	(−) objets modifiables, mais impossibilité de supprimer des éléments	**(+) homogènes : liaisons entre objets Java**	**(+) liaisons entre objets Java facilement navigables**
ASM[2]	**(+) représentations complètement modifiables**	(−) hétérogènes : identité des objets en paramètres, et cible des appels	(−) traitements complexes nécessaires
Jabyce[2]	**(+) représentations complètement modifiables**	**(+) homogènes : identité des objets en paramètres**	(−) traitements complexes nécessaires
Serp[1]	(−) représentations par objets Java complètement modifiables, mais impossible de créer de nouvelles classes	**(+) homogènes : liaisons entre objets Java**	**(+) liaisons entre objets Java facilement navigables**
BCEL[1]	**(+) représentations par objets Java complètement modifiables**	**(+) homogènes : liaisons entre objets Java**	**(+) liaisons entre objets Java facilement navigables**

[1] systèmes offrant une représentation des programmes par des graphes d'objets Java
[2] systèmes offrant une représentation des programmes par des séquences d'interactions

TAB. 5.1 *a* : Synthèse de la comparaison qualitative des systèmes - Mécanisme de représentation interne

	Modèle conceptuel		
	(2.1) niveau d'abstraction du modèle (détails masqués du format de bytecode)	(2.2) granularité (nombre de types d'éléments) et homogénéité du modèle	(2.3) complétude des définitions de relations entre types d'éléments
JOIE	(−) modèle peu abstrait	(−) gros grain, modèle hétérogène pour les instructions	(−) seulement relations de composition et branchements entre instructions
JMangler	*(cf. BCEL)*		
Javassist	**(+) modèle le plus abstrait, masque le plus de détails**	(−) grain très gros, peu de types d'éléments représentables et transformables	(−) seulement relations de composition et branchements entre instructions
ASM	(−) modèle peu abstrait	(−) gros grain, permet de tout transformer, mais le groupement des types d'instructions est basé sur leur structure	(−) seulement relations de composition et branchements entre instructions
Jabyce	**(+) modèle très abstrait (le plus abstrait après celui de Javassist)**	**(+) grain moyen, permet de tout transformer**	**(+) le plus grand nombre de relations (relations inter-classes, etc.)**
Serp	(−) modèle peu abstrait	**(+) grain moyen, permet de tout transformer**	(−) seulement relations de composition et branchements entre instructions
BCEL	(−) modèle peu abstrait	**(+) grain le plus fin, permet de tout transformer**	(−) seulement relations de composition et branchements entre instructions

TAB. 5.1*b* : Synthèse de la comparaison qualitative des systèmes - Modèle conceptuel

	Paradigme de transformation	
	(3.1) encapsulation et configuration des transformateurs et de leur environnement	(3.2) encapsulation des implantations de transformations causalement dépendantes
JOIE	(−) objets Java, interfaces Java définies par le système	(−) problème non abordé
JMangler	(−) objets Java, interfaces Java définies par le système	(−) problème non abordé
Javassist	(−) objets Java, interfaces Java définies par le système	(−) problème non abordé
ASM	(−) objets Java, interfaces Java définies par le système	(−) problème non abordé
Jabyce	**(+) modèle de composant Fractal / Java pour les transformateurs (opérations) et leur environnement**	**(+) les héritages entre les classes abstraites et les interfaces de Jabyce *guident* l'implantation des transformations causalement dépendantes**

TAB. 5.1*c* : Synthèse de la comparaison qualitative des systèmes - Paradigme de transformation

	Stratégie de transformation		
	(4.1) flexibilité et efficacité de la composition des transformateurs	(4.2) flexibilité pour le balayage	(4.3) redondance du balayage
JOIE	(−) séquence de transformateurs à exécuter, spécifiée par programmation (pas de langage)	**(+) balayage implanté dans chaque transformateur : flexibilité maximum**	(−) balayage effectué par chaque transformateur : redondance inutile
JMangler	(−) stratégie imposée pour les transformations d' « éléments externes » ; séquence de transformateurs pour les transformations de code (langage spécifique)	**(+) balayage implanté dans chaque transformateur : flexibilité maximum**	(−) balayage effectué par chaque transformateur : redondance inutile
Javassist	(−) séquence de transformateurs (langage spécifique), mais désérialisation et sérialisation des classes pour chaque transformateur	**(+) balayage implanté dans chaque transformateur : flexibilité maximum**	(−) balayage effectué par chaque transformateur : redondance inutile
ASM	(−) séquence de transformateurs à exécuter, spécifiée par programmation, mais la chaîne des transformateurs est à reconstruire pour chaque classe	(−) balayage non déterministe	**(+) balayage effectué une fois pour une composition de transformateurs**
Jabyce	**(+) stratégie exprimée par liaisons entre composants Fractal ; langage ADL Fractal ; optimisations avec Julia**	(−) balayage non déterministe	**(+) balayage effectué une fois pour une composition de transformateurs**

TAB. 5.1*d* : Synthèse de la comparaison qualitative des systèmes - Stratégie de transformation

	Vérification des transformations		**Représentation de fragments de programmes**
	(5.1) ensemble des erreurs détectées	(5.2) précision du diagnostic des erreurs détectées	(6.1) syntaxe pour la représentation des fragments
JOIE	(−) programmation défensive : détection d'erreurs de syntaxe	**(+) localisation précise des fautes dans les transformateurs**	(−) syntaxe abstraite (« en dur »)
JMangler	*(cf. BCEL)*		(−) syntaxe abstraite (« en dur »)
Javassist	(−) programmation défensive : détection d'erreurs de syntaxe	**(+) localisation précise des fautes dans les transformateurs**	**(+) syntaxe concrète (langage source Java), pour la représentation de nouveaux blocs de code**
ASM	(−) pas de détection d'erreur	(−) localisation imprécise des fautes	(−) syntaxe abstraite (« en dur »)
Jabyce	**(+) erreurs de syntaxe et de sémantique (mais spécifications encore incomplètes)**	**(+) localisation précise des fautes dans les transformateurs**	(−) syntaxe abstraite (« en dur »)
Serp	(−) pas de détection d'erreurs	(−) localisation imprécise des fautes	
BCEL	**(+) implantation complète de l'algorithme de vérification**	(−) localisation imprécise des fautes	

TAB. 5.1 *e* **:** Synthèse de la comparaison qualitative des systèmes - Vérification des transformations et Représentation de fragments de programmes

Troisième partie

Validations expérimentales

Chapitre 6

Evaluation des performances

Ce chapitre complète le chapitre 5, qui apporte une comparaison qualitative de Jabyce et d'autres systèmes de transformation de bytecode Java, par une *comparaison quantitative* en termes de *temps d'exécution* de compositions de transformateurs.

La section 6.1 décrit nos critères d'évaluation des temps d'exécution des transformations, et justifie nos choix des systèmes de transformation qui servent de base à l'évaluation de ces critères (Serp, ASM et Jabyce). La section 6.2 décrit la transformation de programme qui est implantée avec les différents systèmes, et décrit en détails ses implantations. La section 6.3 décrit l'environnement utilisé pour effectuer les mesures. Cet environnement est conçu et implanté en utilisant le modèle de composant Fractal, et exécute les transformateurs dont les implantations sont décrites dans la section 6.2. Les résultats de ces mesures sont donnés et analysés dans la section 6.4. La section 6.5 donne une synthèse de ces analyses.

6.1 Approche générale

Critères d'évaluation de performances

Nous considérons deux critères d'évaluation de performances :

1. le temps de transformation de chaque classe, par chaque transformateur dans une configuration, doit être minimisé de manière à pouvoir composer efficacement de nombreux transformateurs dans une même configuration ;
2. le temps de désérialisation et de sérialisation d'une classe transformée doit être minimisé.

Il est difficile de satisfaire ces deux critères simultanément. En effet, le concepteur d'un système de transformation doit faire un compromis entre le niveau d'abstraction du modèle conceptuel, et les temps de traitement dans chaque transformateur. Comme décrit dans la section 5.4, un modèle conceptuel à un faible niveau d'abstraction, c'est-à-dire qui cache peu de détails du format sérialisé des classes compilées Java, impose des traitements supplémentaires dans chaque transformateur par rapport à un modèle conceptuel à un haut niveau d'abstraction. Par exemple, un modèle conceptuel qui expose les *opcodes* (identifiants d'instructions), permet d'éviter aux (dé)sérialiseurs de décoder et encoder ces *opcodes*, mais impose à chaque transformateur

d'instructions de décoder les opcodes. Ceci peut se traduire par des traitements redondants dans les transformateurs et une augmentation des temps de transformation avec une composition de nombreux transformateurs.

Nous privilégions le critère de **la minimisation du temps de transformation par chaque transformateur**, pour juger de l'application de notre principe 6 de Composition Efficace des Transformateurs.

D'autre part, le concepteur d'un système de transformation doit faire un compromis entre la généralité du système, et les performances de transformation. En effet, il est toujours envisageable de concevoir un système de transformation limité à une certaine catégorie de transformations, par exemple le brouillage (*obfuscation*), et optimisé pour cette catégorie. Cependant, en application de notre principe 4 de Généralité du Système de Transformation, nous avons considéré dans notre comparaison seulement des systèmes de transformation généralistes (cf. la section 5.2).

Caractéristiques qualitatives prises en compte

L'objectif de ce chapitre est de montrer qu'avec ses caractéristiques, Jabyce permet d'obtenir des bonnes performances lors de l'exécution de transformateurs. Plus précisément, ce chapitre montre l'influence des choix dans les dimensions suivantes, sur nos critères de mesure de performances définis ci-dessus :

1. mécanisme de représentation interne : représentation des programmes par des séquences d'interactions, *vs.* représentation par des graphes d'objets (cf. la section 5.3) ;
2. paradigme de transformation : choix d'un modèle de composant général tel que Fractal, *vs.* transformateurs implantés directement en Java (cf. la section 5.5) ;
3. modèle conceptuel : choix d'un modèle conceptuel général qui abstrait de nombreux détails du format sérialisé des classes dans la plupart des systèmes, *vs.* modèle peu abstrait (cf. la section 5.4) ;
4. stratégie de transformation : stratégie de balayage des classes non déterministe mais non redondante, *vs.* stratégie de balayage spécifique à chaque transformateur mais redondante (cf. la section 5.6).

Pour l'évaluation du point 1, nous comparons Serp à Jabyce, parce que Serp est représentatif des systèmes de transformation de bytecode qui représentent les classes par des graphes d'objets. De plus, Serp a le modèle conceptuel le plus proche de Jabyce parmi les systèmes considérés, comme décrit dans la section 5.4, ce qui est essentiel pour la validité de cette évaluation. Pour l'évaluation du point 3, nous comparons ASM et Jabyce, parce que comme expliqué à la fin de la section 5.4, Jabyce offre le modèle conceptuel le plus abstrait, et ASM un des modèles les moins abstraits. Par exemple, ASM expose directement les *opcodes* dans la représentation interne des instructions. De même, comme détaillé dans la section 5.5, ASM offre l'un des paradigmes de transformation les plus simples (les transformateurs sont des objets Java qui implantent des interfaces bien définies) et les plus représentatifs de ceux des autres systèmes, et est donc bien adapté pour l'évaluation du point 2. Pour l'évaluation du point 4, nous comparons Jabyce et ASM, qui offrent un balayage unique mais non déterministe de

chaque classe pour une configuration de transformateurs, avec Serp qui offre comme les autres systèmes un balayage implanté dans chaque transformateur.

Il est à noter que Serp n'est pas un système de transformation à proprement parler, comme indiqué dans la section 5.2. Cependant, pour les expérimentations effectuées dans ce chapitre, nous avons développé un système de tranformation minimal basé sur Serp, très similaire à JMangler qui est basé sur BCEL. Nous avons défini l'interface SerpClassTransformer, qui doit être implantée par les objets transformateur, et qui est donnée dans le programme 6.1. La méthode transformClass(...) prend en paramètre la représentation interne d'une classe à transformer (un objet de la classe BCClass de Serp), et retourne l'ensemble des représentations de classes résultant de cette transformation.

```
public interface SerpClassTransformer {
  /**
   * Transforms the Java class as the specified BCClass,
   * to produce a set of new Java classes.
   * @param project allows for the creation of new classes.
   * @param javaClass the Java class to transform.
   * @return the new Java classes.
   */
  BCClass[] transformClass(Project project, BCClass javaClass);
}
```

Prog. 6.1 : Interface SerpClassTransformer

6.2 Transformation pour la trace de retour de méthode

Nous considérons une seule transformation de programme, et qui est implantée à la fois dans ASM, Jabyce et Serp, pour comparer ces trois implantations. La transformation que nous avons implantée est l'insertion d'instructions de trace avant chaque instruction *x*return de retour de méthode (c'est-à-dire chaque instruction ireturn, lreturn, freturn, dreturn, areturn ou return), dans toutes les méthodes des classes transformées. Cette transformation est similaire à la transformation de documents XML décrite dans la section 3.2.4. D'un point de vue général, cette transformation transforme chaque instruction *x*return en une séquence d'instructions d'affichage d'un message de trace suivie de l'instruction *x*return originale. Le code qui affiche la trace d'exécution est la séquence d'instructions de bytecode donnée dans le programme 6.2. Par exemple, le résultat de la transformation de la classe compilée dont le code source est donné dans le programme 6.3, est une classe compilée qui est équivalente à la classe compilée dont le code source est donné dans le programme 6.4. Les mots-clé return sont explicites dans le code source de ces programmes, bien que ce soit optionnel dans le langage Java, mais illustrent ici l'emplacement des instructions de bytecode return générées par un compilateur Java.

Le choix de cette transformation particulière est motivé par les propriétés dont nous évaluons l'impact sur les performances, identifiées dans la section précédente. Ainsi, l'insertion d'une trace avant chaque instruction *x*return, et non pas par exemple en début de méthode, impose de balayer toutes les implantations de méthodes, pour trouver les instructions *x*return à transformer. Cette transformation nous permet donc d'évaluer l'efficacité du balayage dans

1. getstatic : empile la référence d'objet stockée dans l'attribut statique out défini dans la classe java.lang.System, de type java.io.PrintStream ;
2. ldc : empile la chaîne du message à afficher, identifiée par l'index de la chaîne dans la table des constantes (*constant pool*) ;
3. invokevirtual : appelle la méthode println(String) définie dans java.io.PrintStream, avec pour objet cible et pour argument les références d'objets qui ont été empilées par les instructions exécutées précédemment.

PROG. 6.2 : La séquence d'instructions de bytecode pour afficher une trace

```
public class BonjourMonde {
  public void afficheBonjour() {
    System.out.println("bonjour");
    return;
  }
  public void afficheMonde() {
    System.out.println(" monde");
    return;
  }
  public static void main(String[] argv) {
    BonjourMonde obj = new BonjourMonde();
    obj.afficheBonjour();
    obj.afficheMonde();
    return;
  }
}
```

PROG. 6.3 : Code source original de la classe BonjourMonde à transformer

```
public class BonjourMonde {
  public void afficheBonjour() {
    System.out.println("bonjour");
    System.out.println(
      "retour de afficheBonjour dans BonjourMonde");
    return;
  }
  public void afficheMonde() {
    System.out.println(" monde");
    System.out.println(
      "retour de afficheMonde dans BonjourMonde");
    return;
  }
  public static void main(String[] argv) {
    BonjourMonde obj = new BonjourMonde();
    obj.afficheBonjour();
    obj.afficheMonde();
    System.out.println(
      "retour de main dans BonjourMonde");
    return;
  }
}
```

Prog. 6.4 : Code source équivalent à celui de la classe compilée BonjourMonde transformée

les différents systèmes. De plus, le choix d'une transformation des instructions de bytecode permet d'évaluer le niveau d'abstraction des systèmes de transformation, et notamment leur traitement des *opcodes* (identifiants de types d'instructions). En effet, il existe six opcodes différents pour les instructions de retour, suivant le type de retour de la méthode : ireturn, lreturn, freturn, dreturn, areturn, return. Cette transformation nous permet d'évaluer le coût des traitements de telles instructions dans les transformateurs.

Implantation avec Serp

Dans le système de transformation minimal que nous avons conçu pour Serp, la méthode transformClass(...) est appelée pour chaque classe sur l'objet transformateur, avec en paramètre l'objet BCClass qui représent la classe à transformer. Cette méthode retourne l'ensemble des objets BCClass qui résultent de la transformation, éventuellement vide si la classe doit être supprimée. Le code source du transformateur dans notre système de transformation minimal basé sur Serp est donné dans le programme 6.5. Dans cette implantation de la méthode transformClass(...), la représentation de la classe est balayée par un algorithme itératif, avec une boucle for sur les représentations des méthodes implantées dans la classe, et une deuxième boucle while sur les instructions. Pour chaque représentation d'instruction *x*return (objets de type ReturnInstruction), des objets représentant les instructions de trace sont créés et insérés dans la liste des objets représentant les instructions. La méthode retourne l'objet BCClass représentant la classe balayée et transformée.

```
public class ReturnTraceSerpClassTransformer
    implements SerpClassTransformer {

  public BCClass[] transformClass(Project project,
                                  BCClass javaClass) {
    String className = javaClass.getName();

    /* balayage pour transformer toutes les méthodes */
    BCMethod[] methods = javaClass.getDeclaredMethods();
    for (int i = 0; i < methods.length; i++) {

      BCMethod method = methods[i];
      String methodName = method.getName();
      Code code = method.getCode(false);
      if (code != null) {
        String messageToPrint = "retour de " + methodName
          + " dans " + className;

        /* balayage des instructions dans la méthode */
        while (code.hasNext()) {
          Instruction in = code.next();

          /* transformer si l'instruction est un xreturn */
          if (in instanceof ReturnInstruction) {
            /* insertion d'instructions avant le return */
            code.previous();

            /* instruction 1 :  getstatic */
            code.getfield().setField("java/lang/System",
              "out", "Ljava/io/PrintStream;");
            /* instruction 2 :  ldc*/
            code.constant().setValue(messageToPrint);
            /* instruction 3 :  invokevirtual */
            code.invokevirtual().setMethod(
              "java/io/PrintStream", "println", "V",
              new String[] { "Ljava/lang/String;" } );
            code.next();
    } } } }
    return new BCClass[] { javaClass };
  }
}
```

Prog. 6.5 : Classe ReturnTraceSerpClassTransformer

Implantation avec ASM

Dans ASM, un transformateur de classes Java est un objet décorateur d'un objet implantant l'interface ClassVisitor (décrite dans les sections 5.3 et 5.4), selon le patron de conception Décorateur (cf. l'annexe B.2). Le transformateur ASM le plus simple, qui n'effectue aucune transformation, est un objet implantant l'interface ClassVisitor et qui retransmet tous les appels de méthode reçus vers l'objet ClassVisitor « décoré ». L'implantation de ce transformateur minimal est fournie dans ASM, dans la classe ClassAdapter. Tout transformateur ASM est généralement implanté en étendant la classe ClassAdapter, et en redéfinissant les méthodes pour effectuer des transformations. De même, un transformateur de méthode est un objet Java implantant l'interface CodeVisitor, qui décore un objet du même type. ASM fournit une classe CodeAdapter qui sert de base à l'implantation de tels transformateurs.

Notre transformateur de classe est implanté dans la classe ReturnTraceASMClassAdapter, donnée dans le programme 6.6, qui étend ClassAdapter. La principale méthode redéfinie dans cette classe est la méthode visitMethod(...), pour retourner un objet décorateur du type ReturnTraceASMCodeAdapter, qui transforme les instructions dans les méthodes. Notre classe ReturnTraceASMCodeAdapter, donnée dans le programme 6.7, étend CodeAdapter. Cette classe redéfinit la méthode visit(...) pour transformer les instructions *x*return. Comme décrit dans la section 5.4, la méthode visit(...) est appelée pour représenter des instructions de types très différents, y compris les instructions *x*return. Il est donc nécessaire de tester la valeur de l'*opcode* (identifiant de type d'instruction) passé en paramètre. Les instructions insérées avant chaque instruction *x*return sont représentées par des appels de méthodes sur l'objet CodeVisitor décoré par le transformateur (appels « super.visit*X*(...) »).

```
public class ReturnTraceASMClassAdapter extends ClassAdapter {
  private String className;
  ReturnTraceASMClassAdapter(ClassVisitor classVisitor) {
    super(classVisitor);
  }
  public void visit(int access, String name,
      String superName, String[] interfaces,
      String sourceFile) {
    super.visit(access, name, superName, interfaces,
                sourceFile);
    className = name.replace('/', '.');
  }

  public CodeVisitor visitMethod(int access, String name,
      String desc, String[] exceptions, Attribute attrs) {

    /* appelle le prochain transformateur de classe
       dans la chaîne */
    CodeVisitor codeVisitor = super.visitMethod(access,
      name, desc, exceptions, attrs);

    if (codeVisitor != null) {
      /* retourne un transformateur de code, qui insère le
         code de trace */
      return new ReturnTraceASMCodeAdapter(codeVisitor,
        className, name);
    } else {
      return codeVisitor;
    }
  }
}
```

Prog. 6.6 : Classe ReturnTraceASMClassAdapter

```
public class ReturnTraceASMCodeAdapter
    extends CodeAdapter {
  private String className;
  private String methodName;

  ReturnTraceASMCodeAdapter(CodeVisitor codeVisitor,
      String className, String methodName) {
    super(codeVisitor);
    this.className = className;
    this.methodName = methodName;
  }

  public void visitInsn(int opcode) {

    if ((opcode == Constants.IRETURN)
        || (opcode == Constants.LRETURN)
        || (opcode == Constants.FRETURN)
        || (opcode == Constants.DRETURN)
        || (opcode == Constants.ARETURN)
        || (opcode == Constants.RETURN)) {

      String messageToPrint = "retour de " + methodName
        + " dans " + className;
      /* instruction 1 :  getstatic */
      super.visitFieldInsn(Constants.GETSTATIC,
        "java/lang/System", "out",
        "Ljava/io/PrintStream;");
      /* instruction 2 :  ldc */
      super.visitLdcInsn(messageToPrint);
      /* instruction 3 :  invokevirtual */
      super.visitMethodInsn(Constants.INVOKEVIRTUAL,
        "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
    }

    /* re-représentation du xreturn transformé */
    super.visitInsn(opcode);
  }

/*...*/
}
```

Prog. 6.7 : Classe ReturnTraceASMCodeAdapter

Implantation avec Jabyce

Dans Jabyce, le transformateur transforme les instructions de retour de méthode. C'est un composant Fractal qui expose donc une interface serveur de type ReturnInstructionFactory. Il doit pouvoir représenter des instructions de retour (pour reproduire à l'identique les instructions de retour « transformées »), des instructions d'accès aux attributs (getstatic), des instructions d'empilement de chaînes constantes (ldc), et des instructions d'appels de méthodes (invokevirtual). Notre composant transformateur expose donc quatre interfaces client de type ReturnInstructionFactory, FieldAccessInstructionFactory, StringOrNullConstantPushInstructionFactory, et MethodCallInstructionFactory. Il expose également des interfaces client de type MethodFactory et ClassFactory, pour pouvoir récupérer le nom des méthodes et des classes associés aux objets de contexte de représentation des méthodes et classes (*X*BuildingContext). Un tel composant transformateur est illustré dans la figure 6.1. Il peut être intégré dans un composant opération, comme illustré dans la figure 6.2, en liant ses interfaces aux interfaces internes de l'opération. Seule l'interface serveur de type ReturnInstructionFactory de l'opération est liée à l'interface serveur de même type sur le transformateur, et toutes les autres interfaces serveur sont directement liées aux interfaces client internes correspondantes. Les interfaces client du transformateur sont liées aux interfaces serveur internes correspondantes de l'opération.

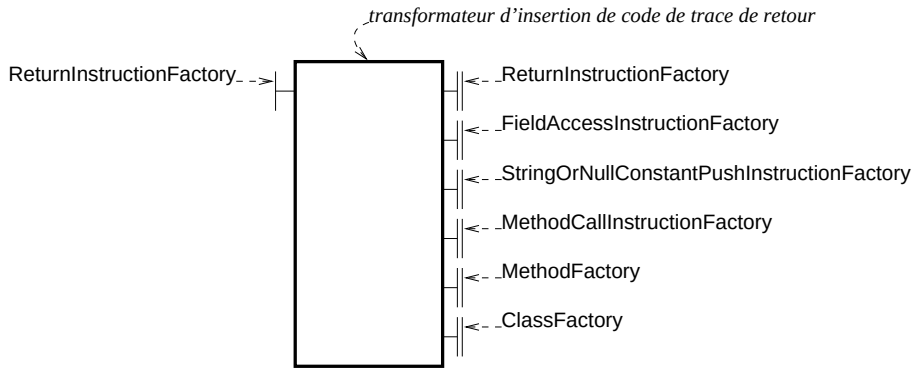


FIG. 6.1 : Transformateur Jabyce d'insertion de code de trace de retour

Le composant transformateur est un composant primitif. Son implantation est donc une classe Java, comme spécifié dans Julia, l'implantation de référence de Fractal en Java. Le code source de la classe du transformateur est donné dans les programmes 6.8*a* et 6.8*b*. Cette classe ReturnTraceTransformer implante l'interface ReturnInstructionFactory, car le composant offre une interface serveur de ce type. L'interface BindingController est définie dans le modèle de composant Fractal. Cette interface définit des méthodes appelées par le moteur d'exécution de Julia pour passer à un composant les références des composants liés sur ses interfaces client. Implanter cette interface est la seule contrainte imposée par le modèle Fractal pour l'implantation de composants primitifs en Java.

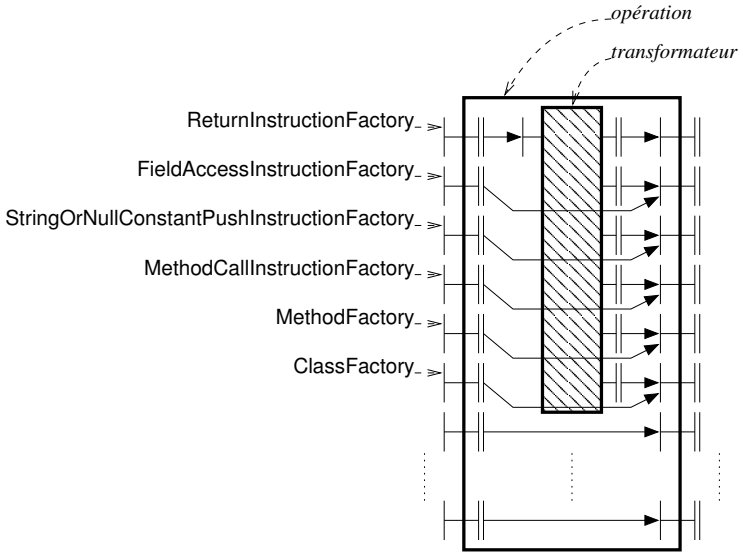


FIG. 6.2 : Opération Jabyce d'insertion de trace de retour

```
public class ReturnTraceTransformer
  implements BindingController, ReturnInstructionFactory {

  /* références vers les composants liés
     sur les interfaces client */
  protected ClassFactory boundClassFactory;
  protected MethodFactory boundMethodFactory;
  protected ReturnInstructionFactory
    boundReturnInstructionFactory;
  protected FieldAccessInstructionFactory
    boundFieldAccessInstructionFactory;
  protected StringOrNullConstantPushInstructionFactory
    boundStringOrNullConstantPushInstructionFactory;
  protected MethodCallInstructionFactory
    boundMethodCallInstructionFactory;

  /* méthodes de BindingController */
  public String[] listFc() /*...*/
  public Object lookupFc(String clientItfName) /*...*/
  public void bindFc(String clientItfName,
                     Object serverItf) /*...*/
  public void unbindFc(final String clientItfName) /*...*/

  /* un appel représente une instruction de retour dans
     la représentation de méthode associée au contexte */
  public void createReturnInstruction(
    MethodBuildingContext compositeMethodBuildingContext) {

    /* récupère le contexte identifiant la représentation
       de la classe */
    ClassBuildingContext classBuildingContext =
      boundMethodFactory.getCompositeClass(
        compositeMethodBuildingContext);

    String className =
      boundClassFactory.getBuiltClassClassType(
        classBuildingContext).toString();
    String methodName = boundMethodFactory
      .getBuiltMethodName(compositeMethodBuildingContext);
```

Prog. 6.8*a* : Classe ReturnTraceTransformer implantant le transformateur Jabyce

```
    /* construit le message à afficher */
    String messageToPrint = "retour de " + methodName
      + " dans " + className;

    /* représentation des nouvelles instructions */
    /* instruction 1 :  getstatic */
    boundFieldAccessInstructionFactory
      .createFieldAccessInstruction(
        compositeMethodBuildingContext,
        new ObjectType("java/lang/System",true),"out",true,
        new ObjectType("java/io/PrintStream",true), true);

    /* instruction 2 :  ldc */
    boundStringOrNullConstantPushInstructionFactory
      .createStringOrNullConstantPushInstruction(
        compositeMethodBuildingContext, messageToPrint);

    /* instruction 3 :  invokevirtual */
    boundMethodCallInstructionFactory
      .createMethodCallInstruction(
        compositeMethodBuildingContext,
        new ObjectType("java/io/PrintStream", true), false,
        "println", false, false, VoidType.VOID,
        new FieldDescriptor[] {
          new ObjectType("java/lang/String", true)});

    /* représentation de l'instruction de retour
       originale */
    boundReturnInstructionFactory.createReturnInstruction(
      compositeMethodBuildingContext);
  }
}
```

PROG. 6.8*b* : Classe ReturnTraceTransformer implantant le transformateur Jabyce

6.3 Architecture

Nous avons conçu un seul système de test qui est utilisé pour déclencher et mesurer les transformations implantées avec tous les systèmes de transformation. Les transformations avec Jabyce, ASM et Serp sont réalisées dans une même exécution de ce système, pour évaluer de manière homogène les temps de transformation. Les mêmes mécanismes de Jabyce sont utilisés pour lire et écrire les classes compilées, pour les transformations avec tous les systèmes, au lieu d'utiliser les mécanismes propres à ASM et Serp, afin de rendre constants les temps de lecture / écriture, et de pouvoir plus facilement comparer les temps de (dé)sérialisation et de transformation avec les différents systèmes.

L'architecture du système exécuté pour les mesures est illustrée dans la figure 6.3. Cette architecture est celle d'un programme de transformation Jabyce typique. Selon les conventions des outils de Fractal, le système est globalement un composant composite offrant une seule interface serveur, de type Runnable. Au démarrage du système, tous les composants sont instanciés automatiquement et récursivement, puis la méthode run() est appelée sur l'interface de type Runnable du composant composite global. Cette interface est implantée par un sous-composant *directeur*, qui dirige l'exécution des transformations. Le système comprend également deux composants *conteneurs*, « d'entrée » et « de sortie », qui contiennent respectivement les classes Java compilées à transformer, et les classes résultant de la transformation. Le système comprend également une chaîne composée d'un composant désérialiseur, de plusieurs composants opération de transformation, et un sérialiseur. Le composant sérialiseur est lié au composant conteneur de sortie, pour que les classes sérialisées soient directement stockées dans ce conteneur.

Lorsque la méthode run() est appelée sur le composant directeur, celui-ci 1) enlève toutes les classes du conteneur de sortie, et 2) itère sur les classes contenues dans le composant conteneur d'entrée, et pour chaque classe la transmet sous sa forme sérialisée au désérialiseur. Le désérialiseur désérialise chaque classe et produit une représentation interne de la classe, qui est transformée successivement par les T composants opération, jusqu'au sérialiseur qui sérialise la classe transformée et la transmet sous forme sérialisée au composant adaptateur qui la stocke dans le conteneur de sortie.

La principale différence entre ce système de test et une application purement basée sur Jabyce, est que le composant directeur effectue également des transformations en utilisant des compositions de T transformateurs ASM, et respectivement T transformateurs Serp, après avoir effectué les transformations avec Jabyce, dans la même exécution du système. Le composant directeur est paramétré avec le nombre T de transformateurs à composer. Pour effectuer les transformations avec ASM et Serp, le composant directeur lit et écrit les classes compilées avant et après transformation, directement en interagissant avec les composants conteneur d'entrée et de sortie.

Dans le contexte de cet exemple, les T *composants opération Jabyce sont tous identiques* dans la configuration, et correspondent à l'opération illustrée dans la figure 6.2. De même, pour comparer les performances des systèmes, le composant directeur effectue également pour chaque classe des transformations en utilisant des chaînes de T *transformateurs identiques* développés respectivement avec ASM et Serp. Ces transformateurs sont implantés par les classes de transformateurs décrites dans la section précédente.

Il est à noter que les transformateurs sont composés sans modifier le code des transformateurs,

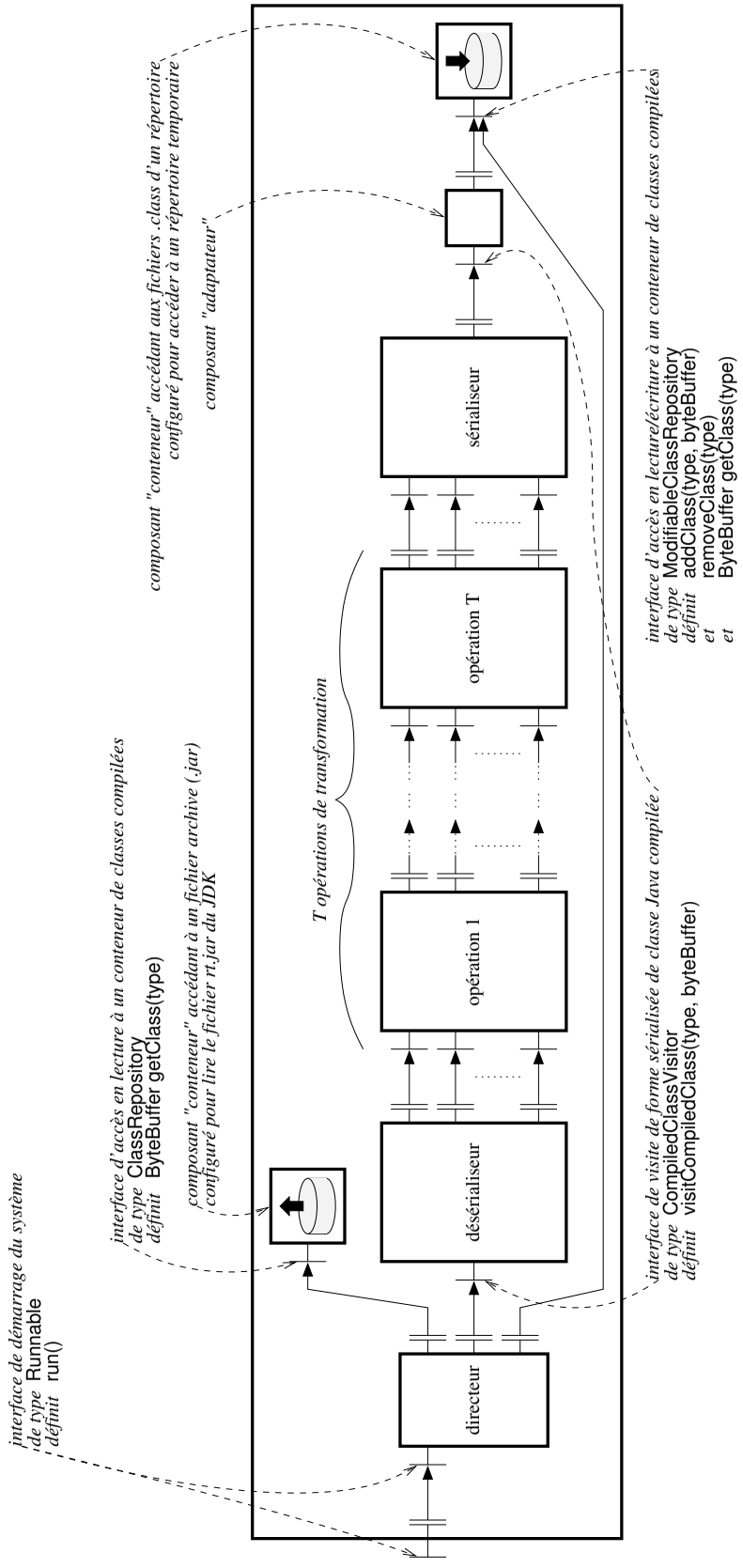


FIG. 6.3 : Architecture du système de test

quelque soit le système. En effet, en pratique, une implantation de transformation est trop complexe pour être modifiée par un utilisateur.

6.4 Mesures et interprétations

La figure 6.4 indique les temps de transformation (en ms) mesurés pour transformer les 9229 classes du JDK (*Java Development Kit*) version 1.4.2 de Sun Microsystems, en fonction du nombre variable de transformateurs T, respectivement pour Serp, ASM et Jabyce. Ces classes contiennent au total $R = 86606$ instructions *x*return. Les mesures ont été effectuées sur un ordinateur dédié à ces mesures, sous le système Debian GNU/Linux, et doté d'un processeur Pentium Xeon 2.8 GHz et de 2 Go de mémoire RAM, en utilisant le JDK version 1.4.2 de Sun Microsystems. Chaque point de mesure est la moyenne de 10 mesures. Nous n'avons pas mesuré de valeurs aberrantes, car l'écart-type calculé pour chaque série de 10 mesures est inférieur à 4% des valeurs mesurées. Le reste de cette section est une analyse de ces mesures.

Dans notre transformation d'exemple, chaque transformateur ajoute trois instructions pour chaque instruction *x*return dans une méthode. Dans Serp et ASM, les représentations d'instructions ajoutées par un transformateur sont balayées / interceptées par chaque transformateur dans la suite de la chaîne. Dans un transformateur Serp, cela augmente le temps d'exécution de la boucle while d'itération sur les instructions d'une méthode (cf. le programme 6.5). Dans un transformateur ASM, ces instructions sont représentées par de nouveaux appels de méthodes sur l'objet ReturnTraceCodeAdapter, qui sont réémis, sans transformation, à l'objet transformateur suivant dans la chaîne (cf. le programme 6.7). Ces traitements supplémentaires, dans les transformateurs Serp et ASM, deviennent non négligeables à l'échelle des $R = 86606$ instructions *x*return présentes dans les 9229 classes transformées dans cet exemple. Par exemple, pour $T = 100$ transformateurs présents dans la chaîne, $T \times R \times 3 = 25981800$ instructions sont ajoutées. En revanche, dans une opération Jabyce, les instructions ajoutées sont représentées par des appels sur les interfaces de type FieldAccessInstructionFactory, StringOrNullConstantPushInstructionFactory et MethodCallInstructionFactory. Or, comme illustré dans la figure 6.2, l'opération Jabyce qui effectue cette transformation n'intercepte pas les interactions sur ces interfaces. Grâce à l'application du principe 7 de Séparation des Interfaces permettant cette architecture interne « creuse » des composants opération, et aux optimisations de Julia (cf. la section 3.2.6), ces interactions sont directement reçues par le composant sérialiseur. Il n'y a aucun coût dû à des indirections dans les transformateurs, comme avec ASM. L'utilisation du modèle de composant Fractal n'a donc aucun impact négatif sur les performances. En conséquence, avec Jabyce, *le temps de transformation augmente linéairement* avec le nombre T d'opérations de transformation, contrairement aux autres systèmes avec lesquels ce temps augmente de manière quadratique. Dans tous les systèmes, le coût de la sérialisation des classes augmente avec le nombre de transformateurs, à cause de l'insertion de nouvelles instructions à sérialiser.

Formules globales du temps d'exécution

La formule des courbes de temps total d'exécution des transformateurs, pour tous les systèmes, est donnée dans la formule (6.1) :

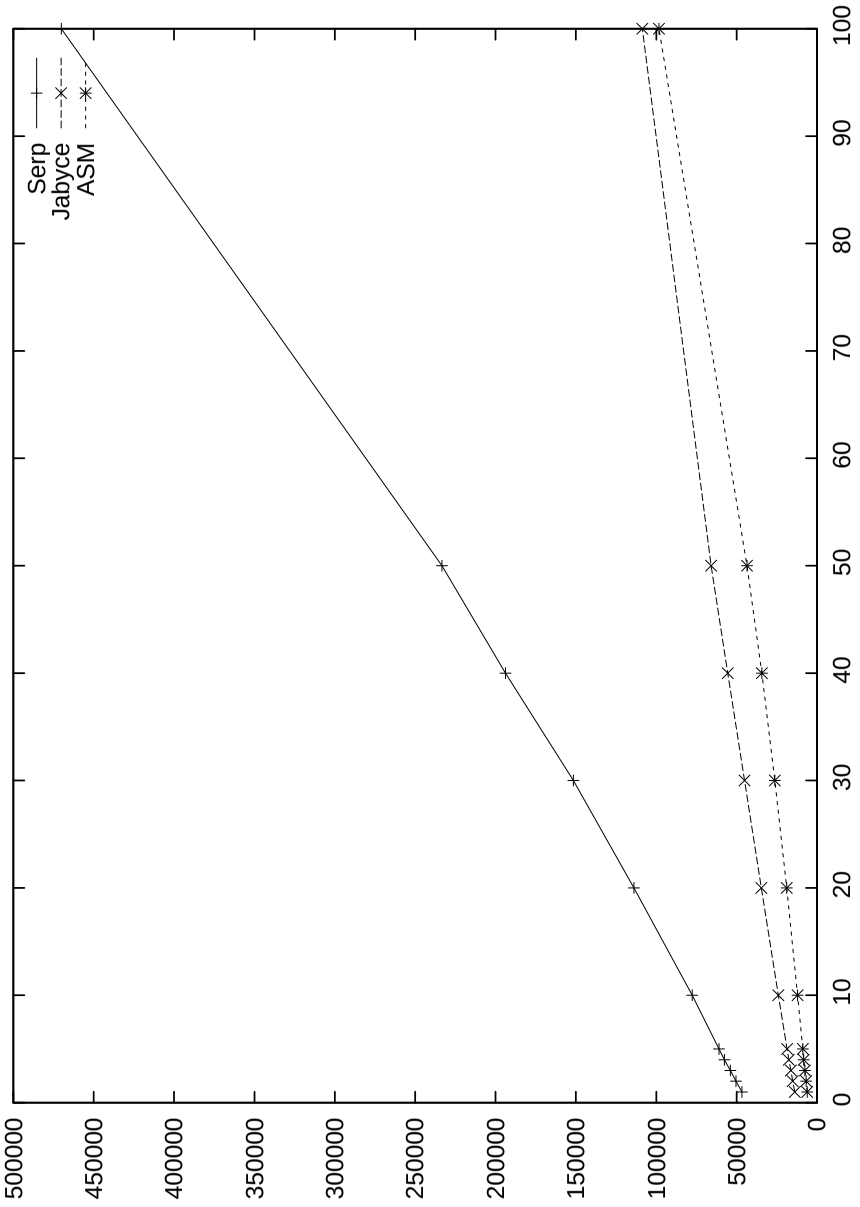


FIG. 6.4 : Temps de transformation (ms) mesurés en fonction du nombre de transformateurs

$$C(T,R) = Cld + Ct(T,R) + Cse(T,R) \tag{6.1}$$

où T est le nombre de transformateurs utilisés pour transformer les classes, R est le nombre d'instructions xreturn dans l'ensemble des classes, $C(T,R)$ est le temps total d'accès à l'ensemble des classes sérialisées et de désérialisation, transformation et sérialisation de ces classes, Cld est le temps de lecture et de désérialisation de toutes les classes, $Ct(T,R)$ est le temps d'exécution de T transformateurs sur l'ensemble des classes contenant R instructions xreturn, et $Cse(T,R)$ est le temps de sérialisation et d'écriture de l'ensemble des classes. Cld est une valeur constante, parce que les classes sont identiques à chaque lecture et désérialisation. La valeur de Cld est spécifique à chaque implantation de désérialiseur dans Jabyce, ASM et Serp. En revanche, le temps de sérialisation augmente avec le nombre T de transformateurs et le nombre R d'instructions xreturn dans les classes, parce que chaque transformateur ajoute 3 instructions pour chaque instruction xreturn. La formule du temps de sérialisation et d'écriture des classes est donné dans la formule (6.2) :

$$Cse(T,R) = Cse_0 + 3 \cdot Csei \cdot R \cdot T \tag{6.2}$$

où Cse_0 est le temps de sérialisation et d'écriture de l'ensemble des classes non transformées. $Csei$ le temps de sérialisation et d'écriture d'une seule instruction ajoutée par un transformateur, que nous supposons constant pour les trois types d'instructions ajoutés (getstatic, ldc et invokevirtual). Les valeurs de Cse_0 et $Csei$ sont spécifiques à chaque implantation de sérialiseur.

Formules du temps de tranformation pour Jabyce

$Ct_{jabyce}(T,R)$ est donné dans la formule (6.3) pour Jabyce :

$$Ct_{jabyce}(T,R) = Ctr_{jabyce} \cdot R \cdot T \tag{6.3}$$

où Ctr_{jabyce} est le temps constant de transformation d'une instruction xreturn par un seul transformateur Jabyce. La formule (6.4) est la formule globale du temps de lecture, désérialisation, transformation, sérialisation et écriture des classes compilées, pour Jabyce :

$$C_{jabyce}(T,R) = (Ctr_{jabyce} + 3 \cdot Csei_{jabyce}) \cdot R \cdot T + Cld_{jabyce} + Cse_{0jabyce} \tag{6.4}$$

Cette formule est celle d'une droite, comme observé dans la figure 6.4.

Formules du temps de transformation pour ASM et Serp

Pour ASM et Serp, le temps de transformation de l'ensemble des classes par un seul transformateur n'est pas seulement fonction du nombre R d'instructions xreturn, parce que les autres éléments des classes sont également balayés (pour Serp) ou représentées par des appels qui sont interceptés par chaque transformateur (pour ASM). Le temps de transformation de l'ensemble des classes *non encore transformées* (« initiales ») par *un seul transformateur* dans une chaîne

est donc donné dans la formule (6.5) pour ASM (une formule $Ctinit_{serp}$ similaire est définie pour Serp), qui comprend un terme constant :

$$Ctinit_{asm}(R) = Ctsansr_{asm} + R \cdot Ctr_{asm} \tag{6.5}$$

$Ctsansr_{asm}$ est la constante du temps de transformation / balayage de l'ensemble des classes non encore transformées, par un seul transformateur dans la chaîne, à l'exclusion des instructions *x*return. Ctr_{asm} est le temps moyen de transformation d'une instruction *x*return par un transformateur (dans ASM, le temps d'un appel à visitInsn(...) pour une instruction *x*return). Les temps de transformation pour plusieurs transformateurs sont calculés par la formule récursive (6.6) :

$$\begin{aligned} Ct_{asm}(0, R) &= 0 \\ Ct_{asm}(T+1, R) &= Ct_{asm}(T, R) + Ctinit_{asm}(R) + 3 \cdot R \cdot Cti_{asm} \cdot T \end{aligned} \tag{6.6}$$

où Cti_{asm} est le temps pour un transformateur pour exécuter une indirection pour un appel représentant une instruction ajoutée par un transformateur placé avant dans la chaîne. Une formule similaire $Ct_{serp}(T, R)$ est définie pour Serp. Après simplification, cette formule devient la formule (6.7) :

$$Ct_{asm}(T, R) = \frac{3 \cdot R \cdot Cti_{asm}}{2} \cdot T^2 + (Ctinit_{asm}(R) - \frac{3 \cdot R \cdot Cti_{asm}}{2}) \cdot T \tag{6.7}$$

La formule (6.8) est la formule globale du temps de lecture, désérialisation, transformation, sérialisation et écriture des classes compilées, pour ASM. La formule (6.9) est la formule équivalente pour Serp.

$$\begin{aligned} C_{asm}(T, R) &= \frac{3 \cdot R \cdot Cti_{asm}}{2} \cdot T^2 \\ &\quad + (Ctinit_{asm}(R) - \frac{3 \cdot R \cdot Cti_{asm}}{2} + 3 \cdot Csei_{asm} \cdot R) \cdot T \\ &\quad + Cld_{asm} + Cse_{0asm} \end{aligned} \tag{6.8}$$

$$\begin{aligned} C_{serp}(T, R) &= \frac{3 \cdot R \cdot Cti_{serp}}{2} \cdot T^2 \\ &\quad + (Ctinit_{serp}(R) - \frac{3 \cdot R \cdot Cti_{serp}}{2} + 3 \cdot Csei_{serp} \cdot R) \cdot T \\ &\quad + Cld_{serp} + Cse_{0serp} \end{aligned} \tag{6.9}$$

Ces formules sont celles de paraboles, comme observé dans la figure 6.4.

Interpolation avec les mesures

La formule (6.4) pour Jabyce est celle d'une droite, et les formules (6.8) pour ASM et (6.9) pour Serp sont celles de paraboles. Par interpolation avec la méthodes des moindres carrés sur les temps mesurés et représentés dans la figure 6.4, nous avons calculé les formules (6.10) pour Jabyce, (6.11) pour ASM, et (6.12) pour Serp, avec une erreur inférieure à 3% pour ces calculs d'approximations (les valeurs sont en ms) :

$$C_{jabyce}(T, 86606) = 972,949 \cdot T + 14305,9 \tag{6.10}$$

$$C_{asm}(T, 86606) = 3,4482 \cdot T^2 + 581,109 \cdot T + 5738,31 \tag{6.11}$$

$$C_{serp}(T, 86606) = 9,17275 \cdot T^2 + 3348,05 \cdot T + 43638,7 \tag{6.12}$$

Interprétation

Les valeurs constantes de ces formules correspondent aux temps $Cld + Cse_0$ de lecture, de (dé)sérialisation et d'écriture des classes. Nous observons que ces valeurs varient fortement suivant les systèmes. Les temps de lecture et d'écriture des classes sont constants quelque soit le système, de par l'architecture de notre système de test (cf. la section 6.3). Nous avons mesuré un temps de lecture et d'écriture de l'ensemble des classes, sans (dé)sérialisation ou transformation, constant et égal à $Cle = 1621$ ms en moyenne. Les variations de $Cld + Cse_0$ suivant les systèmes sont exclusivement dûes à des variations des temps de désérialisation et sérialisation des classes, et donc à la performance des (dé)sérialiseurs. ASM offre les implantations de (dé)sérialiseurs les plus performantes, parce que le modèle conceptuel de ASM est celui qui cache le moins de détails du format des classes compilées (p.ex. les *opcodes* des instructions ne sont pas décodés), ce qui permet de minimiser les traitements dans les (dé)sérialiseurs. En revanche, le modèle conceptuel de Jabyce est l'un des modèles qui masque le plus les détails du format des classes compilées (cf. la section 5.4), ce qui implique des traitements supplémentaires dans les (dé)sérialiseurs. De plus, les implantations des composants (dé)sérialiseurs dans Jabyce s'appuient sur ceux de ASM, et sont donc naturellement moins performants que ceux de ASM, ce qui est vérifié par nos mesures. Les performances de (dé)sérialisation de Serp sont les moins bonnes. Nous supposons que ce surcoût (rapport 4 par rapport à Jabyce) est essentiellement dû à l'utilisation d'une représentation des éléments de programmes par des objets Java : le désérialiseur de Serp doit créer un grand nombre d'objets Java, un pour chaque instruction notamment, ce qui prend beaucoup de temps. **La représentation des programmes par des séquences d'interactions est donc la plus performante en ce qui concerne les temps de désérialisation et de sérialisation des classes compilées.**

Les courbes pour Jabyce et ASM se croisent, pour un nombre de transformateurs $T = 132,4$. **Jabyce est donc plus performant pour la (dé)sérialisation et la transformation de classes que ASM, pour un nombre de transformateurs (seuil) supérieur à 132 dans une même chaîne, dans cette expérience.** Comme expliqué ci-dessus, le coût de l'ajout d'un transformateur Jabyce dans la chaîne est linéaire, au lieu d'être quadratique dans ASM. Ainsi, contrairement à ASM, l'insertion des nouvelles instructions de trace dans les méthodes

par une opération Jabyce n'augmente pas le temps d'exécution des opérations Jabyce suivantes dans la chaîne. Le seuil calculé est élevé, pour la transformation implantée dans ces tests, mais il pourrait être plus bas si par exemple 30 instructions étaient insérées avant chaque instruction *x*return, au lieu de 3 instructions.

La courbe de Serp ne croise pas celle des autres systèmes : Jabyce et ASM sont toujours plus performants que Serp, quelque soit le nombre de transformateurs. Ces bonnes performances sont globalement dûes à la non-redondance du balayage des classes, contrairement à Serp. Dans ASM et Jabyce, chaque classe n'est balayée qu'une seule fois, en même temps qu'elle est représentée, quelque soit le nombre T de transformateurs dans la chaîne. En revanche, dans les systèmes tels que Serp qui représentent les programmes par des graphes d'objets, chaque transformateur balaie chaque classe, ce qui implique une redondance inutile du balayage, comme décrit dans la section 5.6.

Considérons seulement les temps de transformation de l'ensemble des classes par un seul transformateur ($T = 1$ et $R = 86606$), sans inclure les temps de (dé)sérialisation et de lecture / écriture (facteurs constants dans les formules), et en supposant pour simplifier que $Csei_{jabyce} = Csei_{asm} = Csei_{serp} = 0$. Ces temps de transformation sont de 973 ms avec Jabyce, 584 ms avec ASM, et 3357 ms avec Serp. Comme expliqué ci-dessus, **le temps de transformation est supérieur avec Serp qu'avec Jabyce et ASM, à cause du balayage qui doit être effectué dans le transformateur**. En revanche, l'exécution d'un seul transformation ASM prend moins de temps que d'exécuter un seul transformateur Jabyce. Nous considérons que cela est dû à la plus grande compacité de l'implantation du transformateur avec ASM qu'avec Jabyce. Notamment, dans le transformateur Jabyce le nom de la classe et le nom de la méthode, utilisés pour construire le message de trace, sont récupérés en effectuant des appels sur les interfaces du composant opération suivant dans la chaîne (appels boundMethodFactory.getCompositeClass(...), boundClassFactory.getBuiltClassClassType(...) et boundMethodFactory.getBuiltMethodName(...)). En effet, un composant opération Jabyce est utilisé pour représenter / transformer plusieurs classes. Par exemple, deux appels successifs de la méthode createReturnInstruction(...) sur un composant opération peuvent représenter des instructions *x*return dans des méthodes ou des classes différentes, si les séquences d'interactions qui les représentent sont entrelacées. Les noms des méthodes et des classes doivent donc être récupérés en fonction des objets de contexte passés en paramètres, qui identifient les représentations des méthodes et classes. En revanche, avec ASM, un objet transformateur implantant ClassVisitor (resp. CodeVisitor) n'est utilisé pour représenter qu'une seule classe (resp. une seule méthode). Ainsi, dans notre implantation de transformateur avec ASM (cf. les programmes 6.6 et 6.7), le nom de la classe et le nom de la méthode sont récupérés en interceptant les appels visit(...) et visitMethod(...), et sont conservés dans des attributs du transformateur (className et methodName). Ainsi, à chaque transformation d'une instruction *x*return, les noms sont récupérés directement dans ces attributs, et il n'est pas nécessaire d'effectuer des appels complexes pour les récupérer comme avec Jabyce. De plus, l'implantation du transformateur Jabyce comprend de nombreuses créations d'objets, notamment de type ObjectType et FieldDescriptor, ce qui n'est pas le cas avec ASM et Serp. **Ainsi, la compacité du code du transformateur ASM permet de meilleures performances qu'avec le transformateur Jabyce, pour notre exemple de transformation simple.** Cependant, pour effectuer des transformations plus complexes, nous supposons que le modèle conceptuel plus abstrait de Jabyce permet une implantation plus concise des transformateurs qu'avec ASM, et donc que l'écart des temps d'exécution est plus réduit. **Nous considérons que Jabyce est**

plus adapté pour l'implantation de transformations complexes que ASM, alors que ASM est plus adapté pour l'implantation de transformations simples. Notamment, les transformateurs complexes présentés dans le chapitre 7, sont conçus de manière efficace et élégante sous la forme de composants opération Jabyce.

Autres systèmes avec une représentation par graphes d'objets

Nous considérons a priori que Serp est le système offrant une représentation des programmes par des graphes d'objets, qui est le plus performant parmi les systèmes considérés dans le chapitre 5. Serp est conçu de manière très similaire à JOIE, et nous supposons que les temps de transformation sont très proches pour ces systèmes, ce qui reste encore à vérifier par des mesures. Nous avons effectué quelques mesures des temps de transformation avec JMangler / BCEL, et nous avons observé que ces temps augmentent beaucoup plus rapidement qu'avec Serp, avec le nombre de transformateurs composés. En effet, dans JMangler, des objets ClassGen (une classe de BCEL) sont passés en paramètres aux transformateurs. Or, les objets ClassGen contiennent des objets Method, qui sont non modifiables. Pour transformer chaque méthode, il faut donc 1) créer un objet MethodGen à partir d'un objet Method (ce qui correspond à une désérialisation de la méthode), 2) modifier l'objet MethodGen, 3) générer un nouvel objet Method à partir de l'objet MethodGen (ce qui correspond à une sérialisation), et 4) ajouter cet objet Method dans l'objet ClassGen. Ces traitements doivent être effectués par chaque transformateur de méthode. Cette redondance des traitements devient prohibitive lors de la composition de plusieurs transformateurs : à partir de 2 transformateurs, JMangler / BCEL est moins performant que Serp. Nous n'avons pas implanté la transformation d'exemple avec Javassist. Cependant, nous supposons que les temps de transformation sont supérieurs qu'avec Serp. Notamment, nous avons indiqué dans la section 5.6 qu'avec Javassist, lorsque des transformateurs sont composés ensemble, une désérialisation et une sérialisation de chaque classe sont effectuées pour chaque transformateur. Ces (dé)sérialisations redondantes des classes sont a priori très coûteuses en temps d'exécution. De plus, la seule manière d'implanter cette transformation avec Javassist est d'utiliser la syntaxe concrète de Javassist (cf. la section 5.8), en passant du code source Java dans une chaîne en paramètre à un appel à insertAfter(...) sur chaque objet CtMethod représentant une méthode. Ce code source est compilé pour chaque méthode transformée, ce qui implique des temps de traitement que nous supposons non négligeables.

6.5 Synthèse

Ce chapitre présente des mesures et une analyse des temps d'exécution de chaînes de transformateurs, développés respectivement avec Jabyce, ASM et Serp. De manière générale, nous avons étudié l'impact sur les temps de transformation de quatre caractéristiques : 1) la représentation des programmes par des séquences d'interactions *vs.* la représentation par des graphes d'objets, 2) l'impact de l'implantation de Fractal sur les performances de Jabyce, 3) le choix d'un modèle conceptuel abstrait *vs.* peu abstrait, 4) la redondance et le déterminisme des stratégies de transformation.

Une représentation des programmes par des séquences d'interactions est plus performante qu'une représentation par des graphes d'objets. En, effet, pour la transformation simple util-

isée en exemple, Jabyce est le système de transformation de classes compilées Java le plus performant à partir d'un certain nombre de transformateurs composés ensemble. En dessous de ce seuil, ASM est le système le plus performant.

Le choix du modèle de composant Fractal pour la conception des opérations de transformation dans Jabyce n'a aucun impact sur les performances, grâce aux optimisations effectuées automatiquement par Julia, l'implantation de référence de Fractal en Java.

Pour la transformation simple utilisée comme exemple, l'utilisation d'un modèle conceptuel très abstrait comme celui de Jabyce complexifie l'implantation du transformateur et implique des traitements supplémentaires par rapport à un modèle peu abstrait comme celui de ASM. Ainsi, le seuil calculé à partir duquel Jabyce est le plus performant est très élevé : environ 132 transformateurs simples. De plus, une caractéristique de la transformation d'exemple est que les instructions de trace insérées avant chaque instruction *x*return ne contiennent pas d'instructions *x*return qui doivent elles-même être transformées par les transformateurs suivants dans la chaîne. Cette caractéristique avantage Jabyce, en lui permettant d'avoir une croissance linéaire du temps de transformation en fonction du nombre de transformateurs, au lieu d'une croissance quadratique avec ASM et Serp. En considérant la composition d'opérations de transformation différentes, le coût de l'insertion d'une nouvelle opération dans une chaîne pourrait se traduire par une augmentation non linéaire du temps de transformation avec Jabyce, comme nous l'avons observé pour ASM, et donc une augmentation du seuil à partir duquel Jabyce est le plus performant. Cependant, en général, nous supposons que le coût de l'ajout d'une opération de transformation dans une chaîne augmente moins rapidement pour Jabyce que pour ASM. De plus, nous considérons que les transformateurs complexes peuvent être implantés avec Jabyce de manière plus concise qu'avec ASM, parce que Jabyce offre un modèle conceptuel à un niveau d'abstraction plus élevé. Nous considérons que cette relative simplicité pour l'implantation de transformateurs complexes permet à Jabyce d'être plus performant que ASM pour la composition d'un nombre peu élevé de transformateurs complexes. Cette assertion est cependant difficile à vérifier, car elle nécessite d'implanter plusieurs transformations complexes avec plusieurs systèmes de transformation.

Nous avons observé que les temps de transformation augmentent plus vite avec Serp en fonction du nombre de transformateurs composés, qu'avec Jabyce et ASM. Nous supposons que ces temps de transformation élevés sont principalement dûs à la redondance du balayage des classes dans chaque transformateur, contrairement à Jabyce et ASM.

En ce qui concerne la comparaison quantitative des systèmes de transformation de programmes Java compilés, une perspective est la mesure de la consommation de mémoire de ces systèmes, c'est-à-dire *le comptage du nombre d'objets créés* avec chacun des systèmes, et le calcul de leur taille. Il est certain que les systèmes qui offrent une représentation des programmes par des graphes d'objets (JOIE, JMangler, Javassist, BCEL et Serp) créent beaucoup plus d'objets que les systèmes offrant une représentation par des séquences d'interactions (ASM et Jabyce). Notamment, comme expliqué dans la section 5.3, il est nécessaire de créer un objet pour représenter chaque instruction. En revanche, avec ASM et Jabyce, des objets sont créés seulement pour être passés en paramètres des interactions représentant des éléments de programmes. Avec Jabyce, nous supposons a priori qu'un plus grand nombre d'objets sont créés pour être passés en paramètres. Ce point devra être vérifié par des mesures, qui nous permettront également de localiser les parties du code de Jabyce qui créent le plus d'objets, de manière à les optimiser. Notamment, les implantations de désérialiseurs et de sérialiseurs

dans Jabyce sont moins performants que celles de ASM, comme nous l'avons montré dans ce chapitre. Nous supposons que ces temps de traitement plus élevés sont en partie dûs à la création d'un plus grand nombre d'objets lors de la désérialisation. L'une des perspectives pour Jabyce est donc d'optimiser l'implantation de désérialiseur, par exemple en réutilisant les objets déjà créés (*pool* d'objets).

Chapitre 7

Tisseur d'un lien fonctionnel / technique pour la persistance d'objets

Nous utilisons métaphoriquement le terme de *tisseur*, pour désigner un logiciel qui construit un lien entre une application et un ou plusieurs services techniques. D'après notre démarche générale décrite dans la section 2.3, un tisseur établit un lien entre une architecture fonctionnelle et une architecture technique conçues avec des composants Fractal.

Ce chapitre présente une validation expérimentale de l'utilité de Jabyce pour la construction d'un tisseur, pour un service technique particulier : la persistance transparente d'objets Java. Le service technique considéré en pratique est Speedo, qui est une implantation du standard JDO (*Java Data Objects*) pour la persistance transparente d'objets Java. Speedo est développé sous la forme d'un logiciel libre dans le cadre du consortium Objectweb par le laboratoire DTL/ASR de France Telecom R&D [Objd]. JDO et Speedo sont brièvement décrits dans la section 7.1.

Ce chapitre décrit les deux opérations de transformation les plus significatives parmi celles implantées pour la construction d'un tisseur pour Speedo. Cette description se focalise sur l'architecture et la complexité de ces opérations de transformation. L'objectif de ce chapitre est de montrer que 1) Jabyce est bien adapté à la conception, à la réutilisation et à l'adaptation de transformateurs complexes, notamment grâce aux propriétés du modèle de composant Fractal qui est utilisé dans Jabyce, et que 2) Jabyce permet une implantation et une composition efficace de transformateurs complexes. Ces deux opérations de transformation - le mélange de classes, et la réification de l'état des objets - sont décrites respectivement dans les sections 7.2 et 7.3.

7.1 Survol de JDO et de Speedo

Le standard JDO spécifie les interfaces de programmation (API), sous la forme d'interfaces Java, pour la conception d'applications qui accèdent à des données dans des bases de données, des fichiers, ou à travers des moniteurs transactionnels tels que Tuxedo. Afin d'éviter aux programmeurs d'utiliser des API spécifiques à chaque environnement de stockage, JDO offre un moyen de manipuler les données sous la forme d'objets Java, sans se préoccuper de la

manière dont ces objets sont stockés de manière persistante. Speedo implante la version 1.0.1 du standard JDO [Gro03].

Les trois concepts architecturaux de base du standard JDO sont : les *objets persistants* (*JDO Instance*), les *implantations* de service de persistance (*JDO Implementation*), et les *tisseurs JDO* (*JDO Enhancer*). JDO correspond tout à fait à la démarche générale suivie dans notre laboratoire, décrite dans la section 2.3. En effet, le standard JDO a pour objectif de permettre de développer séparément des implantations de service de persistance et des applications fonctionnelles (ensembles d'objets persistants). Conformément à la démarche générale de notre laboratoire, Speedo est conçu en utilisant le modèle de composants Fractal, décrit dans la section 3.2.1. Dans JDO, le lien réalisé par un tisseur entre une application et un service de persistance est transparent. Plusieurs techniques sont proposées dans la spécification de JDO pour réaliser ce lien : 1) par transformation de code compilé, 2) par transformation de code source (*preprocessing*), ou 3) par génération de code source [Gro03]. La spécification de JDO décrit informellement, en langue naturelle (en anglais), les transformations de programme qui doivent être mises en œuvre par un tisseur JDO. La transformation de code compilé est décrite comme la plus transparente (pas de nécessité pour les objets persistants d'offrir une interface PersistenceCapable, etc.). C'est pourquoi le tisseur de Speedo repose principalement sur cette technique, et comprend des transformateurs de classes compilées Java. Pour l'instant, les transformateurs de classes compilées de Speedo sont développés en utilisant le système ASM, décrit dans le chapitre 5. Ces implantations de transformateurs sont difficiles à maintenir, trop spécifiques à Speedo, et difficilement séparables et réutilisables. C'est pourquoi nous avons redéveloppé ces transformateurs, en utilisant Jabyce, avec pour objectif d'aboutir à des transformateurs généraux plus performants, plus configurables et mieux réutilisables. Deux de ces transformateurs sont présentés dans la suite de ce chapitre. La conception de ces transformateurs montre notamment l'apport de l'utilisation du modèle de composant Fractal, dans Jabyce, pour faciliter la conception de transformateurs complexes.

La manière de réaliser la projection (*mapping*) des objets vers un support de persistance est spécifique aux implantations particulières de services de persistance JDO. En particulier, Speedo se base sur le canevas logiciel JORM, développé sous la forme d'un logiciel libre dans le cadre du consortium Objectweb par le laboratoire DTL/ASR de France Telecom R&D [ACBD+04, Objc]. JORM offre plusieurs mécanismes de projection d'objets Java : vers des bases de données relationnelles, vers des fichiers, vers des bases de données en mémoire, etc. Le choix d'un de ces mécanismes, et les paramètres de la projection d'un ensemble d'objets Java, sont spécifiés dans des fichiers XML. Speedo repose sur JORM pour la persistance des objets, et offre une implantation des interfaces (API) définies dans le standard JDO, notamment pour l'exécution de requêtes sur des objets persistants.

Pour assurer la persistance des objets, JORM s'appuie sur une technique de génération de code source de classes d'objets utilitaires. Speedo s'appuie également sur la génération de code source de classes d'objets utilitaires, qui héritent des classes générées par JORM, et adaptées aux interfaces de JDO (PersistenceCapable, etc.). Pour rendre transparente l'utilisation de ces classes générées, et pour diminuer le nombre d'objets utilitaires pendant l'exécution de l'application, le tisseur JDO de Speedo comprend principalement une transformation de code Java compilé de *mélange des classes* d'objets utilitaires dans les classes d'objets persistants. Cette transformation est actuellement implantée avec ASM. Ainsi, après transformation, le système résultant contient seulement les classes originales d'objets persistants, transformées mais toujours compatibles du point de vue de la compatibilité binaire [DWE98]. La génération

de code source étant une technique plus simple que la transformation de code compilé, lorsqu'il s'agit de générer de grandes quantités de code, cette combinaison de la génération de code source et du mélange de code compilé est un bon compromis en termes de performance, de transparence et de simplicité. Ce transformateur de mélange a été redéveloppé avec Jabyce, et sa conception est détaillée dans la section 7.2.

Un autre transformateur a été développé avec Jabyce, avec pour objectif de remplacer les transformateurs actuellement implantés avec ASM. Ce transformateur, présenté dans la section 7.3, transforme les classes compilées pour extraire des attributs hors des classes, vers des classes d'objets « état ». Ainsi, ces objets état *réifient l'état des objets* des classes transformées, pour faciliter la manipulation de l'état des objets. Notamment, cette transformation simplifie la gestion de l'accès transactionnel aux objets persistants, dans le cas d'une gestion de transactions optimiste : il suffit de créer et de gérer un objet état pour chaque transaction touchant un objet persistant.

Ces deux transformateurs, de mélange des classes et de réification de l'état des objets, ont été développés pour être indépendants du contexte de leur utilisation dans Speedo.

7.2 Transformation pour le mélange de classes

Le transformateur de mélange présenté dans cette section permet de « copier » tous les éléments (attributs, méthodes, déclarations d'interfaces implantées, etc.) d'une classe, dans sa super-classe, en transformant le code compilé de ces super-classes. Considérons les classes compilées A et B, avec B qui hérite de A, dont le code source est donné dans les programmes 7.1 et 7.2. Le résultat du mélange est la classe compilée A, dont le code source équivalent est donné dans le programme 7.3. Cet exemple est simplifié.

Nous imposons des limitations, sous la forme de conventions de programmation, pour nous faciliter l'implantation de la transformtion de mélange d'une sous-classe vers sa super-classe (classe de base) :

- une classe de base doit définir un constructeur sans argument, et non privé ;
- une sous-classe à mélanger ne doit définir qu'un seul constructeur sans argument ;
- un constructeur d'une sous-classe à mélanger ne doit pas appeler un constructeur de la super-classe autre que le constructeur sans argument, pour initialiser « this » ;
- une sous-classe à mélanger ne doit pas définir des attributs de même nom que des attributs de sa super-classe, ni redéfinir des méthodes de sa super-classe : aucun renommage des membres n'est effectué.

7.2.1 Appels aux initialiseurs des sous-classes

La seule transformation apportée au code original de A est l'insertion du code d'un appel à la méthode $_synth_mixedInitCaller_A(), dans tous les initialiseurs d'instance (constructeurs) de A, juste avant toute instruction *x*return de retour de méthode. Cette méthode, générée par le transformateur, appelle tous les initialiseurs transformés (copiés et renommés) à partir des sous-classes. Cet appel doit être réalisé juste avant le retour des initialiseurs, pour respecter la sémantique de l'héritage de Java. Ce mécanisme mis en œuvre par notre transformateur est

```
public class A {
  public int attribut_a_1;
  private static int attribut_a_2;
  public A() {
    super();
    attribut_a_1 = 0;
    return;
  }
  public A(int v) {
    A();
    if (v > 0) {
      attribut_a_1 = v;
      return;
    } else {
      attribut_a_1 = 0;
    }
    return;
  }
  static {
    attribut_a_2 = 0;
  }
}
```

Prog. 7.1 : Code source original de la classe A à transformer

```
public class B extends A {
  public int attribut_b_1;
  public B attribut_b_2;
  private static int attribut_b_3;
  public B() {
    A();
    attribut_b_1 = 123;
    attribut_b_2 = null;
  }
  public void methode_b_1(B v) {
    attribut_a_1 = 0;
    attribut_b_2 = v;
  }
  static {
    attribut_b_3 = 0;
  }
}
```

Prog. 7.2 : Code source original de la classe B

```
public class A {
  public int attribut_a_1; private static int attribut_a_2;
  public A() {
    super();
    attribut_a_1 = 0;
    $_synth_mixedInitCaller_A(); return;
  }
  public A(int v) {
    A();
    if (v > 0) {
      attribut_a_1 = v;
      $_synth_mixedInitCaller_A(); return;
    } else { attribut_a_1 = 0; }
    $_synth_mixedInitCaller_A(); return;
  }
  public int attribut_b_1; public A attribut_b_2;
  private static int attribut_b_3;
  public $_synth_mixedInit_B() {
    /* wrapped not to execute: */ A();
    attribut_b_1 = 123;
    attribut_b_2 = null;
  }
  public void methode_b_1(A v) {
    attribut_a_1 = 0;
    attribut_b_2 = v;
  }
  private boolean $_synth_mixedInitCalled_A;
  private void $_synth_mixedInitCaller_A() {
    if ($_synth_mixedInitCalled_A) { return; }
    $_synth_mixedInitCalled_A = true;
    $_synth_mixedInit_B();
  }
  static {
    attribut_a_2 = 0;
    $_synth_mixedClinit_B; return;
  }
  private static void $_synth_mixedClinit_B {
    attribut_b_3 = 0;
  }
}
```

Prog. 7.3 : Code source équivalent à celui de la classe compilée A transformée

une fonctionnalité supplémentaire par rapport au transformateur de mélange originalement implanté avec ASM dans Speedo, parce qu'il permet d'initialiser les attributs copiés des sous-classes.

Il est possible qu'un initialiseur soit appelé par un autre initialiseur de la même classe, comme c'est le cas dans la classe A. Il est donc nécessaire de garantir que la méthode $_synth_mixedInitCaller_A() n'est appelée qu'une seule fois pour chaque objet. C'est pourquoi un nouvel attribut booléen est généré par le transformateur ($_synth_mixedInitCalled_A). Cet attribut est testé et positionné en début de la méthode $_synth_mixedInitCaller_A().

Un mécanisme similaire est mis en œuvre pour les appels des initialiseurs statiques (parties static { ... } dans le langage Java) copiés à partir des sous-classes, tels que l'initialiseur statique de B, qui est transformé en la méthode $_synth_mixedClinit_B(). Des appels à ces initialiseurs statiques sont insérés avant chaque instruction *x*return de retour de l'initialiseur statique de la superclasse. Ce mécanisme est plus simple que pour les initialiseurs d'instances. En effet, il ne peut exister au plus qu'un initialiseur statique dans une classe.

7.2.2 Transformation des initialiseurs d'instance des sous-classes

Les initialiseurs d'instance (constructeurs) des sous-classes appellent l'initialiseur d'instance sans argument de la super-classe, pour initialiser l'objet « this ». Dans notre exemple, l'initialiseur B() appelle l'initialiseur A() pour initialiser l'objet « this ». Après mélange, cet appel à l'initialiseur de la super-classe ne doit plus être effectué. Notre transformateur transforme donc tout appel à l'initialiseur de A dans l'initialiseur de B. Cette transformation est difficile à implanter, à cause du problème de la détermination des instructions d'appel à transformer : seuls les appels avec « this » comme cible ne doivent pas être effectués.

Une approche simpliste pourrait consister à supposer que le code des classes transformées est directement le résultat de la compilation de code source Java, par un compilateur standard (p.ex. javac de Sun Microsystems). En effet, le langage Java impose que la première instruction d'un constructeur soit un appel à un constructeur de la super-classe ou d'un autre constructeur de la même classe, pour initialiser l'objet « this ». Or, le langage Java est beaucoup plus restrictif que la spécification de la JVM pour le code compilé Java. Un exemple de code compilé Java d'un initialiseur, qui respecte la spécification de la JVM, est donné dans le programme 7.4. Ce code n'est pas décompilable, c'est-à-dire qu'il n'existe pas de code source Java qui puisse être compilé vers ce code compilé. Un pseudo-code source Java de cet initialiseur est donné dans le programme 7.5.

D'après la spécification de la JVM, la création d'un objet est réalisée en 2 phases : d'abord 1) la création de l'objet non initialisé avec une instruction new, puis 2) l'initialisation de l'objet par un appel à un *initialiseur* (constructeur) avec une instruction invokespecial et avec l'objet non initialisé en cible de l'appel.[1] Dans le code d'un initialiseur, l'objet « this » est d'abord non initialisé, jusqu'à ce qu'un initialiseur de la super-classe ou de la même classe soit appelé, avec l'objet « this » en cible. Cette initialisation doit être effectuée exactement une fois, dans un initialiseur. Dans notre contexte, le problème est d'identifier quelle instruction invokespecial est appelée avec « this » en cible, pour appeler un initialiseur de la super-classe, pour pouvoir ignorer cet appel.

[1] Les initialiseurs d'instances ont tous le nom spécial <init>, et les initialiseurs de classes le nom <clinit>.

```
public class SousClasse extends SuperClasse{
public SousClasse(boolean);
  Code:
   0:   iload_1
   1:   ifeq    8  // test du paramètre booléen
   4:   aload_0    // empilement de «this»
   5:   goto    11
   8:   new     #1 // crée un objet SuperClasse
                   // non initialisé
   11:  invokespecial #2 // appelle
     // SuperClasse.<init>() pour initialiser
     // l'objet sur la pile (empilé par instruction 4 ou 8,
     // suivant le paramètre booléen)
   14:  iload_1
   15:  ifne    22 // test du paramètre booléen
   18:  aload_0    // empilement de «this»
   19:  invokespecial #2 // appelle
     // SuperClasse.<init>() pour initialiser
     // l'objet sur la pile (empilé par instruction 18)
   22:  return
}
```

Prog. 7.4 : Code compilé d'un constructeur complexe

```
public class SuperClasse {
  public SuperClasse() {
    /* ...  */
  }
}
public class SousClasse extends SuperClasse {
  public SousClasse(boolean b) {
    (b ? this : new SuperClasse).superclasse_init();
    if (! b) { this.superclasse_init(); }
  }
}
```

Prog. 7.5 : Pseudo-code source d'un constructeur complexe

L'heuristique simple qui consiste à considérer seulement la première instruction invokespecial d'appel d'une méthode <init> de la super-classe, n'est pas valide. Dans notre exemple, cette première instruction est à la position 11, mais n'est pas toujours exécutée avec l'objet « this » en paramètre. Plus généralement, il n'est pas possible de déterminer statiquement quelle instruction invokespecial est appelée pour initialiser l'objet « this », sauf dans des cas particuliers. Dans notre exemple, l'instruction d'initialisation de « this » est soit l'instruction 11, soit l'instruction 19, en fonction du paramètre booléen.

Pour simplifier l'implantation de notre transformateur, l'instruction invokespecial d'appel de l'initialiseur d'instance de la super-classe pour « this », est déterminée pendant l'exécution. Pour cela, notre transformateur transforme chaque instruction invokespecial d'appel à un initialiseur de la super-classe, pour ajouter du code qui teste la cible de l'appel. A l'exécution, si la cible de l'appel est « this », l'appel n'est pas effectué. Le résultat de la transformation de l'initialiseur du programme 7.4 est donné dans le programme 7.6. Les instructions précédées d'un astérisque (*) sont les instructions insérées par le transformateur. La référence à l'objet « this » est passée dans la variable locale à l'index 0, au début de tout appel d'une méthode. Cette variable locale peut être modifiée dans une méthode. Notre transformateur insére donc du code pour sauvegarder cette référence dans une autre variable locale non modifiée (à l'index 2 dans l'exemple), au début de la méthode.

```
public class SousClasse extends SuperClasse{
public SousClasse(boolean);
  Code:
 * 0:   aload_0    // sauvegarde de «this»
 * 1:   astore_2   // dans la variable 2
   2:   iload_1
   3:   ifeq    10
   6:   aload_0
   7:   goto    13
   10:  new     #1
 * 13:  dup // test de la cible de l'appel invokespecial
 * 14:  aload_2 // empilement de «this»
 * 15:  if_acmpeq 24 // comparaison avec «this»
   18:  invokespecial #2 // appel effectué
 * 21:  goto    25
 * 24:  pop // appel non effectué
   25:  iload_1
   26:  ifne    42
   29:  aload_0
 * 30:  dup // test de la cible de l'appel invokespecial
 * 31:  aload_2 // empilement de «this»
 * 32:  if_acmpeq 41 // comparaison avec «this»
   35:  invokespecial #2 // appel effectué
 * 38:  goto    42
 * 41:  pop // appel non effectué
   42:  return
}
```

Prog. 7.6 : Code compilé d'un constructeur complexe après transformation

Cet exemple d'initialiseur est plus complexe que les cas considérés dans notre contexte. Notamment, nous imposons que les initialiseurs n'aient pas d'argument. Cependant, le problème se pose toujours dans des cas plus particuliers mais toujours conformes à la spécification de la JVM, tels que par exemple la création d'une instance de la super-classe et son affectation à un attribut privé, avant l'appel à l'initialiseur de la super-classe pour « this ».

7.2.3 Renommage des sous-classes

En plus des aspects difficiles décrits ci-dessus, lors du mélange d'une sous-classe, par exemple de B dans A, il est nécessaire de transformer toute référence à B en une référence à A. Ceci constitue l'essentiel des transformations effectuées. Les éléments de programme de 13 types, qui font référence ou peuvent faire référence à des noms de classes Java, sont transformés : méthodes (type de retour), arguments des méthodes, attributs (type), déclarations de classes internes, descriptions de variables locales, déclarations d'exceptions levées, déclarations de gestionnaires d'exception (si une sous-classe est utilisée comme type d'exception), instructions de test de type, instructions d'accès aux attributs, instructions d'appels de méthodes, instructions de création de tableaux (si une sous-classe est utilisée comme type « composant » de tableau) monodimensionnels et multidimensionnels, et instructions de création d'objets.

7.2.4 Architecture de l'opération de transformation

L'architecture d'une opération de mélange de classes est illustrée dans la figure 7.1. Cette opération est un composant composite. L'architecture de ce composant composite semble relativement complexe, car il y a deux niveaux de composition et ce composant contient 394 liaisons, bien que la structure soit très régulière. Les trois sous-composants primitifs qui ont été développés spécifiquement pour cette transformation sont colorés en gris foncé. Le code des interfaces Java et des classes Java d'implantation de ces composants comprend au total 3360 lignes de code Java commenté.

Le *composant transformateur des classes de base*, en haut à gauche, transforme les classes de base dans lesquelles sont mélangées des classes. Ce composant est lié à un composant contenant des méta-données, via une interface de type ClassMixingMetaDataRepository (cf. le programme 7.7). Les méta-données sont les relations entre les classes de base et les classes à mélanger. Ainsi, pour chaque classe représentée sur les interfaces serveur du transformateur (appels à createClass(...)), le transformateur détermine grâce à ces méta-données si la classe est une classe de base dans laquelle doivent être mélangées des classes. Si c'est le cas, le transformateur applique à cette classe de base les transformations pour insérer des appels aux initialiseurs des classes mélangées (cf. la section 7.2.1), et délègue à un *composant mélangeur* la représentation des éléments transformés à partir de chaque classe à mélanger, via une interface de type SubClassMixer (cf. le programme 7.8). La méthode mixSubClass(...) prend en paramètres le nom de la classe de base, le nom de la classe à mélanger, et l'objet ClassBuildingContext qui identifie la classe de base pour pouvoir représenter les éléments mélangés (méthodes, attributs, etc.).

Cette interface d'accès aux méta-données est générale, et différents composants peuvent être implantés pour accéder à des représentations de méta-données différentes. Par exemple, dans Speedo, un composant peut être implanté pour accéder aux graphes d'objets Speedo qui

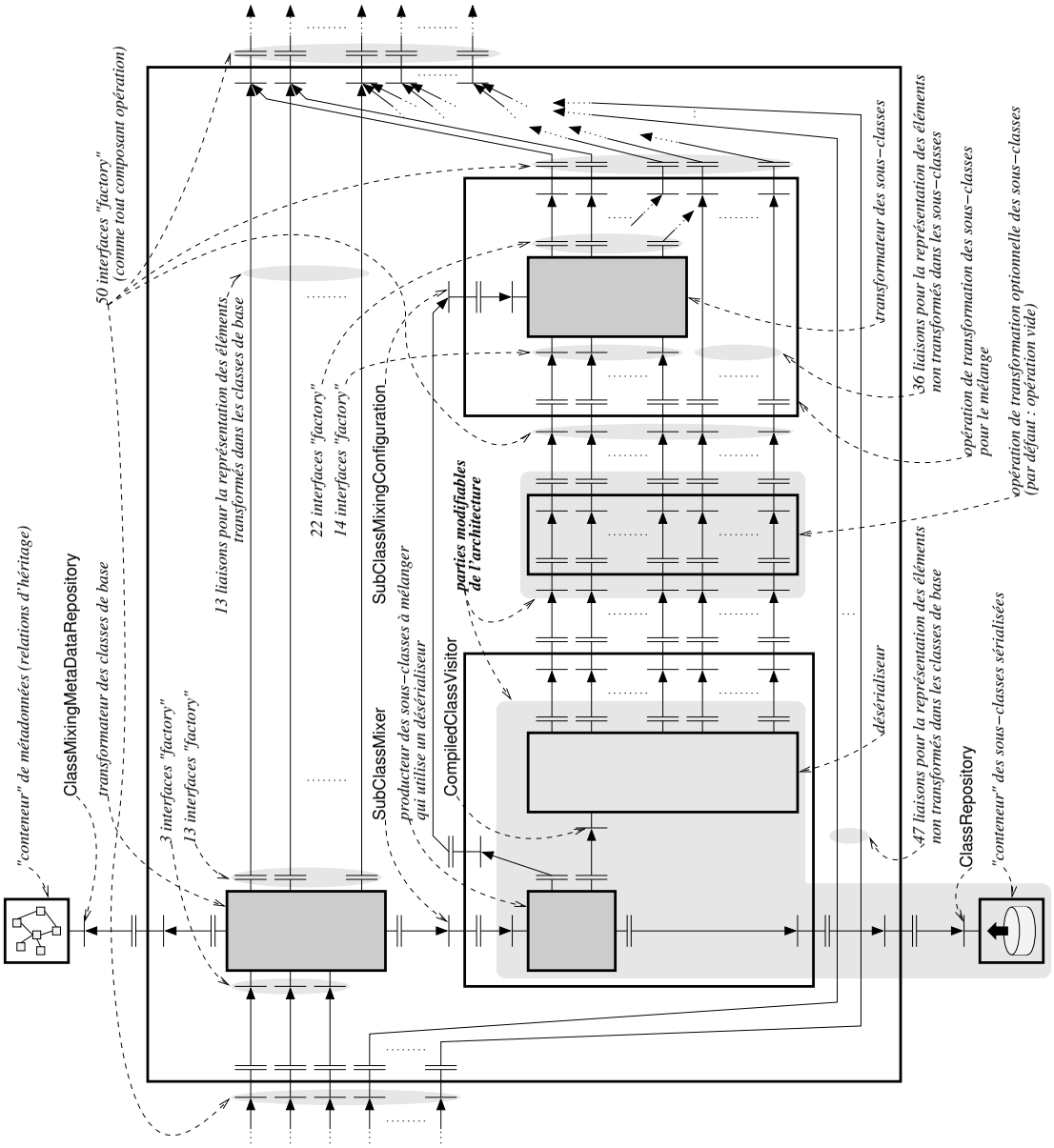


FIG. 7.1 : Architecture d'une opération de mélange de classes

```
public interface ClassMixingMetaDataRepository {
  boolean containsClassesToMixInto(ObjectType baseClassType);
  ObjectType[] getClassesToMixInto(ObjectType baseClassType);
}
```

PROG. 7.7 : Interface ClassMixingMetaDataRepository

```
public interface SubClassMixer {
  void mixSubClass(ObjectType baseClassType,
      ClassBuildingContext baseClassBuildingContext,
      ObjectType mixedClassType);
}
```

Prog. 7.8 : Interface SubClassMixer

représentent les méta-données de persistence des applications. Cependant, grâce à cette « inversion des dépendances », l'opération de transformation est indépendante des détails des méta-données de Speedo, et peut être réutilisée dans d'autres contextes, contrairement au transformateur initialement implanté avec ASM.

Il peut y avoir plusieurs implantations de *composant mélangeur*. Par exemple, une implantation pourrait compiler dynamiquement une spécification abstraite d'une telle classe, pour représenter des méthodes, attributs, etc. à mélanger dans les classes. L'implantation que nous avons réalisée est un composant composite, offrant une interface de type SubClassMixer, et qui utilise un composant désérialiseur pour désérialiser les classes à mélanger. Les classes sérialisées sont stockées dans un composant conteneur, et accédées via une interface de type ClassRepository (interface Java définie dans Jabyce). Ces classes désérialisées doivent être transformées, comme décrit dans les sections 7.2.2 et 7.2.3 : c'est le rôle de *l'opération de transformation des classes mélangées*, en bas à droite dans la figure 7.1. Ce composant est lié au composant mélangeur, à travers des interfaces de type SubClassMixingConfiguration (cf. programme 7.9), pour pouvoir représenter les éléments des classes mélangées dans la représentation de la classe de base.

```
public interface SubClassMixingConfiguration {
  void beginMixing(ObjectType baseClassType,
      ClassBuildingContext baseClassBuildingContext,
      ObjectType mixedClassType);
  void endMixing();
  boolean isMixing();
  ObjectType getBaseClassType();
  ClassBuildingContext getBaseClassBuildingContext();
  ObjectType getMixedClassType();
}
```

Prog. 7.9 : Interface SubClassMixingConfiguration

7.2.5 Architecture pour mélange à deux niveaux

Comme illustré dans la figure 7.1, une opération de transformation est liée, entre *le composant mélangeur* et *l'opération de transformation de classes mélangées*, pour permettre éventuellement de transformer les sous-classes avant qu'elles soient mélangées. Par défaut, cette opération est une opération vide, qui n'a aucun impact sur les performances. Cependant, il est possible de remplacer cette opération vide par n'importe quelle opération.

Le principal avantage de cette architecture est de permettre de mélanger des classes à plusieurs niveaux d'héritage. En effet, l'architecture d'opération de mélange illustrée dans la figure 7.1 permet de mélanger seulement une classe de base et ses sous-classes directes, à un seul niveau d'héritage, en un seul balayage de la classe de base. Or, dans Speedo, il est nécessaire de réaliser un mélange à deux niveaux d'héritage, par exemple une classe C dans sa super-classe B, puis B dans sa super-classe A. Il est possible de réaliser une telle transformation, en remplaçant l'opération vide dans une opération de mélange, par un autre composant opération de mélange, comme illustré dans la figure 7.2. Le sous-composant opération de mélange mélange les classes au plus bas niveau dans la hiérarchie d'héritage, par exemple la classe C dans la classe B. L'opération de mélange de plus haut niveau mélange les classes au plus haut dans la hiérarchie, par exemple la classe nouvelle clase B dans la classe A. Ce composant composite contient globalement 740 liaisons, et 8 composants primitifs (en gris foncé dans la figure).

Cette *architecture fractale* est très performante : chaque classe n'est désérialisée et balayée qu'une seule fois. En effet, le transformateur précédemment développé avec ASM désérialise la classe de base plusieurs fois pour chaque mélange, ce qui est un traitement inutilement redondant. Cette architecture est fractale, parce qu'il est possible de répéter cette auto-similarité du composant opération de mélange, autant de fois qu'il y a de niveaux dans la hiérarchie d'héritage entre les classes à mélanger.

7.2.6 Synthèse

L'implantation de cette transformation de mélange de classes avec Jabyce illustre l'apport de l'utilisation d'un modèle de composant général tel que Fractal pour la conception d'un système de transformation. Un avantage immédiat est une grande modularité des transformateurs, et une composition efficace et flexible des composants effectuant des transformations. Plus originalement, la composition hiérarchique offerte par Fractal permet de traiter des cas complexes de transformation récursive de programmes, qui peuvent ainsi être traités par des *architectures fractales* d'opérations de transformation, comme illustré dans la figure 7.2. Du point de vue des temps de transformation, la conception de notre opération de mélange est optimale : chaque classe Java compilée n'est désérialisée et balayée qu'une seule fois, et l'architecture interne « creuse » des opérations de transformation permet d'atteindre de bonnes performances comme décrit dans le chapitre 6.

7.3 Réification de l'état des objets

Cette section présente une opération de transformation pour l'extraction automatique des attributs d'une classe, vers une classe d'*objet état*. Ainsi, un objet état *réifie l'état d'un objet* d'une classe transformée. L'architecture de cette opération est plus simple que celle de l'opération de mélange de classes décrite dans la section précédente. Comme illustré dans la figure 7.3, cette opération est un composant composite contenant un composant transformateur primitif. Ceci est l'architecture classique que nous avons proposée pour la conception d'opérations de transformation Jabyce, décrite dans la section 3.2.6.

En revanche, les transformations effectuées par ce transformateur sont plus complexes que celles effectuées pour le mélange des classes. Par exemple, le résultat de la transformation des

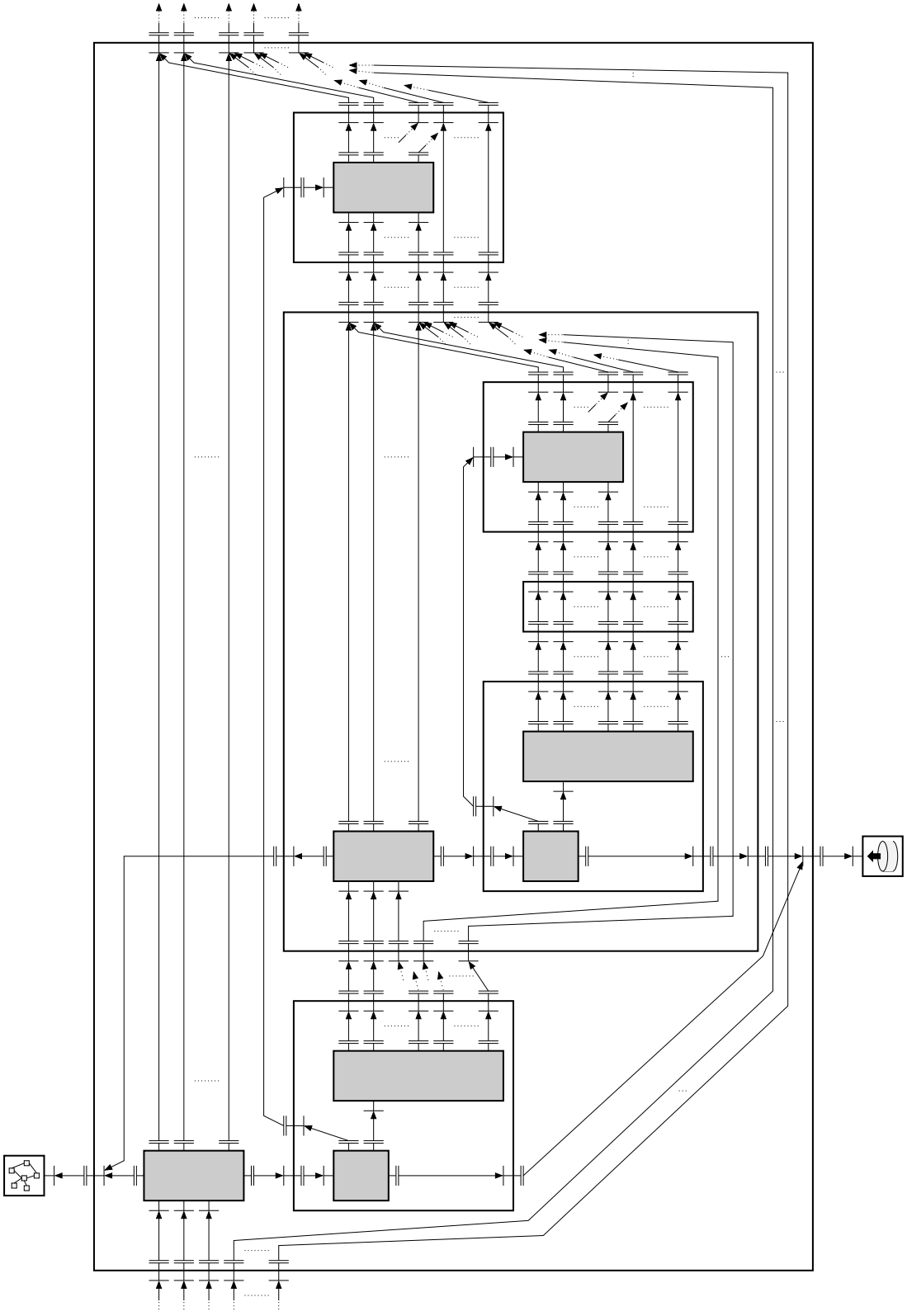


Fig. 7.2 : Architecture « fractale » d'une opération de mélange de classes à deux niveaux

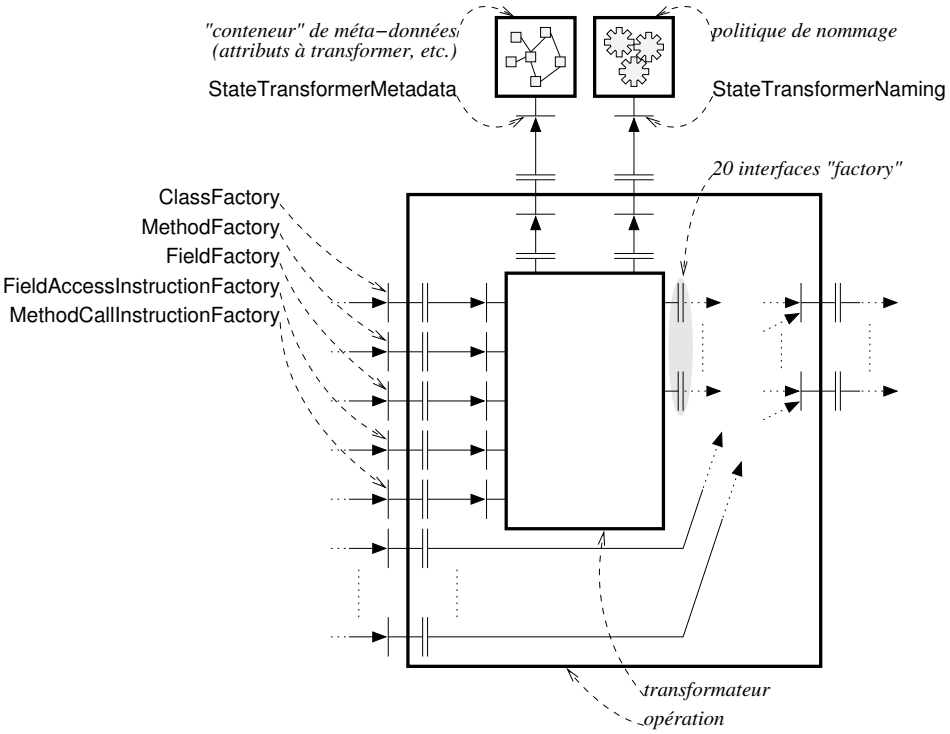


FIG. 7.3 : Architecture d'une opération de transformation pour l'application automatique du patron de conception Memento

classes compilées dont le pseudo-code source est donné dans le programme 7.10, est l'ensemble des classes compilées dont un pseudo-code source à peu près équivalent est donné dans les programmes 7.11*a* à 7.11*e*.

```
public class DesAttributs {
  private int attribut_1;
  public double attribut_2;
  protected int attribut_3;
  public DesAttributs() { super(); attribut_1 = 1; }
  public DesAttributs(int i) { super(); attribut_1 = i; }
  public void affiche() { System.out.println(attribut_1); }
}

public class AutresAttributs extends DesAttributs {
  private boolean attribut_4;
  private int attribut_5;
  public AutresAttributs(int i) {
    attribut_4 = true;
    super(i);
    attribut_3 = 4;
    attribut_5 = 12;
  }
  public AutresAttributs() { AutresAttributs(0); }
}

public class UnClient {
  public void uneMethode(AutresAttributs obj) {
    System.out.println(obj.attribut_1);
  }
}
```

Prog. 7.10 : Classes DesAttributs, AutresAttributs et UnClient avant transformation

```
public class EtatDesAttributs extends EtatAbstrait {

  public int etat_attribut_1;
  public double etat_attribut_2;

  public EtatDesAttributs() { super(); }

  public void copieEtatDesAttributs(
      EtatDesAttributs o) {
    this.etat_attribut_1 = o.etat_attribut_1;
    this.etat_attribut_2 = o.etat_attribut_2;
  }

}
```

Prog. 7.11*a* : Classe générée EtatDesAttributs

```
public class EtatAutresAttributs
    extends EtatDesAttributs {

  public int etat_attribut_4;

  public EtatAutresAttributs() { super(); }

  public void copieEtatAutresAttributs(
      EtatAutresAttributs o) {
    copieEtatDesAttributs(o);
    this.etat_attribut_4 = o.etat_attribut_4;
  }

}
```

PROG. 7.11*b* : Classe générée EtatAutresAttributs

Dans le contexte de cette transformation, nous définissons deux catégories de classes : 1) les *classes décomposées* (« énucléées »), dont des attributs sont extraits, et 2) les *classes clientes*, qui accèdent aux attributs extraits de ces classes. Dans notre exemple, les classes DesAttributs et AutresAttributs sont des classes décomposées et clientes, et la classe UnClient est une classe cliente.

7.3.1 Interfaces de paramétrage

L'interface de type StateTransformerMetadata (cf. le programme 7.12) du transformateur permet de paramétrer les attributs qui doivent être extraits par le transformateur vers des classes d'objets état. Dans l'exemple, seules les classes DesAttributs et AutresAttributs sont identifiées dans les métadonnées pour être décomposées. Dans ces classes, seuls les attributs attribut_1, attribut_2 et attribut_4 sont identifiés pour être extraits vers des classes d'objet état. De même que pour l'opération de transformation de mélange présentée dans la section 7.2, différents composants peuvent être implantés pour offrir cette interface d'accès aux métadonnées. Notamment, un composant peut être implanté pour manipuler les graphes d'objets de métadonnées Speedo de persistance des objets, qui identifient entre autres les classes et attributs qui doivent donc être transformés pour être persistants.

Pour chaque classe décomposée, les attributs sont redéfinis dans une nouvelle classe d'objet état, qui est générée par l'opération de transformation. Dans notre exemple, l'opération de transformation représente les nouvelles classes compilées EtatDesAttributs et EtatAutresAttributs. Si la super-classe de la classe décomposée n'est pas elle-même décomposée, la classe est transformée pour implanter une interface générique d'accès à l'objet état réifiant l'état courant. Dans cet exemple, c'est une interface Java AccesEtat (inventée pour l'exemple), qui définit les méthodes retourneEtat() et modifieEtat(...). Ces méthodes accèdent à un attribut qui stocke l'objet état courant. Cet attribut est appelé etat dans notre exemple. Les implantations de ces méthodes d'accès sont générées automatiquement, de même que l'attribut référençant l'objet état, et tous les éléments des classes d'objets état. Toute classe d'objet état générée implante l'interface Java Etat, inventée pour notre exemple.

```
public class DesAttributs implements AccesEtat {
  private Etat etat;
  public Etat retourneEtat() { return etat; }
  public void modifieEtat(Etat e) { etat = e; }
  private EtatDesAttributs etatInitial;
  protected int attribut_3;

  public DesAttributs() {
    tv = this; ev = etatInitial;
    if (ev == null) { ev = new EtatDesAttributs();
                      etatInitial = ev; }
    cible = ...  /* this */;
    cibleEstThis = (cible == tv);
    super();
    if (cibleEstThis) {
      modifieEtat(ev);
    }
    cible = ...  /* this */;
    if (cible == tv) { ec = ev; } else {
      ec = (DesAttributs) cible.retourneEtat(); }
    ec.etat_attribut_1 = 1;
  }

  public DesAttributs(int i) {
    tv = this; ev = etatInitial;
    if (ev == null) { ev = new EtatDesAttributs();
                      etatInitial = ev; }
    cible = ...  /* this */;
    cibleEstThis = (cible == tv);
    super();
    if (cibleEstThis) {
      modifieEtat(ev);
    }
    cible = ...  /* this */;
    if (cible == tv) { ec = ev; } else {
      ec = (DesAttributs) cible.retourneEtat(); }
    ec.etat_attribut_1 = i;
  }

  public void affiche() {
    tv = this; ev = (EtatDesAttributs) retourneEtat();
    cible = ...  /* this */;
    if (cible == tv) { ec = ev; } else {
      ec = (DesAttributs) cible.retourneEtat(); }
    System.out.println(ec.etat_attribut_1);
  }
}
```

Prog. 7.11*c* : Classe DesAttributs après transformation

```
public class AutresAttributs extends DesAttributs {

  private int attribut_5;

  private EtatDesAttributs etatInitial;

  public AutresAttributs(int i) {
    tv = this; ev = etatInitial;
    if (ev == null) { ev = new EtatAutresAttributs();
                      etatInitial = ev;

    cible = ...  /* this */;
    if (cible == tv) { ec = ev; } else {
      ec = (AutresAttributs) cible.retourneEtat(); }
    ec.etat_attribut_4 = true;

    cible = ...  /* this */;
    cibleEstThis = (cible == tv);
    super(i);
    if (cibleEstThis) {
      ev.copieEtatDesAttributs(
        (EtatDesAttributs) retourneEtat());
      modifieEtat(ev);
    }

    cible = ...  /* this */;
    if (cible == tv) { ec = ev; } else {
      ec = (DesAttributs) cible.retourneEtat(); }
    ec.etat_attribut_3 = 4;
    attribut_5 = 12;
  }

  public AutresAttributs() {
    tv = this; ev = etatInitial;
    if (ev == null) { ev = new EtatAutresAttributs();
                      etatInitial = ev; }
    AutresAttributs(0);
  }

}
```

Prog. 7.11*d* : Classe AutresAttributs après transformation

```
public class UnClient {
  public void uneMethode(AutresAttributs obj) {
    cible = ...  /* this */;
    ec = (DesAttributs) cible.retourneEtat();
    System.out.println(ec.etat_attribut_1);
  }
}
```

Prog. 7.11*e* : Classe UnClient après transformation

```
public interface StateTransformerMetadata {
  boolean classMustBeTransformed(ObjectType classType);
  boolean fieldMustBeTransformed(ObjectType classType,
      String fieldName, FieldDescriptor fieldType);
}
```

Prog. 7.12 : Interface StateTransformerMetadata

Tous les noms des éléments générés sont paramétrables. En effet, le transformateur utilise une interface de type StateTransformer (cf. le programme 7.13), qui lui retourne les noms de méthodes d'accès, d'interface d'accès, d'interface d'objet état, etc. Ainsi, dans notre exemple, le composant de politique de nommage retourne les noms EtatAbstrait, Etat, etat, AccesEtat, retourneEtat, modifieEtat, copieEtat..., etc. Il est possible d'implanter un tel composant de politique de nommage pour retourner des noms spécifiques à Speedo. De cette manière, l'opération de transformation peut être adaptée à Speedo sans modifier le code du composant transformateur.

```
public interface StateTransformerNaming {
  String getStateFieldName();
  ObjectType getStateManipulationInterfaceType();
  String getStateAccessorMethodName();
  String getStateMutatorMethodName();
  ObjectType getStateAbstractType();
  String getStateCopyMethodName(ObjectType classType);
  String getTransformedFieldName(ObjectType classType,
      String fieldName, FieldDescriptor fieldType);
}
```

Prog. 7.13 : Interface StateTransformerNaming

7.3.2 Transformation des accès aux attributs et optimisation

Dans les classes décomposées et dans les classes clientes, les lectures et modifications d'attributs extraits, sont transformées en accès aux nouveaux attributs des objets état. Ainsi, le programme 7.11*e* donne le pseudo-code source d'une méthode transformée qui accède à l'attribut attribut_1 d'un objet de la classe DesAttributs. L'accès direct à l'attribut est transformé

en accès à l'attribut de l'objet EtatDesAttributs. Cet objet état est retourné par un appel à retourneEtat() sur l'objet DesAttributs accédé.

Une optimisation est effectuée par notre transformateur dans le cas où le code d'une méthode d'une classe décomposée accède à un attribut extrait de cette même classe. Dans chaque méthode d'une classe décomposée, du code est inséré en début de méthode pour charger l'objet état correspondant à l'objet « this », et le stocker dans une variable locale. Lors de l'accès à un attribut extrait de cette classe, la référence de l'objet cible de l'accès est comparée avec « this ». Si la cible de l'accès est « this », l'objet état utilisé est celui qui est stocké dans la variable locale. Sinon, il est récupéré en appelant la méthode d'accès à l'état de l'objet (retourneEtat() dans notre exemple). Si l'application est développée en applicant le principe de Masquage de l'Information, en encapsulant complètement les objets, les cas où les attributs accédés seront accédés sur « this » seront les plus courants. Notre optimisation permet donc d'économiser un appel de méthode dans la plupart des accès aux attributs, ce qui peut apporter un gain significatif. Cette optimisation est illustrée dans la méthode transformée affiche() de la classe DesAttributs, dans le programme 7.11*c* : l'accès à l'attribut attribut_1 de « this » est optimisé.

7.3.3 Transformation des initialiseurs d'instances

La principale difficulté que nous avons rencontrée dans l'implantation de cette transformation, est la transformation des initialiseurs d'instance des classes décomposées, pour la création des objets état associés aux objets dont ils réifient l'état.

Comme expliqué dans la section 7.2.2, une méthode d'initialisation d'instance (constructeur) est appelée alors que l'objet référencé par « this » n'est pas encore dans l'état « initialisé ». L'objet « this » ne devient initialisé qu'après avoir appelé l'initialiseur de la super-classe, avec « this » comme cible de cet appel. D'après la spécification de la JVM, la seule utilisation possible d'un objet non initialisé, est l'appel d'un initialiseur sur cet objet (instruction invokespecial). Cependant, un cas particulier dans la spécification est qu'une méthode d'initialisation peut accéder aux *attributs privés* (seulement) de « this » alors que « this » n'est pas encore initialisé.

Il est donc nécessaire de créer l'objet état dès le tout début de toute méthode initaliseur, pour prendre en compte le cas où un intialiseur d'une classe décomposée accède à un attribut privé, extrait vers une classe d'objet état, sur l'objet « this » non initialisé. Ce cas est illustré dans le programme 7.10 pour l'initialiseur de AutresAttributs, avec l'accès à l'attribut attribut_4. Ainsi, comme illustré dans les programmes 7.11*c* et 7.11*d*, les initialiseurs des classes DesAttributs et AutresAttributs sont transformés pour insérer du code qui crée un objet état dès le début de la méthode, et le stocker dans une variable locale. Tout accès à un attribut est transformé de la même manière que dans toute autre méthode, comme expliqué dans la section 7.3.2.

Cependant, un initialiseur d'une classe peut appeler un autre initialiseur de la même classe, pour initialiser « this ». Comme chaque appel d'initialiseur crée son propre objet état dans le contexte de cet appel, il reste à trouver un moyen de rendre cohérentes les modifications d'attributs réalisées dans chaque appel. Pour cela, il est nécessaire de garantir qu'un seul objet état est créé pour un objet, au niveau d'une classe donnée. C'est pourquoi nous avons choisi de stocker l'objet état créé au début de chaque initialiseur, dans un attribut privé qui n'est utilisé que pour l'initialisation des objets (appelés etatInitial dans notre exemple). Au début

de chaque initialiseur, le contenu de cet attribut est testé : s'il est null un nouvel objet état est créé et affecté à cet attribut, sinon l'objet état existant est utilisé tel quel. Ainsi, un seul objet état est créé pour chaque objet décomposé, au niveau d'une classe décomposée donnée. Cette « astuce » n'est possible justement que grâce au cas particulier de la spécification de la JVM, qui autorise un initialiseur à accéder aux attributs privés de l'objet « this » non initialisé.

Dès que l'objet « this » est initialisé, cet objet peut être passé en paramètre, et des méthodes peuvent être appelées sur cet objet, etc., Des accès à l'état de « this » peuvent donc être effectués, qui se traduisent par des appels à retourneEtat(). Il est donc nécessaire de transformer les initialiseurs pour insérer du code d'appel à la méthode modifieEtat(...) sur « this », avec l'objet état en paramètre, dès que « this » est initialisé, c'est-à-dire juste après l'appel à l'initialiseur de la super-classe sur « this ». Ainsi, la méthode retourneEtat() retourne toujours un objet état valide. Cette transformation est détaillée ci-dessous.

7.3.4 Prise en compte de l'héritage entre classes décomposées

Les relations d'héritage entre les classes décomposées se traduisent par des relations d'héritage entre les classes d'objet état générées. Ainsi, comme illustré dans le programme 7.11*b*, la classe générée EtatAutresAttributs hérite de la classe générée EtatDesAttributs, parce que la classe AutresAttributs hérite de la classe DesAttributs.

Le principale problème posé par l'héritage est celui de l'initialisation des objets état, dans les méthodes d'initialisation des classes décomposées. En effet, lorsqu'un objet de type AutresAttributs est créé, un initialiseur de la classe AutresAttributs est appelé, qui crée un objet état de type EtatAutresAttributs, puis qui appelle lui-même un initialiseur de la classe DesAttributs, qui crée à son tour un autre objet état de type EtatDesAttributs. Or, un objet ne doit avoir qu'un objet état : il faut donc fusionner les objets EtatAutresAttributs et EtatDesAttributs, pour obtenir un seul objet EtatAutresAttributs. Cette fusion est effectuée après chaque retour d'appel de l'initialiseur de la super-classe décomposée (DesAttributs dans l'exemple), dans chaque initialiseur de classe décomposée (AutresAttributs dans l'exemple). L'objet EtatDesAttributs est récupéré par un appel à retourneEtat(), et tous ses attributs sont copiés dans l'objet EtatAutresAttributs, en utilisant la méthode de copie que le transformateur a généré dans chaque classe d'objet état (méthodes copieEtatDesAttributs(...) et copieEtatAutresAttributs(...)). C'est pour pouvoir effectuer cette fusion que nous avons implanté la génération de ces méthodes de copie, dans le transformateur. L'objet résultant de la fusion (objet EtatAutresAttributs) est utilisé comme unique objet état pour l'objet « this », en appelant modifieEtat(...) avec cet objet.

Cette transformation des appels aux initialiseurs des super-classes est très similaire à la transformation effectuée par le transformateur de mélange, détaillée dans la section 7.2.2. Une difficulté supplémentaire ici, est que les méthodes d'initialisation peuvent avoir des arguments. Par conséquent, pour pouvoir tester si la cible d'un appel est « this », il est nécessaire de générer du code qui 1) commence par dépiler les arguments de l'appel, et les stocke dans de nouvelles variables locales, puis 2) compare la cible de l'appel avec « this », et 3) réempile les arguments pour pouvoir effectuer l'appel. Lorsque l'appel à l'initialiseur est effectué, si la cible de l'appel était « this », la fusion des objets état est effectuée, et seulement dans ce cas. Ce code est illustré par le pseudo-code source du programme 7.11*d*.

7.3.5 Synthèse

L'opération de transformation pour la réification de l'état permet de décomposer automatiquement une classe, en extrayant certains attributs des classes, vers des classes d'objets état. Cette opération de transformation est complètement paramétrable, pour les classes à décomposer et les attributs à extraire, et aussi pour les noms des éléments générés dans les classes résultant d'une transformation. Cette opération est réutilisable indépendamment de Speedo. Notre opération de transformation prend en compte les cas particuliers de la spécification de la JVM, et notamment la possibilité d'accéder aux attributs privés sur les objets non initialisés, dans un soucis de généralité en accord avec notre principe 4 de Généralité du Système de Transformation.

Une originalité de cette opération de transformation, qui a rendu son implantation difficile, est qu'elle transforme chaque classe en un seul balayage, pour optimiser les temps de transformation. C'est-à-dire que les classes d'objets état, les méthodes de copie dans ces classes, les accès aux attributs extraits, les initialiseurs dans les classes décomposés, etc. sont tous générés / transformés « en parallèle », lors d'un seul balayage de chaque classe. Le code source Java complet du composant transformateur et des interfaces de paramétrage prend 2217 lignes de code commenté.

7.3.6 Perspectives : état transactionnel

L'utilité de cette opération de transformation dans Speedo apparaît clairement pour la gestion optimiste de transactions. Dans ce contexte, le gestionnaire de persistance Speedo doit gérer un état pour chaque objet, et pour chaque transaction qui « traverse » un objet. Une telle gestion d'état transactionnel est facilement réalisable en implantant une opération de transformation complémentaire à l'opération décrite dans cette section. Il suffit de transformer la méthode d'accès à l'objet état d'un objet (retourneEtat() dans l'exemple), pour ne plus retourner le contenu de l'attribut etat de l'objet, mais pour retourner un objet état associé à la transaction liée à l'activité (*thread*) qui appelle cette méthode. Il n'est pas nécessaire de transformer la méthode de modification de l'objet état (modifieEtat(...) dans l'exemple), parce que cette méthode est appelée seulement dans les méthodes d'initialisation des objets, et n'est jamais appelée dans un autre contexte. Nous n'avons pas encore implanté cette opération de transformation complémentaire, mais nous prévoyons qu'elle sera facilement implantable.

7.4 Conclusion

Ce chapitre décrit en détails deux opérations de transformation Jabyce, utilisables pour lier des applications avec le service technique de persistance Speedo. Les transformations implantées sont complexes, surtout parce qu'elles nécessitent de prendre en compte de nombreux détails subtils de la spécification de la JVM. De plus, nous avons développé ces opérations de transformation pour minimiser les temps de transformation. C'est pourquoi les classes sont transformées en un seul balayage, ce qui augmente la complexité de l'implantation. Cette complexité n'est pas due à l'utilisation de Jabyce pour la conception : des transformateurs équivalents développés avec ASM seraient aussi complexes.

En revanche, l'utilisation du modèle de composant Fractal dans Jabyce permet de contrôler facilement le paramétrage et la configuration des transformateurs développés, sans nécessiter de modifier l'implantation. En effet, nous avons conçu nos opérations de transformation selon le patron de conception Stratégie, pour pouvoir paramétrer les classes à transformer, la stratégie de nommage des éléments générés, etc., par de simples liaisons entre les opérations et des composants de paramétrage, tels que des composants « conteneurs » de méta-données. Le paramétrage est facilité, notamment par les outils autour du langage de description d'architecture (ADL) de Fractal.

Plus originalement, comme montré dans la section 7.2 pour notre opération de mélange de classes, l'utilisation de Fractal permet de configurer des opérations de transformation récursives complexes. En effet, Fractal permet de configurer facilement des architectures réellement *fractales*, avec des opérations de transformation similaires contenues l'une dans l'autre. Une telle architecture est plus difficile à concevoir avec un système de transformation non basé sur un modèle de composant hiérarchique, tel que ASM par exemple.

Le chapitre 8 traite également de la problématique de la configuration, dans la continuité de ce chapitre, mais à une échelle plus large pour la conception de tisseurs. Le chapitre 8 traite surtout de la composition de tisseurs pour lier des applications à plusieurs services techniques.

Chapitre 8

Construction adaptable d'un tisseur de lien fonctionnel / technique

Ce chapitre aborde la problématique de l'architecture des *tisseurs*, c'est-à-dire des logiciels qui construisent un lien entre applications et services techniques, à une plus large échelle que l'architecture des opérations de transformation qui peuvent constituer les tisseurs. Plus précisément, deux problèmes sont abordés : 1) la conception de tisseurs qui nécessitent de combiner plusieurs techniques et mécanismes (compilation de code source, plusieurs passes de transformation de classes compilées Java, etc.), et 2) la composition de plusieurs tisseurs, pour obtenir par exemple un seul tisseur qui construit un lien entre une application et plusieurs services techniques. Ce chapitre décrit notre proposition de démarche et d'outils de conception de tisseurs, pour résoudre ces problèmes.

Cette proposition s'appuie sur nos expérimentations présentées dans le chapitre 7, pour la conception d'opérations de transformation de classes compilées Java utilisables pour lier des applications avec Speedo, un service technique de persistance. Comme décrit dans la section 7.1, le tisseur de Speedo s'appuie sur plusieurs techniques : transformation de code compilé, génération de code source, etc. Bien que le service Speedo soit lui-même conçu avec des composants Fractal, conformément à la démarche générale de notre laboratoire (cf. la section 2.3), le tisseur de Speedo est pour l'instant développé directement en Java. L'architecture de ce logiciel est donc cachée dans le code Java, ce qui rend difficile l'évolution de ce logiciel. Surtout, ceci limite la réutilisation des éléments du tisseur de Speedo pour le composer avec les éléments d'autres tisseurs d'autres services techniques, tels que la démarcation de transactions, la sécurité (contrôle d'accès), la duplication, etc.

La section 8.1 présente un modèle en flot de données (*data-flow*) d'un tisseur. En se basant sur cette modélisation, les sections 8.2 et 8.3 présentent notre proposition de démarche de conception et d'architecture d'un tisseur. Les sections 8.4 et 8.5 présentent notre proposition de formalisme de spécification d'architecture de tisseur, qui est une extension du langage ADL de Fractal, ainsi qu'un prototype de génération automatique d'ordonnanceur de tâches qui se base sur de telles spécifications. La section 8.6 montre l'intérêt de notre proposition pour la composition de tisseurs pour différents services techniques.

8.1 Un modèle de flot de données

Comme évoqué dans la section 7.1, Speedo utilise différentes techniques pour établir un lien entre une application et le service de persistance : génération de code source pour des classes utilitaires, et transformation de classes compilées Java. De plus, dans les fichiers de configuration JDO il est possible de ne pas donner toute l'information sur les classes et attributs qui doivent être persistants. Dans ce cas, un tisseur JDO doit balayer les classes compilées pour analyser les éléments des classes persistantes, et ainsi compléter les méta-données de persistance. Il faut composer ces différentes techniques pour construire un lien complet entre une application et un service de persistance Speedo. En généralisant, nous considérons que tous les tisseurs reposent sur une composition de différentes techniques.

Les traitements effectués par le tisseur de Speedo peuvent être décrits naturellement par le diagramme de *flot de données* (*data-flow*) donné dans la figure 8.1. Les ovales représentent les *tâches*, et les rectangles représentent les *ressources* consommées ou produites par les tâches.

Les données en entrée sont 1) les classes Java compilées de l'application à rendre persistante, et 2) des fichiers de paramétrage de la projection des objets persistants, dans un format spécifique au standard JDO. Les différentes tâches réalisées sur ces données sont :

1. analyse lexicale et syntaxique des fichiers de paramétrage du service de persistance fournis par l'utilisateur, pour construire *un graphe d'objets de méta-information* ;
2. les fichiers de paramétrage peuvent être incomplets (la notion de « paramétrage par défaut » existe dans la spécification de JDO). Ceci impose d'*analyser les classes compilées* fournies par l'utilisateur, pour déterminer les noms et types des attributs persistants s'ils ne sont pas spécifiés dans les fichiers de paramétrage. Cette analyse peut être effectuée par exemple par une opération de transformation Jabyce. L'opération de transformation complète le graphe d'objets de méta-information, en fonction des éléments présents dans les classes de l'application ;
3. des transformations « légères » sont appliquées aux classes de l'application, *pour permettre de compiler les classes* qui vont y être mélangées : changement des modificateurs d'accès aux attributs (qui deviennent public), génération de méthodes « factices » pour l'accès aux attributs (méthodes « get » et « set »), etc.
4. génération du code source de classes utilitaires spécifiques au canevas logiciel de persistance JORM ;
5. génération du code source de classes utilitaires spécifiques à Speedo, qui spécialisent les classes générées par JORM ;
6. compilation du code source des classes utilitaires générées par JORM et Speedo ;
7. mélange des classes utilitaires compilées dans les classes compilées de l'application (cf. la section 7.2), puis transformation pour la décomposition des classes (réification de l'état des objets, cf. la section 7.3), puis application de transformations « légères » : rétablissement des modificateurs d'accès aux attributs, etc.

Les données en sortie sont les classes compilées de l'application, qui ont été transformées pour que les objets soient persistants.

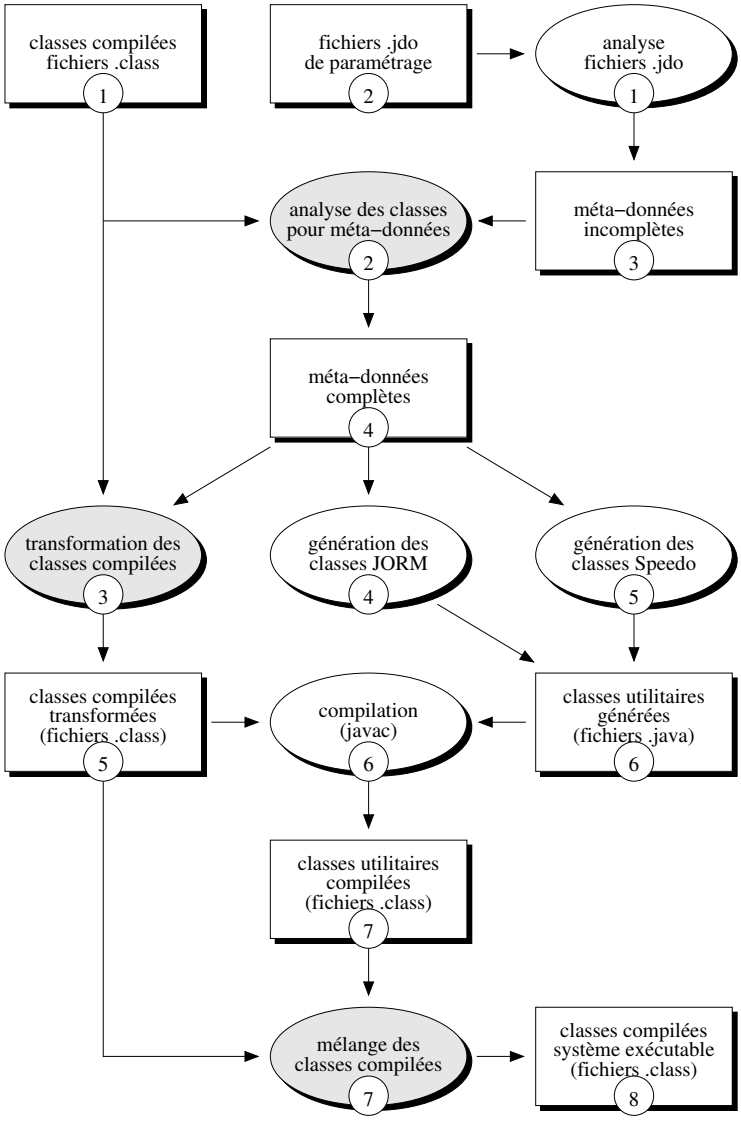


FIG. 8.1 : Diagramme de flot de données de l'exécution du tisseur de Speedo

Les tâches 2, 3 et 7 peuvent être implantées par des chaînes d'opérations de transformation Jabyce. Les tâches 2 et 3 peuvent être réalisées en implantant deux opérations de transformation Jabyce qui sont composées dans une même chaîne. En effet, ces deux tâches peuvent être effectuées en un seul balayage des classes de l'application. Cependant, pour simplifier, nous considérons dans la suite que ces deux opérations de transformation sont effectuées dans des tâches différentes, c'est-à-dire composées dans des chaînes d'opérations différentes.

8.2 Conception par composants Fractal

Pour maîtriser l'architecture des tisseurs, il est nécessaire de les modulariser, et de séparer la configuration d'un tisseur de son implantation, en application du principe 3 de Séparation Configuration / Utilisation. C'est pourquoi la base de notre proposition est de concevoir *les tâches et les ressources comme des composants Fractal* bien encapsulés, et réutilisables pour construire des liens fonctionnel / technique dans d'autres contextes. La figure 8.2 illustre l'architecture que nous proposons pour le tisseur de Speedo.

Nous proposons de concevoir un tisseur avec deux types de composants : des *tâches*, et des *ressources*. Ainsi, un diagramme de flot de données d'un tisseur, tel que celui donné dans la figure 8.1 pour Speedo, se traduit directement dans l'architecture du tisseur. Un exemple de composant ressource, directement réutilisable dans une telle architecture, est un composant Jabyce FilesystemRepository qui offre un accès à un ensemble de classes compilées sérialisées, comme illustré dans la figure 6.3 dans la section 6.3. Les composants ressource sont passifs, et n'initient aucune interaction. Toutes les liaisons sont donc établies entre les interfaces client des composants tâche, et les interfaces serveur des composants ressource.

Un composant tâche peut initier des interactions sur une liaison, soit pour consommer une ressource (récupérer des données à partir du composant ressource), soit pour produire une ressource (envoyer des données au composant ressource). Pour simplifier la conception, nous spécifions que lors de l'exécution d'une tâche, celle-ci consomme toutes les données de ses *ressources d'entrée*, pour produire toutes les données dans ses *ressources de sortie*. Par exemple, la tâche 6 compile tous les fichiers source Java contenus dans sa ressource d'entrée, pour produire tous les fichiers de classes compilées Java dans sa ressource de sortie, en une seule exécution.

8.3 Ordonnancement

Les composants tâche échangent des données à travers les composants ressource. Par exemple, la tâche 6 de compilation de code source compile les fichiers source Java qui sont contenus dans le composant ressource 6. Ces fichiers source sont produits par les tâches 4 et 5. Se pose alors le problème de l'ordonnancement des exécutions de tâches. Par exemple, la tâche 6 ne peut être exécutée qu'après que les tâches 4 et 5 aient été exécutées. L'enjeu de l'ordonnancement consiste à exécuter des tâches en respectant les contraintes des consommations et productions de ressources : une ressource peut être consommée seulement après avoir été produite.

Certaines tâches peuvent être exécutées en parallèle, c'est-à-dire dans n'importe quel ordre. Par exemple, les tâches 4 et 5 peuvent être exécutées en parallèle.

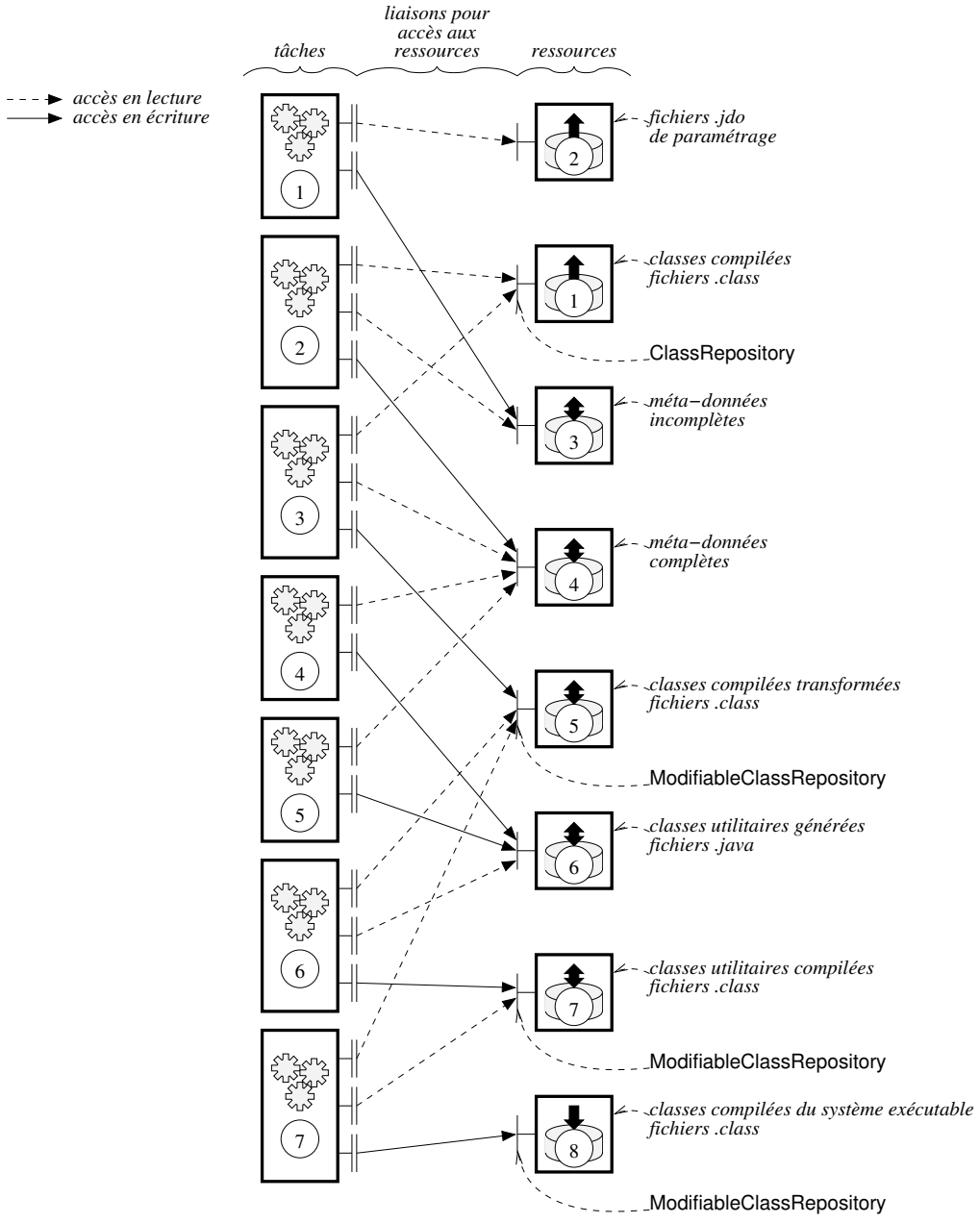


FIG. 8.2 : Conception par composants des tâches et des ressources dans un tisseur

Approche simpliste : *multi-thread Java*

Une approche simpliste de l'ordonnancement des tâches est d'exécuter chaque tâche dans une activité (*thread* Java), toutes les tâches s'exécutant en concurrence. L'ordonnancement des tâches est réalisé en synchronisant les activités des différentes tâches. Dans cette approche, tous les composants participent à la synchronisation en échangeant des messages de synchronisation. Les tâches communiquent entre elles seulement à travers les ressources. Chaque composant resource doit donc offrir une nouvelle interface de synchronisation, qui définit par exemple des méthodes waitConsumeResource() et notifyResourceProduced(). Les implantations de ces méthodes, dans chaque composant ressource, s'appuient sur les mécanismes de synchronisation de Java : les moniteurs et les méthodes wait() et notify() de la classe Object. D'un point de vue général, les liaisons entre les composants tâche et les composants ressource sont utilisées non seulement pour communiquer des données avec les ressources, mais aussi pour échanger des messages de synchronisation.

Inversion de contrôle

Cependant, pour simplifier la conception des composants tâche et des composants ressource, nous proposons de ne pas imposer que les composants soient développés en se basant explicitement sur les primitives de synchronisation de Java. En effet, en application du principe 2 d'Inversion de Contrôle, nous proposons que les composants soient développés indépendamment des préoccupations du contrôle du séquencement des actions. Par exemple, l'implantation de composant FilesystemRepository, fournie dans Jabyce pour le stockage de fichiers .class, n'utilise aucune primitive de synchronisation de Java, et n'est donc pas utilisable en environnement concurrent sans modification du code. L'application du principe 2 d'Inversion de Contrôle facilite ainsi la réutilisation de composants existants. En conséquence, l'ordonnancement n'est pas mis en œuvre par une exécution concurrente des tâches, c'est-à-dire en exécutant plusieurs tâches dans des activités concurrentes, mais comme un choix non déterministe d'une séquence d'exécutions de tâches qui respecte les contraintes des consommations et productions des ressources.

D'un point de vue architectural, nous proposons que chaque composant tâche offre une interface de type Runnable, dont la méthode run() est appelée pour exécuter la tâche. Les différentes tâches sont déclenchées par un composant *ordonnanceur*, qui est lié à l'interface de type Runnable de chaque composant tâche. Ce composant offre lui-même une interface de type Runnable, pour exécuter la séquence de toutes les tâches, en respectant les contraintes des consommations et productions des ressources. Nous proposons, comme architecture globale d'un *tisseur*, celle d'un composant composite, contenant tous les composants tâche et ressource, et le composant ordonnanceur, et qui « exporte » l'interface de type Runnable du composant ordonnanceur. Cette architecture est illustrée dans la figure 8.3. L'avantage de cette architecture est qu'elle est directement exécutable par les outils ADL de Fractal, comme décrit dans la section 6.3.

L'avantage de concevoir les tâches comme des composants Fractal offrant une interface de type Runnable, est de pouvoir réutiliser directement des applications existantes, avec des modifications mineures de configuration, mais surtout sans modifier le code des composants. Notamment, notre application d'exemple, présentée dans la section 6.3 et illustrée dans la section 6.3, peut être réutilisée directement, en nécessitant seulement des modifications mineures dans la

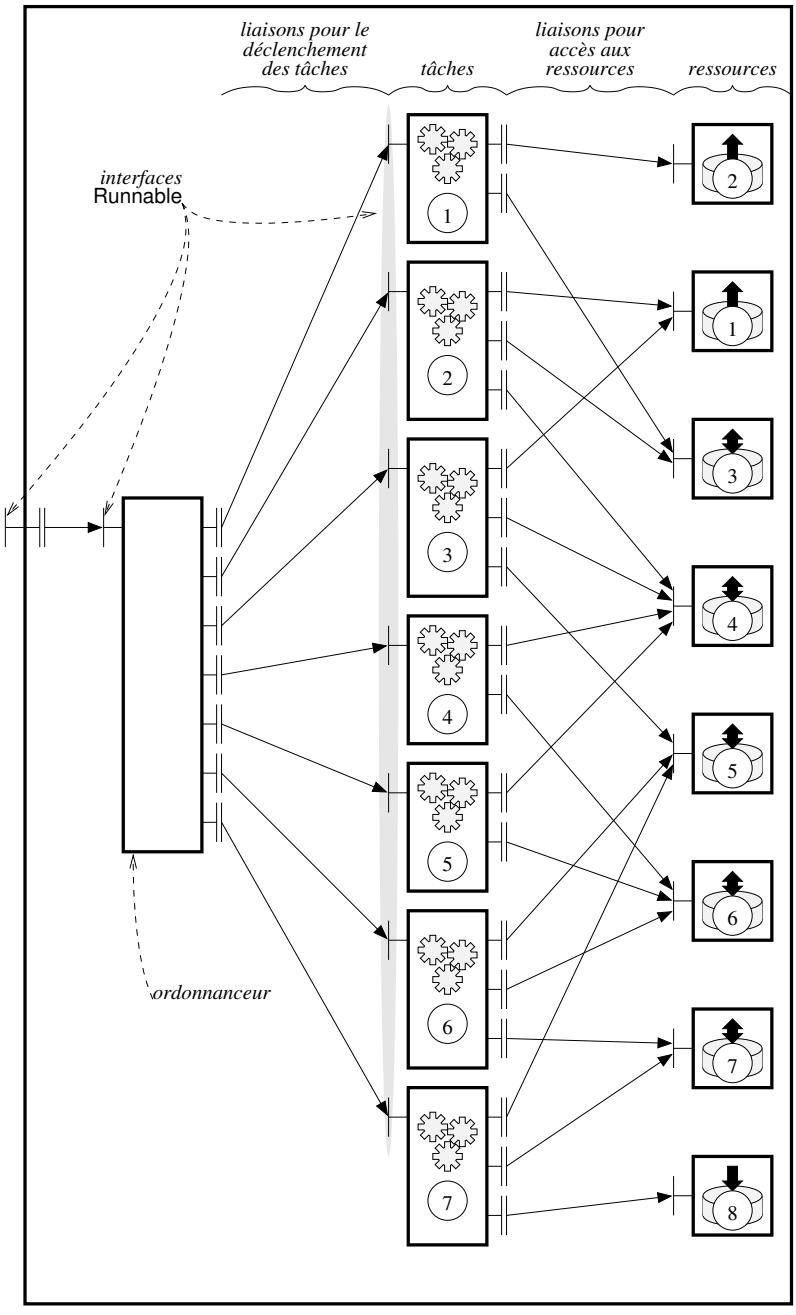


FIG. 8.3 : Proposition d'architecture globale du tisseur de Speedo

configuration (les composants « conteneurs » de classes compilées doivent être sortis du composant composite, et devenir des composants ressource). Cette application exécute une chaîne d'opérations de transformation Jabyce pour un ensemble de classes. Cette application peut donc facilement être adaptée pour concevoir les tâches 2, 3, et 8, en composant les opérations de transformation adéquates. Notamment, la chaîne d'opérations dans le composant tâche 8 contiendra essentiellement les opérations de mélange et de décomposition des classes, présentées dans le chapitre 7.

8.4 Spécification de flot de données

Dans notre architecture, la difficulté de l'ordonnancement est reportée sur l'implantation du composant ordonnanceur. Une approche simpliste consiste à implanter « à la main » cet ordonnanceur comme un composant primitif, en codant explicitement la séquence d'exécution des tâches. Les problèmes avec cette approche sont : 1) la difficulté de déterminer « à la main » un ordonnancement correct pour une configuration complexe, et 2) la nécessité de modifier le code de l'ordonnanceur lorsque la configuration est modifiée.

Formalisme de spécification

C'est pourquoi nous proposons de générer automatiquement les implantations de composants ordonnanceur, ou d'implanter des composants ordonnanceur paramétrables. Nous ne souhaitons pas imposer d'implantation particulière de composant ordonnanceur, ni même de mécanisme de paramétrage ou de génération automatique. En revanche, nous proposons la spécification d'un formalisme général de définition de flots de données, qui permet d'identifier les composants ressource et tâche dans une architecture, et leurs relations. Ce formalisme est une extension simple du langage ADL de Fractal, pour définir le nouveau type d'élément <dataflowtask/>. Cette extension sera facilement mise en œuvre en utilisant les mécanismes d'extension (API et outils) fournis dans Fractal ADL. Un exemple de spécification partielle pour le tisseur Speedo est donnée dans la figure 8.4. Les éléments <dataflowtask/> servent à identifier les relations de consommation et de production de ressources, entre les interfaces client des composants tâche, et les interfaces serveur des composant ressource. Après traitement, cette spécification peut être transformée automatiquement en la spécification donnée dans la figure 8.5, dans le langage ADL standard de Fractal. Cette extension simple du format ADL de Fractal est nécessaire pour pouvoir identifier les relations de consommation et de production de ressources. En effet, avec le langage ADL standard (cf. la figure 8.5), il n'est pas possible de distinguer les liaisons de consommation et les liaisons de production parmi les définitions de liaisons.

Cette identification des relations de consommation et de production de ressource est nécessaire et suffisante pour ordonnancer l'exécution des tâches, et donc pour générer ou paramétrer un composant ordonnanceur.

```
<definition name="ExempleTisseur">
  <component definition="AnalyseFichiersParam" name="tache1"/>
  <component definition="AnalyseClasses" name="tache2"/>
  <component definition="ClassesOriginales" name="ressource1"/>
  <component definition="FichiersParam" name="ressource2"/>
  <component definition="Metadonnees" name="ressource3"/>
  <component definition="Metadonnees" name="ressource4"/>
  <!-- etc. -->
  <dataflowtask name="tache1">
    <input interface="fichiersParam"
      resource="ressource2.fichiersParam"/>
    <output interface="metadonnees"
      resource="ressource3.metadonnees"/>
  </dataflowtask>
  <dataflowtask name="tache2">
    <input interface="classes"
      resource="ressource1.classRepository"/>
    <input interface="metadonnees"
      resource="ressource3.metadonnees"/>
    <output interface="metadonneesCompl"
      resource="ressource4.metadonnees"/>
  </dataflowtask>
  <!-- etc. -->
</definition>
```

Fig. 8.4 : Spécification ADL de l'architecture du tisseur de Speedo, en utilisant une extension de l'ADL

```
<definition name="ExempleTisseur">
  <component definition="AnalyseFichiersParam" name="tache1"/>
  <component definition="AnalyseClasses" name="tache2"/>
  <component definition="ClassesOriginales" name="ressource1"/>
  <component definition="FichiersParam" name="ressource2"/>
  <component definition="Metadonnees" name="ressource3"/>
  <component definition="Metadonnees" name="ressource4"/>
  <!-- etc. -->
  <binding client="tache1.fichiersParam"
    server="ressource2.fichiersParam"/>
  <binding client="tache1.metadonnees"
    server="ressource3.metadonnees"/>
  <binding client="tache2.classes"
    server="ressource1.classRepository"/>
  <binding client="tache2.metadonnees"
    server="ressource3.metadonnees"/>
  <binding client="tache2.metadonneesComp"
    server="ressource4.metadonnees"/>
  <!-- etc. -->
</definition>
```

Fig. 8.5 : Spécification ADL de l'architecture du tisseur de Speedo, après transformation

8.5 Ordonnanceur implanté en Erlang

Pour valider notre approche de génération / paramétrage d'un composant ordonnanceur, nous avons développé un prototype dans le langage Erlang. Ce langage est présenté dans la section 4.1.3. Erlang est basé sur un calcul de processus : l'abstraction de base est le *processus*. Les processus Erlang échangent des messages pour offrir des services. L'idée de base de notre prototype est de modéliser les tâches et les ressources qui sont présentes dans l'architecture d'un tisseur, seulement du point de vue de la synchronisation des traitements. Cette utilisation d'Erlang pour le contrôle des applications est l'une des utilisations classiques d'Erlang [R03].

Dans notre prototype, l'ordonnanceur est un composant Fractal primitif, qui gère une instance de machine virtuelle Erlang. Cette machine virtuelle Erlang exécute un ensemble de processus Erlang en parallèle, où chaque processus modélise une tâche ou une ressource. Ces processus échangent des messages qui servent seulement à la synchronisation des traitements. Ces messages sont propagés seulement à l'intérieur de la machine virtuelle Erlang, et pas vers l'extérieur. Erlang supporte un grand nombre de processus dans une même machine virtuelle Erlang : le nombre de tâches et de ressources qui peuvent être modélisés dans un ordonnanceur n'est donc pas limité. Les types de messages de synchronisation échangés sont :

- produce (tâche → ressource) : la ressource a été produite par la tâche ;
- consume (ressource → tâche) : la ressource peut être consommée par la tâche ;
- run (tâche → ordonnanceur) : lance l'exécution d'un composant tâche.

L'algorithme d'un processus ressource est le suivant :

1. attendre un message produce de chaque processus tâche producteur ;
2. émettre un message consume à chaque processus tâche consommateur ;
3. terminer.

L'algorithme d'un processus tâche est le suivant :

1. attendre un message consume de chaque processus ressource consommée ;
2. émettre un message run au processus ordonnanceur ;
3. émettre un message produce à chaque processus ressource produite ;
4. terminer.

Le composant ordonnanceur interagit avec la machine virtuelle Erlang qu'il encapsule, en utilisant les mécanismes de files de messages réparties offerts par Erlang. Lorsque la méthode run() est appelée sur l'interface serveur de type Runnable du composant ordonnanceur, ce composant démarre les processus modélisant les tâches et les ressources, en émettant des messages vers la machine virtuelle Erlang à travers la file de messages, puis attend la réception de messages run dans la file. Pour chaque message run reçu dans la file, le composant ordonnanceur exécute la méthode run(), sur l'interface client de type Runnable qui correspond à la tâche identifiée dans le message. Les messages run sont traités dans l'ordre, pour respecter l'ordonnancement déterminé par l'ordonnancement des processus Erlang. Lorsque tous les messages run ont été traités, l'appel à run() sur le composant ordonnanceur se termine. Cette architecture est illustrée dans la figure 8.6.

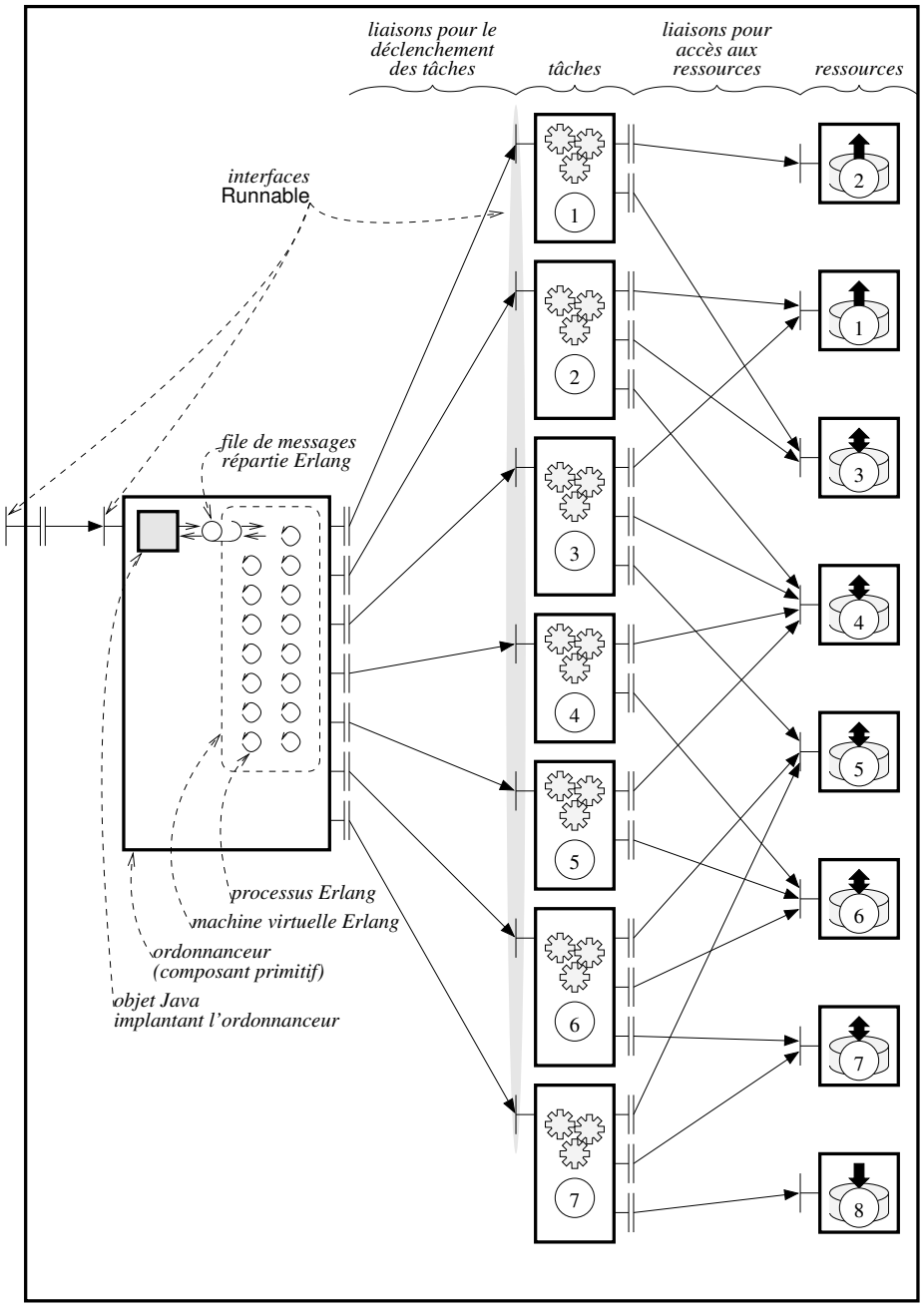


FIG. 8.6 : Ordonnanceur de tâche implanté en Erlang

L'apport de notre approche est d'offrir une véritable séparation de la préoccupation de l'ordonnancement, de l'implantation des composants tâche et ressource. Le système est simultanément représenté selon deux points de vue : 1) du point de vue architectural, sous la forme de composants Fractal, et 2) du point de vue de l'ordonnancement, sous la forme de processus Erlang. Ces deux représentations sont complémentaires et symétriques. La symétrie est maintenue par les outils automatique que nous envisageons de développer, qui se basent sur une description unique du système (avec notre extension de l'ADL, cf. la figure 8.4), à la fois 1) pour établir les liaisons entre composants du point de vue architectural, et 2) pour créer et configurer les processus Erlang du point de vue de l'ordonnancement.

L'utilisation d'Erlang est une solution techniquement viable. En effet, Erlang est effectivement utilisé en production dans l'industrie des télécommunications, et la machine virtuelle Erlang est stable et robuste et consomme peu de ressources comparée par exemple à une machine virtuelle Java. De plus, l'utilisation d'Erlang sera complètement cachée aux utilisateurs, en encapsulant une machine virtuelle Erlang dans un composant primitif ordonnanceur, et en offrant des interfaces abstraites qui masquent la complexité de l'implantation. A l'heure actuelle, l'implantation en Erlang de notre prototype est complète. Ce prototype pourra donc être utilisé en pratique, et n'est pas limité à une « preuve de concept ». En revanche, l'intégration de la machine virtuelle Erlang dans un composant Fractal / Java n'est pas encore réalisée. Pour l'instant, nous simulons le système en implantant le « processus ordonnanceur » en Erlang pour afficher une trace d'exécution. Nous avons déjà implanté en Erlang les fonctions de création et de configuration des processus, à partir d'une description des tâches dans la même forme que celle de la figure 8.4. C'est pourquoi l'intégration avec Java sera réalisée facilement.

8.6 Composition de tisseurs pour différents services techniques

Grâce à l'architecture et les outils (extensions de l'ADL Fractal...) que nous proposons, il est facile de composer plusieurs tisseurs qui construisent chacun un lien entre des applications et un service technique, pour former un seul tisseur qui construit un lien entre des applications et l'ensemble des services techniques.

Il suffit de définir un composant composite, en concaténant (« copier-coller ») les définitions des différents tisseurs dans notre extension de l'ADL (p.ex. la définition donnée dans la figure 8.4 pour le tisseur de Speedo). Ainsi, l'architecture résultante sera celle d'un composant composite, selon l'architecture illustrée dans la figure 8.3, qui contient tous les composants ressource et tâche de tous les tisseurs qui sont composés. La composition est ensuite effectuée en redéfinissant la relation entre les tâches et les ressources, pour que certaines tâches utilisent des ressources définies dans d'autres tisseurs.

D'autres travaux du laboratoire DTL/ASR se sont focalisés sur la problématique complémentaire de l'ordonnancement de l'exécution de différents services, pendant l'exécution d'une application [Bea03, BC02]. Ces travaux définissent que deux services techniques sont *non orthogonaux*, lorsque les séquences d'exécution de ces deux services techniques ne sont pas parallèles, c'est-à-dire lorsque ces séquences d'exécution ne peuvent pas être entrelacées de manière arbitraire. Ces travaux proposent de concevoir les services techniques sous la forme d'automates réactifs, qui émettent et reçoivent des messages, et qui s'exécutent en parallèle. La composition des services techniques est alors vue comme un produit d'automates. Des

contraintes d'entrelacement des séquences d'exécution sont exprimées par un utilisateur sous la forme de barrières de synchronisation entre plusieurs automates. Les automates et les contraintes de synchronisation sont spécifiés par programmation, en utilisant les interfaces de programmation du système Brenda, qui a été développé dans le cadre de ces travaux. Brenda traduit les automates et les contraintes de synchronisation dans un langage réactif, tel que Esterel, qui peut ensuite être compilé en Java pour générer l'implantation d'un composant Fractal / Java optimisé. Cette compilation est réalisée en utilisant un compilateur Esterel vers Java qui a également été développé dans le laboratoire. Le choix d'un ordonnancement qui respecte les contraintes de synchronisation est réalisé statiquement, lors de la compilation Esterel vers Java. Notre approche pour la composition des tisseurs pour différents services techniques rejoint ces travaux, en appliquant la même idée de réduire la problématique de la composition à une problématique d'ordonnancement, dans laquelle la composition d'automates est remplacée par la synchronisation de processus Erlang. Cependant, le contexte d'application est différent : la problématique du tissage du lien fonctionnel / technique n'est pas pris en compte dans Brenda, alors que c'est la problématique que nous avons principalement abordée.

8.7 Synthèse

Ce chapitre présente notre proposition pour la conception de *tisseurs*, c'est-à-dire de logiciels qui construisent un lien entre applications et services techniques. La base de notre proposition est une modélisation par flot de données (*data-flow*) d'un tisseur, en termes de tâches et de ressources, et de consommation et de production de ressources. Notre proposition d'architecture se base sur le modèle de composant Fractal et sur cette modélisation en flots de données, avec une conception des tâches et des ressources par des composants Fractal. Pour traiter le problème de l'ordonnancement des tâches, nous avons proposé une extension du langage ADL de Fractal, qui permet de spécifier explicitement les composants tâche et les composants ressource dans l'architecture d'un tisseur. A partir d'une telle spécification, des outils peuvent générer ou paramétrer automatiquement un composant ordonnanceur, pour déclencher l'exécution des composants tâche en respectant les relations de consommation / production des ressources.

Ces outils n'ont pas encore été implantés pour l'instant. Il reste également à appliquer notre démarche pour ré-architecturer le tisseur du service Speedo de persistance transparente de objets. Cependant, des chaînes d'opérations de transformation Jabyce existantes sont directement utilisables en tant que composants tâche dans une telle architecture. De plus, nous avons implanté un prototype avancé de composant ordonnanceur, en Erlang. Nous estimons donc que l'effort pour aboutir à un prototype fonctionnel complet de tisseur pour Speedo sera peu élevé.

Chapitre 9

Compilation Java vers C dans le système Jivaro

Ce chapitre présente des travaux effectués dans le laboratoire DTL/ASR de France Telecom, pour développer un système d'exploitation complet basé sur des composants, et pour développer une machine virtuelle Java, au moyen de composants logiciels, située directement au niveau d'un système d'exploitation (concept de « Java OS »). Le projet Think, développé conjointement avec le projet SARDES de l'INRIA, est une implantation du modèle de composant Fractal en C, pour la construction de systèmes d'exploitation [FSLM02]. Les travaux décrits dans ce chapitre utilisent Think, pour développer une machine virtuelle Java sur un système d'exploitation spécialisé et minimal [TFL+04]. L'implantation de cette machine virtuelle se base sur une compilation des classes Java vers un code C. Cette compilation s'effectue de manière statique, avant l'exécution du système. Ce compilateur est basé sur Jabyce. Ce chapitre décrit ce compilateur Java vers C.

Les sections 9.1 et 9.2 présentent respectivement le système Think et le système Jivaro. La section 9.3 présente en détails la conception du compilateur Java vers C qui constitue l'essentiel du système Jivaro. La section 9.4 montre l'intérêt de l'architecture de ce compilateur, pour exécuter des opérations de transformation de tisseurs (cf. le chapitre 8), de manière efficace lors de la compilation d'une application Java. La section 9.5 présente les mécanismes qui peuvent être mis en œuvre dans Jivaro pour effectuer la vérification des applications Java pendant leur compilation et leur exécution. La section 9.6 présente des perspectives de ce travail, qui concernent essentiellement la modularité de l'architecture du compilateur Jivaro.

9.1 Le canevas logiciel Think

Think est une implantation du modèle de composant Fractal, décrit dans la section 3.2.1, pour la conception de systèmes d'exploitation. Think est principalement décrit dans [FSLM02]. Think n'impose pas d'architecture prédéfinie, et a été utilisé pour développer des systèmes d'exploitation de natures très différentes : exo-noyaux, micro-noyaux, noyaux monolithiques classiques, systèmes spécialisés pour des applications (p.ex. le jeu vidéo Doom), systèmes d'exploitation Java, etc. Think est une implantation du modèle de composants qui permet de

développer des composants Fractal à une granularité arbitrairement petite, avec un environnement d'exécution minimal.

Think est complété par Kortex, une librairie d'implantations de composants Fractal / Think. Les composants de Kortex peuvent être composés librement en établissant des liaisons entre des composants pour construire le système d'exploitation conforme aux contraintes matérielles et logicielles. Les composants de base implantés dans Kortex réifient les ressources matérielles : CPU, gestionnaire de mémoire (MMU), et gestion des exceptions / interruptions. Sont également disponibles différentes implantations de composants de gestion de mémoire (modèle *plat* ou *paginé*, etc), de composants d'ordonnancement de tâches (*coopératif*, *round-robin*, avec *priorités*, etc.), de composants de systèmes de fichier, pour la communication répartie, etc. Think a été porté sur les architectures PowerPC, Intel x86 et ARM, et des composants Kortex ont été développés spécifiquement pour chaque architecture. Des logiciels ont été développés autour de Think, notamment pour l'assemblage statique et efficace des composants d'un système.

Une motivation importante dans la conception de Think, est de permettre de spécialiser un système d'exploitation pour des applications particulières, dans un objectif de performances, de simplicité, et de sécurité. Plus généralement, nous considérons que l'objectif de Think est de « casser » la notion de *couche logicielle*, ou de *machine abstraite*. La conception des systèmes d'exploitation comme une hiérarchie de machines abstraites est très classique [Dij68, Dij83]. Cependant, une telle architecture peut être inefficace car inadaptée aux applications. Les approches telles que les exo-noyau [EKJ95] proposent de résoudre ce problème, en offrant comme seule machine abstraite des abstractions minimales telles que l'ordonnancement des tâches et l'isolation mémoire. Cependant, par soucis d'abstraction par rapport au matériel, ces machines abstraites imposent des mécanismes d'accès aux ressources matérielles qui peuvent se révéler inadaptés aux applications et aux ressources matérielles elles-mêmes. C'est pourquoi l'approche de conception de Think et Kortex n'impose aucune machine abstraite : les ressources réifiées par des composants Think / Kortex sont minimales et spécifiques à chaque architecture matérielle. Ceci est une caractéristique originale de Think et Kortex. L'adaptation d'une application à un système Think (ou inversement) est réalisée par l'implantation et la composition de composants, sans modifier le code de composants existants.

9.2 Jivaro, système d'exploitation Java adaptable

Un projet interne de France Telecom a pour objectif de développer un *système d'exploitation Java* (*Java OS*), c'est-à-dire un système d'exploitation spécialisé dans l'exécution de code Java, au plus bas niveau d'abstraction possible. Ce système, appelé Jivaro, est décrit dans [TFL+04]. Les objectifs sont 1) de minimiser la consommation de ressources pour l'exécution d'applications Java dans un environnement restreint, tel qu'un environnement embarqué, et 2) d'offrir une grande adaptabilité dans le support d'exécution des applications. Jivaro est développé sur le système Think.

Dans Jivaro, d'un point de vue architectural, une application Java s'exécute dans un *domaine Java*. Un domaine Java est défini dans Jivaro comme un composant Fractal composite, qui agit comme un *domaine d'isolation* (dans le sens donné dans RM-ODP [ISO95b], cf. l'annexe C) pour les *composants implantés en Java* qu'il contient. Un domaine Java Jivaro est conçu pour être exécuté dans le système Think, en interagissant directement avec les composants du

système d'exploitation. La figure 9.1 illustre l'architecture d'un domaine Java qui contient un seul composant Java.

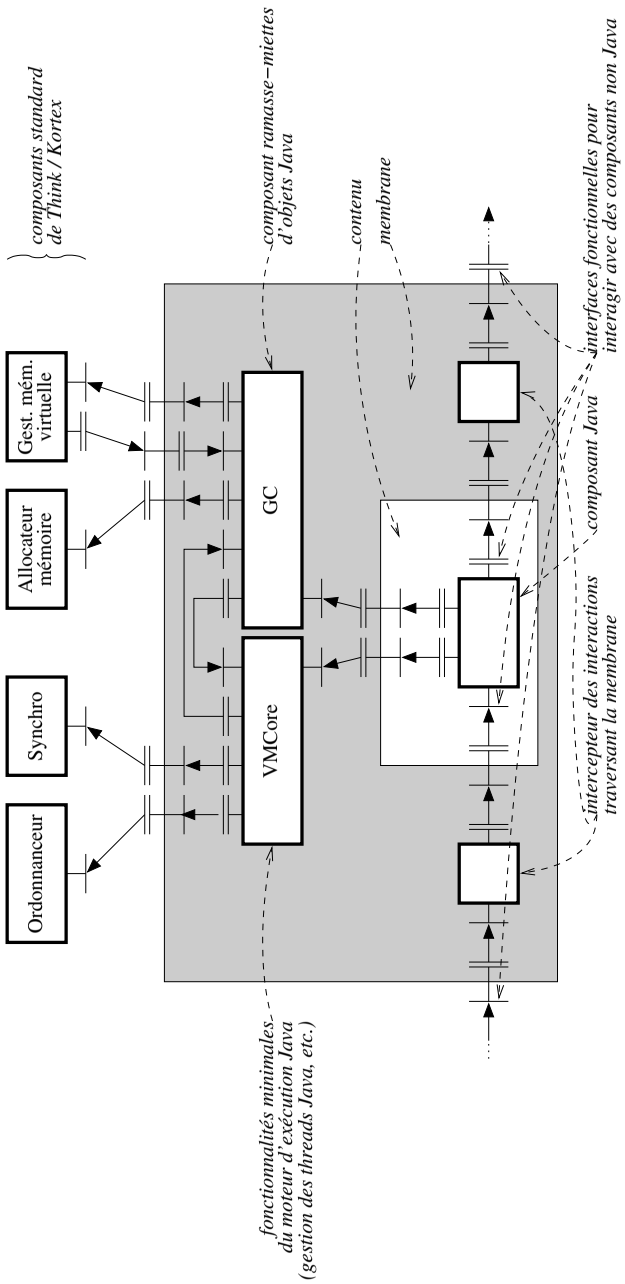


FIG. 9.1 : Architecture d'un domaine Java dans Jivaro

Ainsi chaque programme Java, lorsqu'il s'exécute dans un domaine, a l'illusion de disposer d'une JVM classique, alors même que plusieurs programmes Java s'exécutent simultanément dans des domaines différents. Chaque domaine Java gère donc 1) son propre espace mémoire isolé, 2) le « ramasse-miette » d'objets Java, et 3) son propre ensemble de valeurs d'attributs

statiques de classes Java. Le contrôle de l'exécution dans un domaine Java est réalisé par trois types de contrôleurs, placés dans la membrane du composant composite afférent au domaine considéré :

- un composant qui implante une politique de ramasse-miette (GC, *garbage collector*) pour les objets Java des composants Java contenus dans le composant composite ;
- un support d'exécution de base (VMCore), qui gère notamment la synchronisation des objets et la gestion des activités (*threads* Java) ;
- des intercepteurs des interactions traversant la membrane.

Un contrôleur GC s'appuie sur les mécanismes offerts par un contrôleur VMCore. Plusieurs contrôleurs GC peuvent être implantés. A tout instant, dans un domaine un seul contrôleur GC est actif, mais il peut être interchangé, sans modifier l'implantation de contrôleur VMCore fournie dans Jivaro. C'est pour faciliter cette adaptabilité que ces deux contrôleurs sont conçus séparément.

Un composant Java peut exporter / importer des interfaces de composants Fractal / Think, à travers la membrane de son domaine Java. Par exemple, dans l'architecture illustrée dans la figure 9.1, le composant Java importe une interface (client), et exporte une interface (serveur). Ainsi, il est possible de lier un composant Java avec tout composant hors du domaine, tels qu'un composant implanté en C, ou un autre composant Java dans un autre domaine. Il est également possible de lier deux composants Java dans le même domaine, sans nécessiter de « traverser » la membrane du domaine. Généralement, dans une JVM classique, la communication entre une application Java et l'environnement de la JVM doit être réalisée par des mécanismes basiques et complexes (JNI, *Java Native Interface*). Au contraire, dans Jivaro, la capacité d'établir des liaisons arbitraires entre des composants implantés indifféremment en Java ou C, est une fonctionnalité de base du système. Ainsi, les composants sont considérés de manière homogène dans le système, indépendamment de leur langage d'implantation (Java ou C). De plus, cette fonctionnalité est implantée de manière efficace. Notamment, pour effectuer une interaction entre deux domaines Java liés entre eux, un changement de contexte n'est pas nécessaire, comme par exemple entre deux JVM classiques animées chacune par un processus Unix.

Cependant, les seules interactions autorisées entre un composant Java et un composant dans l'environnement de son domaine, sont celles qui passent des valeurs par copie, ou des références de composant (Java ou autre). Il n'est pas autorisé de passer en paramètre une référence d'objet Java. En effet, si deux composants Java sont contenus dans des domaines Java différents, et sont liés entre leurs interfaces, la possibilité de passer une référence d'objet Java pourrait causer des défaillances dans la gestion de la mémoire (ramasse-miette) propre à chaque domaine. C'est grâce à cette propriété que nous qualifions les domaines Java de *domaines d'isolation*. L'interception des interactions avec les interfaces externes, dans la membrane, sert à gérer les références de composants qui sont transmises dans les interactions (création d'objets *stubs*, etc.).

Les contrôleurs dans la membrane (VMCore et GC) doivent être liés à d'autres composants du système d'exploitation (composants de la librairie Kortex) : ordonnanceur d'activités, synchronisation (sémaphores, etc.), allocateur de mémoire, et gestionnaire de mémoire virtuelle (MMU). Ainsi, plusieurs configurations peuvent être envisagées, telles qu'un ordonnancement préemptif des activités avec une gestion de mémoire paginée, ou un ordonnancement coopératif avec une gestion de mémoire plate, etc. Plusieurs implantations de ramasse-miettes d'objets

Java (GC) ont également été développées dans Jivaro : 1) par un algorithme incrémental, et 2) par un algorithme exact. D'autres systèmes d'exploitation Java similaires à Jivaro existent, tels que JX [GKB01]. Cependant, la plus grande limitation de ces systèmes est liée à la capacité de configuration du moteur d'exécution Java. Seul Jivaro offre une grande modularité et un contrôle complet de la configuration du support d'exécution Java. De plus, Jivaro permet de reconfigurer un domaine Java *dynamiquement*. Par exemple, il est possible de modifier la liaison avec le composant ordonnanceur, pour utiliser un autre algorithme d'ordonnancement.

L'originalité de Jivaro tient dans la démarche qui a été suivie. Cette démarche évite de concrétiser la machine abstraite Java pendant l'exécution du système, grâce à une compilation native des applications Java. Au contraire, les même concepts simples (les concepts du modèle de composant Fractal) sont utilisés pour concevoir un domaine Java Jivaro, et son environnement. Notamment, l'isolation d'un domaine Java est réalisée par la composition hiérarchique de composants Fractal, le contrôle est structuré grâce à la décomposition de tout composant en une membrane (qui contient les contrôleurs GC et VMCore) et un contenu (les objets Java), et la configuration et la reconfiguration dynamique sont réalisés à travers les contrôles offerts en standard par Fractal. Ainsi, Jivaro s'intègre parfaitement dans la démarche globale de notre laboratoire, décrite dans la section 2.3, en permettant de concevoir un système de manière homogène par des composants Fractal, à tous les niveaux d'abstraction : au niveau des applications Java, au niveau des supports d'exécution Java, et au niveau des systèmes d'exploitation. Dans ce contexte, on peut voir Jivaro comme un mécanisme qui permet de remplacer C par Java dans le développement de composants logiciels Fractal de bas niveau, ce qui donne accès à la programmation par objets, à un typage fort, à une gestion automatique du ramassage des objets Java, etc. Grâce à Jivaro, il est donc plus facile de concevoir des composants au niveau du système d'exploitation.

9.3 Compilateur modulaire basé sur Jabyce

L'implantation d'un domaine Jivaro ne repose pas sur l'interprétation du bytecode Java. En effet, dans un environnement construit sur des ressources restreintes, il est préférable de compiler toute l'application Java à exécuter sous forme de code binaire, avant l'exécution. L'intérêt de cette approche, appelée aussi *compilation en avance de phase* (*way-ahead-of-time compiling* [PTB+97]), est d'éliminer les temps de compilation du bytecode Java en code binaire pendant l'exécution (*just-in-time compiling*) ou d'interprétation du bytecode Java, tout en offrant des fonctionnalités similaires [VB04]. De plus, des optimisations plus « agressives » peuvent être effectuées lors de la compilation, optimisations qui sont particulièrement coûteuses en temps d'exécution. Dans Jivaro, le code compilé Java des composants Java est traduit vers l'implantation en C d'un composant Think qui offre des interfaces vers un contrôleur VMCore (pour la synchronisation des *threads*, etc.) et un contrôleur GC de ramasse-miettes. Ce code est ensuite compilé et assemblé comme pour n'importe quel composant Think, avant de pouvoir exécuter le système.

L'approche de la compilation en avance de phase de code compilé Java en code C est une approche classique. De nombreux moteurs d'exécution Java se basent sur la compilation de code compilé Java, vers du code C ou du code binaire [VB04]. Harissa [MS99] et Toba [PTB+97] sont les compilateurs Java vers C les plus représentatifs. Ces différentes approches diffèrent par le compromis qui est fait dans chaque approche entre 1) la complexité du compilateur

et du moteur d'exécution, 2) les performances des applications à l'exécution, et 3) la taille du code généré. La plupart des approches existantes, à l'exception notable de [VB04], ont pour objectif d'optimiser les temps d'exécution des applications, au détriment de la taille du code. Or, la taille du code peut être critique dans les environnements embarqués. C'est pourquoi la compilation en avance de phase est souvent décrite comme inadaptée aux environnements embarqués. Cependant, de même que dans l'approche suivie par [VB04], Jivaro est développé spécifiquement pour les environnements embarqués. C'est pourquoi, dans Jivaro, les contrôleurs de domaines Java sont minimaux et leur code est d'une taille restreinte. Surtout, le compilateur de Jivaro est conçu pour appliquer le maximum d'optimisations, avec pour objectifs d'optimiser les temps d'exécution du code compilé, mais aussi de réduire la taille du code généré.

La difficulté de la conception de Jivaro se reporte essentiellement sur le compilateur Java vers C. En effet, celui-ci doit effectuer le plus possible de transformations de programme « aggressives » pour optimiser le code généré en profitant des informations de haut niveau disponibles dans le bytecode Java, avant la traduction en code C. Cependant, les transformations de programme d'optimisation sont parmi les transformations les plus complexes à implanter. Pour maîtriser la configuration du compilateur, et favoriser la conception, la composition et la réutilisation des transformateurs d'optimisations complexes, le compilateur Jivaro est basé sur le système de transformation Jabyce que nous avons conçu. L'utilisation d'un système de transformation généraliste, tel que Jabyce, est une approche originale par rapport aux autres approches de compilation de code Java.

Le premier prototype de compilateur, baptisé Janook, a été développé dans le cadre de cette thèse. Ce travail est essentiellement une « preuve de concept », et n'a pas abouti à un compilateur complètement fonctionnel. L'architecture de ce prototype est illustrée dans la figure 9.2. L'implantation du *composant traducteur* offre toutes les interfaces serveur d'une opération Jabyce, et produit directement des fichiers source C sérialisés sur son interface client. Plusieurs opérations de transformation sont liées dans la chaîne, pour simplifier le format des classes qui sont traduites par le composant traducteur. Ainsi, le composant traducteur traite seulement un sous-ensemble de la spécification de la JVM, ce qui simplifie son implantation. Par exemple, les instructions jsr (saut à une sous-routine) et ret (retour d'une sous-routine) sont transformées en instruction goto, ce qui permet au traducteur de ne pas avoir à traduire ces instructions. L'originalité de notre traducteur est qu'il traduit les classes Java compilées en code C en un seul balayage de chaque classe. Le principal inconvénient est qu'aucune optimisation n'est appliquée sur le code généré.

Ce prototype a été réarchitecturé et réimplanté pour aboutir à une implantation complète. L'architecture de ce compilateur est similaire à celle de notre prototype initial, sous la forme d'une chaîne de composants Fractal. La limitation de notre premier prototype est la difficulté d'implanter des transformations complexes, telles que des optimisations, avec une représentation des classes Java compilées par des séquences d'interactions comme dans Jabyce. C'est pourquoi ce nouveau composant traducteur construit un graphe d'objets Java pour chaque classe Java représentée, de la même manière que dans les systèmes de transformation classique tels que BCEL (cf. le chapitre 5). Deux passes sont effectuées sur chaque graphe, pour générer le code C. Ce traducteur se base sur le fait que la plateforme Jivaro ne permet pas de charger dynamiquement du code Java compilé pendant l'exécution d'une application Java. Toutes les classes de l'application et les classes de Java utilisées (p.ex. les classes java.lang.String, java.util.List, etc.) sont compilées en une fois, pour produire du code C pour l'ensemble des

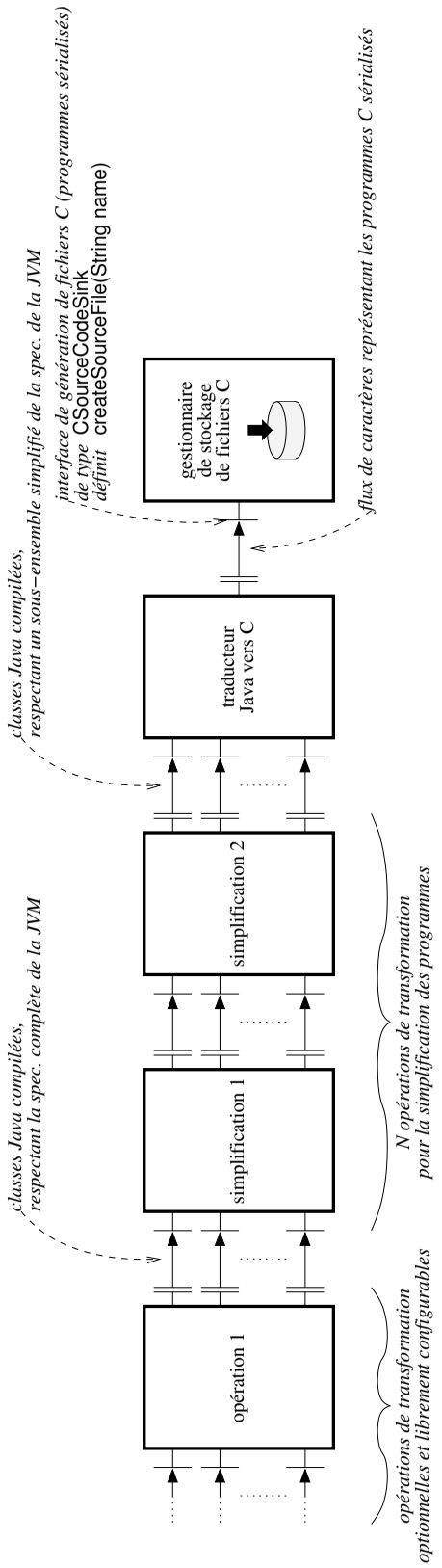


FIG. 9.2 : Architecture du prototype Janook de compilateur Java vers C

classes. Cette approche *« compile-the-world »* permet de ne pas respecter complètement les règles de compatibilité binaire des classes, et ainsi d'appliquer certaines optimisations « agressives » : la résolution statique des décalages d'addresses pour l'accès aux attributs, la dévirtualisation des appels de méthode non privées et non statiques, lorsque ceci est possible, etc.

9.4 Lien fonctionnel / technique à la compilation

Comme illustré dans la figure 9.2, il est possible de lier arbitrairement des opérations de transformation Jabyce dans la chaîne des opérations de transformation d'un compilateur Jivaro. Dans le contexte de notre problématique, qui est la construction efficace d'un lien entre architecture fonctionnelle et architecture technique, il est ainsi possible de lier des opérations de transformation qui construisent un lien entre les classes transformées puis compilées, et un ensemble de services techniques. Ainsi, par exemple, les opérations de transformation que nous avons développées pour le service de persistance Speedo peuvent être directement intégrées dans une telle chaîne d'opérations, pour être effectuées lors de la compilation des programmes Java. Ces opérations de transformation sont présentées dans le chapitre 7.

Ainsi, dans Jivaro, un domaine Java peut être considéré comme un moteur d'exécution mettant en œuvre un ensemble arbitraire de services techniques de manière transparente, en plus des services de bases offerts par toute JVM. Cette approche de la transformation pendant la compilation est le moyen le plus général et le plus efficace de composer de manière homogène et transparente un système d'exploitation et un ensemble de services techniques.

9.5 Vérification de la correction des programmes

La plateforme Jivaro n'implante pas complètement le processus de vérification des classes Java compilées, qui doit pourtant être implanté dans toute JVM. Nous proposons d'appliquer notre approche de vérification d'opérations de transformation Jabyce, détaillée dans le chapitre 4, basée sur l'observation des interactions qui représentent des éléments de programmes. Cette approche permettra d'implanter les passes 1 à 3 du processus de vérification des classes, pendant la compilation, pour détecter les erreurs de correction syntaxique et sémantique des classes Java compilées.

La passe 4 du processus de vérification est en revanche mise en œuvre dans Jivaro, par le moteur d'exécution, pour vérifier les erreurs à l'exécution. Cette « passe » consiste en la vérification de préconditions pour l'exécution d'instructions de bytecode Java. Par exemple, le support d'exécution doit vérifier que l'objet cible d'un appel de méthode (instruction invokevirtual) n'est pas null, sinon une exception NullPointerException doit être levée. Dans Jivaro, cette vérification est effectuée explicitement par du code de vérification qui est généré par le compilateur au moment de la génération du code C. Pour optimiser le code généré pour des applications « de confiance » (*trusted*), il est possible de désactiver la génération du code de vérification. Dans ce cas, les applications non fautives s'exécutent normallement avec de meilleures performances, mais les éventuelles applications fautives ont un comportement non déterminé si une erreur se produit.

Un problème avec l'approche de vérification dans Jivaro, est que les vérifications peuvent être inutilement redondantes. Dans certains cas, qui peuvent être déterminés à la compilation, le

code de vérification pourrait ne pas être généré pour certaines instructions. Notamment, dans l'implantation de méthode donnée dans le programme 9.1, la vérification de la cible d'appel n'a pas besoin d'être effectuée pour la deuxième instruction d'appel (invokevirtual), car les deux appels sont effectués avec le même objet cible, qui est vérifié être non null dès le premier appel.

```
public class UneAutreClasse
public methodeOptimisable();
  Code:
   0:   aload_0
   1:   dup
   2:   invokevirtual #2
   5:   invokevirtual #2
   8:   return
}
```

Prog. 9.1 : Code compilé où les vérifications sont optimisables

Une perspective pour l'évolution de Jivaro est d'introduire de nouveaux types d'éléments de *pseudo-instructions de vérification*, dans le modèle conceptuel. Par exemple, il est envisageable d'introduire le type CallNullTargetCheck, pour représenter des « pseudo-instructions de vérification de l'objet cible d'un appel de méthode ». Il est à noter qu'une telle pseudo-instruction n'est typiquement pas traduisible directement dans une seule instruction de bytecode, mais en une séquence d'instructions. Une opération de transformation peut être implantée facilement pour créer de tels éléments dans les séquences d'interactions qui représentent des classes Java compilées. Par exemple, une telle opération de transformation crée un élément CallNullTargetCheck avant chaque représentation d'instruction d'appel de méthode. Une opération de transformation peut ensuite optimiser le code, par une analyse de flot de données, pour supprimer les instructions de vérification pour lesquelles il peut être vérifié statiquement que les préconditions sont toujours vraies pour tous les chemins d'exécution vers cette instruction.

9.6 Perspectives pour l'architecture du compilateur

Le principal inconvénient du traducteur (le composant produisant du code C, dans la châine de composants formant le compilateur) actuellement implanté dans Jivaro, est qu'il est implanté sous la forme d'un composant Fractal primitif, ce qui ne permet pas de reconfigurer le compilateur pour effectuer des optimisations arbitraires lors de la compilation, telles que la duplication d'implantations de méthodes (*inlining*). L'une des perspectives de ces travaux est de réarchitecturer le traducteur de Jivaro, pour le décomposer et réintroduire la modularité de notre premier prototype Janook. L'architecture du compilateur que nous envisageons de développer est illustrée dans la figure 9.3.

Dans cette proposition d'architecture, la chaîne des composants formant un compilateur est constituée des parties suivantes :

- une chaîne d'opérations de transformation optionnelles et composées arbitrairement pour construire un lien entre les applications compilées et des services techniques, comme décrit dans la section 9.4 ;

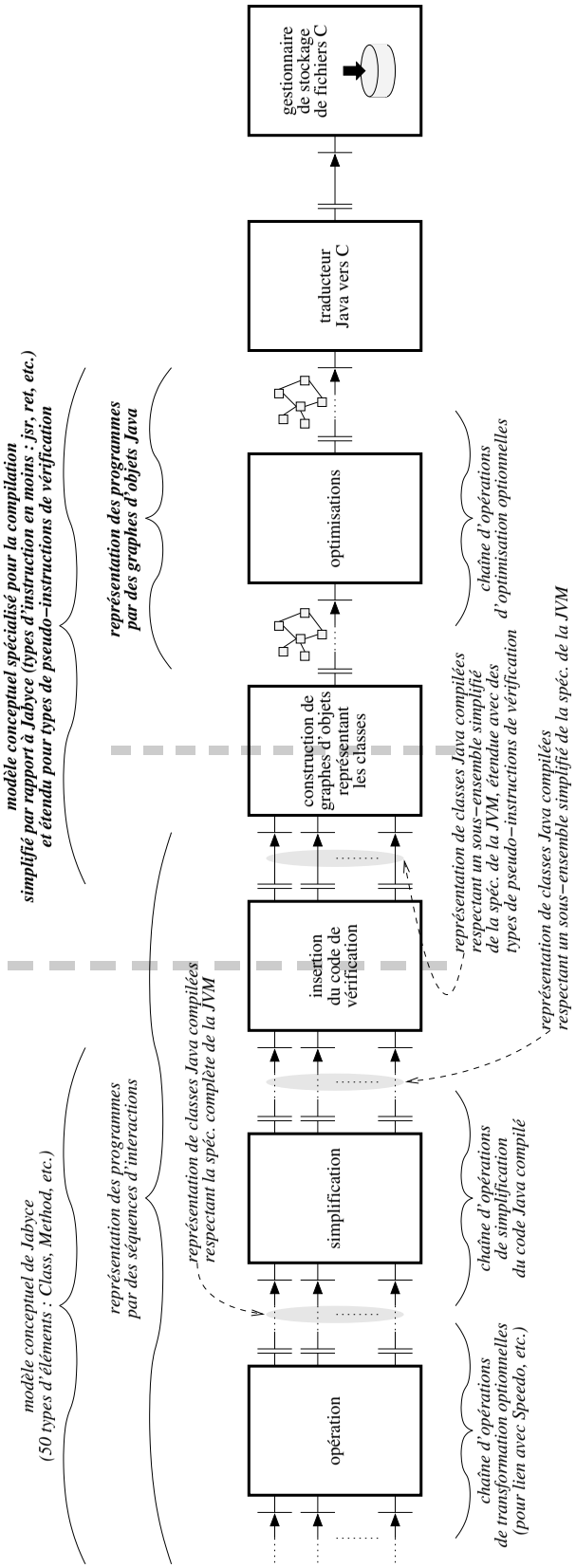


FIG. 9.3 : Proposition d'architecture pour le compilateur Java vers C de Jivaro

- une chaîne d'opérations de *normalisation*, c'est-à-dire de réduction des classes Java compilées à un sous-ensemble de la spécification de la JVM [LY99], par exemple en transformant les instructions jsr et ret en instructions goto, etc. (cf. la section 9.3) ;
- une opération de génération de code de vérification (passe 4 du vérificateur) sous la forme de *pseudo-instructions de vérification* ;
- un composant qui transforme les classes Java représentées sur ses interfaces serveur par des séquences d'interactions, en des *graphes d'objets Java équivalents* qui représentent les mêmes classes Java (représentation plus appropriée pour effectuer des optimisations) ;
- une chaîne de composants qui transforment des graphes d'objets Java représentant des classes Java pour les optimiser, par exemple en supprimant les pseudo-instructions de vérification redondantes (cf. la section 9.5) ;
- le composant traducteur, qui transforme les graphes d'objets représentant les classes Java, en fichiers C.

L'opération de génération de code de vérification est un *traducteur*, selon l'architecture définie dans la section 3.3.2. Cette opération offre toutes les interfaces serveur définies dans Jabyce pour la représentation d'éléments de classes Java, et en offre les interfaces client pour un autre système de transformation similaire (un autre modèle conceptuel), qui correspond à la spécification de la JVM simplifiée et étendue avec les types de pseudo-instruction de vérification (cf. la section 9.5).

La représentation des classes Java compilées par des graphes d'objets est plus appropriée que la représentation par des séquences d'interactions, pour supporter des opérations de transformation complexes et notamment des opérations d'optimisation. C'est pourquoi la chaîne comprend un composant qui construit un graphe d'objets représentant chaque classe Java compilée qui est représentée par des interactions sur ses interfaces serveur. Chaque graphe est ensuite transmis dans la chaîne des opérations d'optimisation, qui le transforment. Chaque graphe est ensuite balayé par le traducteur Java vers C, qui génère le code C correspondant. Dans le compilateur actuellement implanté dans Jivaro, les classes Java pour la construction de ces graphes ont été codées « à la main ». L'une des perspectives est d'étendre notre logiciel Jaidee, est utilisé pour générer l'essentiel du code d'un système de transformation tel que Jabyce (cf. la section 3.3), pour générer également automatiquement les classes des noeuds pour les graphes d'objets. Cette perspective est abordée plus en détails dans le chapitre 11.

Grâce à la décomposition du compilateur en une chaîne de composants, l'implantation du composant traducteur Java vers C est simplifiée. D'autres composants traducteur pourront facilement être développés, pour générer du code dans d'autres langages. Jivaro deviendra donc un *compilateur multi-cible*. Une implantation de composant traducteur pourrait par exemple se baser sur la librairie OpenJIT [OSM+00], pour générer directement du code binaire pour plusieurs architectures matérielles, à partir de code compilé Java.

D'un point de vue plus général, une autre perspective du développement de Jivaro est l'utilisation de nos propositions pour l'ordonnancement de tâches pour la configuration de tisseurs, détaillées dans le chapitre 8. En effet, la génération d'un système Jivaro comprend de nombreuses tâches : compilation de code Java vers C, compilation de code C, génération de code spécifique au système Think, etc. Ces tâches sont déjà décrites sous la forme d'un diagramme de flot de données, dans [TFL+04]. Ces travaux permettront de concevoir facilement un *tisseur Jivaro*, qui pourra être facilement composé avec d'autres tisseurs, tel que le tisseur de Speedo, comme décrit dans la section 8.6.

Quatrième partie

Conclusion et perspectives

Chapitre 10

Conclusion

Cette thèse aborde la problématique de la construction d'un lien entre les parties fonctionnelle et technique (non-fonctionnelle) d'un système. La technique qui a été choisie pour la construction d'un tel lien est la transformation de programmes compilés pour la machine virtuelle Java, car c'est la technique la plus générale et la plus performante parmi les techniques de lien existantes. La problématique qui est plus précisément abordée est celle de la conception d'un système de transformation, notamment d'un point de vue architectural, pour prendre en compte une généralisation de l'utilisation de la transformation de programme pour construire un tel lien fonctionnel / technique. Il devient en effet nécessaire de pouvoir composer efficacement des ensembles arbitraires de transformateurs de programme complexes. Les résultats de nos travaux comprennent à la fois des concepts et des réalisations pratiques, et concernent à la fois la conception d'un système de transformation de programme et celle d'outils concernant les problématiques associées, telles que la correction des programmes transformés ou le traitement des erreurs.

D'un point de vue théorique, l'une de nos principales contributions est l'introduction de la *représentation des programmes par des interactions* (« quelque chose qui arrive »), dans un système de transformation de programme. Ce mécanisme de représentation est opposé à la représentation des programmes par des graphes d'objets (« quelque chose qui est »), qui est utilisée dans tous les systèmes de transformation existants. L'introduction de ce nouveau mécanisme de représentation nous a conduit à étendre une catégorisation bien connue existante des systèmes de transformation, pour pouvoir comparer notre approche à celle des systèmes existants. D'un point de vue plus pratique, nous avons introduit l'utilisation d'un modèle de composants général (Fractal), afin de faciliter la conception de *composants transformateur*, et pour permettre de composer facilement des ensembles complexes de transformateurs en utilisant les mécanismes de liaisons du modèle de composant Fractal. Nous avons développé le système de transformation Jabyce, qui combine la représentation des programmes par interactions et l'utilisation du modèle de composants Fractal, pour la conception et la composition d'opérations de transformation (composants qui effectuent des transformations). Une comparaison avec les autres systèmes existants, d'un point de vue qualitatif et quantitatif (mesures de temps de transformation), a montré que ces deux choix de conception permettent de faciliter la réutilisation et la composition des transformateurs, et permettent d'obtenir de meilleures performances lorsque de nombreux transformateurs sont composés. Nos expérimentations concrètes ont montré l'avantage de l'utilisation de Fractal dans Jabyce, sur les autres systèmes,

pour la conception de transformateurs complexes. Notamment, il est possible de composer certaines opérations de transformation Jabyce récursivement, pour obtenir une *architecture fractale*. Cependant, d'un point de vue général, la comparaison de Jabyce avec les autres systèmes montre qu'il est plus difficile de développer certaines catégories de transformations avec Jabyce. La représentation des programmes par des séquences d'interactions rend notamment plus difficile l'implantation de transformations d'optimisation de programme. C'est pourquoi l'une de nos principales perspectives dans le développement de Jabyce, est d'introduire des mécanismes pour faciliter le développement, et notamment d'étendre Jabyce pour supporter également une représentation des programmes par des graphes d'objets. Ces perspectives sont détaillées dans le chapitre 11.

A l'intersection entre les domaines de l'architecture logicielle, la transformation de programme et la sûreté de fonctionnement, nous avons avons proposé une approche de validation des opérations de transformation Jabyce. Pour aborder cette nouvelle problématique, nous avons proposé une architecture de *domaine de traitement d'erreur*, pour vérifier pendant l'exécution de transformations, que les opérations de transformation produisent seulement des programmes syntaxiquement et sémantiquement corrects. Notre proposition s'appuie sur des travaux existants de gestion de contrats sur des composants Fractal, auxquels nous avons participé, et que nous avons étendus et adapté à la vérification de contrats de correction d'opérations de transformation Jabyce.

Un cas d'application concret de Jabyce, est son utilisation pour développer des opérations de transformation de programme complexes dans le contexte de Speedo, un service technique de persistance transparente d'objets Java. Ces opérations de transformation transforment les classes d'une application, pour lier cette application au service Speedo. La conception de ces opérations de transformation nous a permis de mettre en évidence les avantages de l'utilisation du modèle de composant Fractal pour la conception de transformations, en termes de modularité et de réutilisabilité. D'un point de vue plus général, le tisseur de Speedo, qui établit un lien entre les applications et le service Speedo, ne repose pas seulement sur la transformation de programme, mais également sur la génération de fichiers source, sur la compilation, etc. Pour permettre de combiner ces différentes techniques de manière adaptable, nous avons proposé une architecture pour l'ordonnancement de *tâches de tissage*, où les tâches sont conçues comme des composants Fractal, et sont ordonnancées automatiquement en fonction d'une spécification formelle des tâches et des ressources qu'elles consomment et produisent. Parmi les tâches du tisseur de Speedo, nous proposons que plusieurs d'entre elles soient des ensembles d'opérations de transformation de programme développées avec Jabyce.

Jabyce a également été utilisé pour développer le compilateur Java vers C d'un environnement d'exécution Java minimal, appelé Jivaro. Cet environnement d'exécution repose sur un système d'exploitation minimal, développé en utilisant Think, un canevas logiciel pour la construction de systèmes d'exploitation à l'aide de composants Fractal. Le compilateur de Jivaro est adaptable, et permet d'effectuer des transformation de tissage d'un lien fonctionnel / technique pendant la compilation, de manière très efficace et flexible. Les perspectives de ces travaux sont principalement le développement d'optimisations de transformation pour l'optimisation des programmes compilés en C.

D'autres perspectives d'utilisation de Jabyce sont envisageables, telles que la conception d'un tisseur d'aspects pour la programmation par aspects (AOP, *Aspect-Oriented Programming*), et le développement d'opérations de transformation pour le tissage avec d'autres services tech-

niques. La problématique de la programmation par aspects concerne plus la sémantique des langages d'aspects et du tissage des aspects que la transformation de programme qui est mise en œuvre pour le tissage. Les autres services techniques pour lesquels il est envisageable de développer des opérations de transformation de tissage, sont notamment la gestion de transactions [Pro03] et la sécurité (contrôle d'accès) [JFL04], pour lesquels des services techniques ont déjà été conçus sous la forme de configurations de composants Fractal. L'utilisation de la transformation de programme permettra de lier plus efficacement les applications avec ces services techniques.

Les deux chapitres suivants présentent plus particulièrement deux pistes de perspectives de travaux autour de Jabyce, en plus de celles qui sont présentées dans les différents chapitres de ce mémoire, et celles évoquées ci-dessus. Le chapitre 11 présente des perspectives d'évolution immédiate de Jabyce, pour faciliter la conception de transformateurs. Le chapitre 12 présente des perspectives à plus long terme, pour la constitution de librairies d'opérations de transformation Jabyce réutilisables, qui permettront de faciliter la composition de chaînes d'opérations de transformation.

Chapitre 11

Perspectives pour la simplification de Jabyce

Ce chapitre présente des perspectives d'extension de Jabyce pour simplifier le développement de transformateurs de programme. La section 11.1 présente notre proposition pour l'introduction d'une représentation des programmes par des graphes d'objets dans Jabyce, en complément de la représentation par des séquences d'interactions. La section 11.2 présente les perspectives en ce qui concerne la spécification de langages de haut niveau pour simplifier l'implantation de transformateurs.

Notre démarche se base sur une décomposition du système de transformation Jabyce en : 1) des mécanismes de bas niveau, correspondant au système Jabyce actuel, et 2) des abstractions de plus haut niveau pour simplifier l'implantation de transformateurs. Les abstractions de plus haut niveau peuvent être spécifiques à certains domaines. Cette décomposition est similaire à celle réalisée dans le cadre des langages spécifiques (DSL, cf. la section 2.2). L'enjeu de la conception des mécanismes est la généralité (le plus grand nombre de transformations doit pouvoir être implantable) et l'efficacité (la composition des transformateurs doit être efficace). Nous avons montré dans cette thèse que Jabyce remplit ces critères.

Cette démarche de séparation a récemment été proposée dans le cadre du système de transformation JOIE [CC04]. Ces travaux proposent différents outils basés sur JOIE, pour simplifier certaines catégories de transformations. Cependant, comme nous l'avons montré dans le chapitre 5, JOIE est moins général que Jabyce. Notamment, le modèle conceptuel de JOIE n'est pas homogène et les classes de JOIE, dont les objets représentent des éléments de programmes, sont peu modifiables, ce qui empêche d'implanter de nombreuses transformations telles que le renommage des méthodes. De plus, nous avons montré dans le chapitre 6 que la représentation des programmes par des graphes d'objets, qui est utilisée dans JOIE, offre de moins bonnes performances que la représentation par des séquences d'interactions, qui est utilisée dans Jabyce. De plus, dans de nombreux cas, la représentation par des séquences d'interactions est un mécanisme de représentation suffisant pour implanter de nombreuses transformations, dont la complexité ne nécessite pas d'utiliser une représentation par graphes d'objets. Jabyce est donc un meilleur système que JOIE pour jouer le rôle de mécanismes de bas niveau, dans une décomposition d'un système de transformation en deux niveaux.

11.1 Représentation par des graphes d'objets

Au niveau de la couche des « mécanismes de base » du système de transformation Jabyce, nous proposons d'introduire la représentation des programmes par des graphes d'objets, en complément de la représentation par des séquences d'interactions. La représentation par des graphes d'objets est plus appropriée pour implanter des transformations qui effectuent des analyses de flot de contrôle, par exemple. Ainsi, l'un ou l'autre mécanisme de représentation pourra être choisi pour développer chaque opération de transformation. Avec cette approche, il est possible de composer dans une même chaîne des opérations qui utilisent les deux mécanismes de représentation, avec le même modèle conceptuel. La conversion entre les deux mécanismes de représentation est réalisée par des composants de conversion interactions → graphes et graphes → interactions, comme nous le proposons dans la future architecture du compilateur Jivaro illustrée dans la figure 9.3.

Nous proposons d'intégrer les travaux du projet SEN1 (*Interactive Software Development and Renovation*) du CWI (Institut National de Recherche en Mathématiques et en Informatique des Pays-Bas), dans le domaine de la transformation de programme. Notamment, afin d'optimiser la consommation mémoire des graphes d'objets construits, nous proposons d'utiliser la librairie Java ATerm (*Annoted Term*) [vdBdJKO00]. La librairie ATerm permet de représenter efficacement des graphes, et notamment de factoriser automatiquement les sous-graphes pour économiser la mémoire. De nombreux outils ont été développés autour de ATerm. Notamment, ApiGen [dJO04] permet de générer automatiquement des classes Java de haut niveau pour les nœuds de graphes, qui masquent les détails des classes de base de la librairie ATerm. ApiGen génère le code source de ces classes à partir d'une spécification de grammaire dans le formalisme SDF (*Syntax Definition Formalism*). Nous envisageons d'étendre notre logiciel Jaidee, pour générer des spécifications SDF à partir de nos spécifications plus générales et plus simples. Basés sur les classes générées par ApiGen, les logiciels JJForester [KV01] et JJTraveler [DV04, vDV02] permettent respectivement de générer le code de classes de visiteurs de graphes, et de combiner arbitrairement des visiteurs. Ces classes générées de visiteurs permettront de simplifier significativement l'implantation de transformateurs qui manipulent des graphes. Basé également sur ApiGen, JTom [MRV03] est un compilateur d'expressions de patrons, pour la recherche de patrons (*pattern matching*) dans des graphes d'objets ApiGen. Des expressions JTom sont spécifiées directement dans le code source Java de transformateurs, dans une syntaxe spécifique, et transformées par JTom pour produire un code source Java transformé qui implante la recherche des patrons. L'utilisation de JTom sera un apport original de Jabyce par rapport aux autres systèmes de transformation de code compilé Java.

Un parallèle peut être fait entre ces outils, et les concepts et outils que nous avons utilisés ou proposés pour la conception de Jabyce. Les classes générées par ApiGen et ATerm permettent de représenter des éléments de programmes, de même que les interfaces de composant opération de transformation définies dans Jabyce et les liaisons entre les composants. ApiGen génère le code source des classes qui supportent la représentation des éléments de programmes, de manière similaire à Jaidee qui génère le code source des interfaces de composants qui supportent la représentation des éléments de programmes. JJForester et JJTraveler permettent de contrôler la stratégie de transformation de graphes d'objets, de la même manière que les liaisons entre composants Fractal et le langage ADL de Fractal permettent de contrôler la stratégie de transformation dans Jabyce. Ces deux ensembles d'outils sont donc au même niveau, et offrent des fonctionnalités similaires, tout en étant complémentaires car manipulant

des programmes représentés avec des mécanismes différents (graphes d'objets *vs.* séquences d'interactions).

Une limitation de la librairie ATerm est qu'elle permet de représenter seulement des graphes acycliques, par exemple des arbres, mais pas des graphes quelconques [vdBdJKO00]. Dans Jabyce, il sera donc possible de représenter, par des relations entre graphes ATerm, seulement les relations de composition entre éléments de programmes. Il ne sera pas possible de représenter de cette manière les relations de dépendance sémantique entre éléments, par exemple la relation entre une instruction d'accès à une variable locale et une déclaration de variable, qui conduirait à un cycle dans les relations entre termes. Ce problème de représentation de ces relations peut être résolu techniquement en les « encodants » par des relations entre les attributs des représentations d'éléments, de la même manière qu'elles sont encodées dans Jabyce et ASM. Comme décrit dans la section 5.3, ces relations sont encodées par l'égalité d'objets identifiants passés comme attributs des éléments représentés.

11.2 Langages de transformation

En ce qui concerne des langages de transformation de haut niveau, plusieurs formes de langages pourront être abordées : 1) les langages de spécification de fragments de programmes, 2) les langages de transformation généralistes, et 3) des langages de transformation spécialisés pour le paramétrage de certains transformateurs particuliers. Cette section aborde également des propriétés souhaitables des transformations de programme, notamment la réversibilité. Nous prévoyons que ces propriétés peuvent être atteintes systématiquement, en proposant des langages de transformation appropriés.

11.2.1 Fragments de programme

Il est envisageable de doter Jabyce d'un langage (*syntaxe concrète*) pour spécifier des fragments de programme. En effet, comme montré dans la section 5.8, un fragment de programme qui est créé par un transformateur Jabyce est actuellement spécifié dans l'implantation du transformateur explicitement sous la forme du code Java qui initie les appels qui représente les éléments du fragment. Pour faciliter l'implantation des transformateurs, il est envisageable de doter Jabyce d'un langage tel que celui du système Javassist, pour la spécification de fragments de programmes. Il sera également intéressant d'étendre l'utilisation d'un tel langage à la détection de patrons (*pattern matching*), et non plus seulement pour la génération d'éléments de programmes, comme dans Javassist. Un tel langage de spécification de fragments est offert dans le langage de transformation Stratego [Vis02], dont nous pourrons nous inspirer.

Le langage de spécification de fragment de Javassit est un sous-ensemble du langage source Java, qui n'est pas approprié pour la détection de patrons. En effet, si du code source Java peut facilement être compilé en code compilé Java, du code compilé ne peut pas toujours être décompilé en code source Java (cf. la section 7.3.3). Nous proposons donc de ne pas utiliser le langage source Java comme langage de spécification de fragments et de patrons, mais plutôt d'utiliser un langage « d'assemblage » propre à la spécification du code Java compilé, tel que le langage Jasmin [Mey96].

Du point de vue de l'implantation, Javassist compile les spécifications de fragments de programmes pendant l'exécution des transformations. Cette approche offre beaucoup de souplesse, mais consomme souvent inutilement du temps d'exécution. C'est pourquoi nous proposons plutôt une approche de compilation statique des fragments de programmes. Il est également envisageable de permettre la spécification de *fragments de programmes paramétrés*, dont certaines parties peuvent être spécifiées pendant l'exécution des transformations utilisant les fragments. Pour pouvoir utiliser de fragments de programmes comme des patrons de recherche, une approche simple est de traduire les spécifications de fragments en spécifications JTom (cf. la section précédente), qui sont ensuite compilées avec cet outil.

11.2.2 Langages généralistes

En complément de ces outils qui aident le développement de *règles procédurales*, c'est-à-dire de transformateurs à l'aide d'un langage impératif tel que Java, il est envisageable de proposer l'intégration d'un langage de transformation *déclaratif* dans Jabyce. Les langages de transformation déclaratifs les plus significatifs sont Stratego [Vis04] et ASF [vdBKV03]. Les spécifications de règles de réécriture dans ces langages sont compilées en C pour effectuer des transformations sur des graphes d'objets (graphes de « termes »). Les règles ASF sont compilées pour transformer des graphes d'objets représentés en utilisant la librairie ATerm. ASF est donc le langage de transformation déclaratif le plus cohérent avec nos propositions d'évolution de Jabyce, qui se basent sur l'utilisation de ATerm. Il n'existe pas de compilateur de règles ASF vers Java. Il n'est donc pas possible d'encapsuler directement des règles ASF compilées dans des opérations de transformation Jabyce implantées en Java. Cependant, la librairie ATerm offre des fonctions efficaces de sérialisation / désérialisation de graphes en flux d'octets. Il est donc possible d'interagir entre un processus exécutant des règles ASF compilées, et une chaîne d'opérations de transformation Jabyce. Les problèmes posés par cette intégration restent à étudier.

11.2.3 Langages spécialisés pour l'interception

L'une des catégories de transformations les plus couramment implantées est l'*interception* d'appels de méthodes Java. Notamment, l'interception est courramment utilisée pour implanter des conteneurs et la programmation par aspects (cf. la section 2.2). Pour faciliter l'implantation des transformations pour l'interception, certains travaux ont proposé des langages spécifiques pour spécifier le code qui doit être exécuté lors d'une interception, tels que Inject/J [GK03] pour la transformation d'interception dans du code source Java. Cependant, l'ensemble des transformations qui peuvent être spécifiées est généralement limité. Notamment, un seul mécanisme est généralement offert pour la mise en œuvre de l'interception : 1) avec un protocole à méta-objets, où le code d'interception est compilé dans le code d'un méta-objet, ou 2) l'intercepteur prend la forme d'un objet décorateur (cf. l'annexe B.2), ou encore 3) le code d'interception est inséré dans le code des méthodes dont les appels sont interceptés, ou autour des instructions d'appels dans les classes appelantes. Le choix d'un de ces mécanismes est un compromis entre l'adaptabilité à l'exécution pour configurer les interceptions à mettre en œuvre, le nombre d'objets (méta-objets ou décorateurs), et l'impact sur les temps d'exécution. Nous proposons de spécifier un langage de spécification de code d'interception, indépendant de

tout mécanisme de mise en œuvre, et d'implanter plusieurs implantation de transformateurs pour l'interception, qui supportent chacun la mise en œuvre d'un mécanisme.

11.2.4 Transformations réversibles

D'un point de vue plus théorique, une propriété intéressante d'une transformation de programme est la *réversibilité*, c'est-à-dire la capacité de retrouver un programme avant transformation, à partir d'un programme transformé. En effet, les versions les plus récentes de la spécification de la JVM (version 1.5) permettent de remplacer les classes Java implantant un système, pendant l'exécution du système. Il devient donc possible de transformer le code Java compilé pendant l'exécution. Dans le cadre du tissage d'un lien fonctionnel / technique entre une application et des services techniques, des transformations de tissage de lien réversibles permettraient de lier ou de délier facilement une application avec un service technique, pendant l'exécution. Le problème des transformations réversibles a surtout été étudié dans le domaine de la maintenance des vues dans le domaine des bases de données, et dans l'édition de documents semi-structurés (p.ex. XML). Dans ce domaine, les vues sont définies comme le résultat d'une transformation d'une base de données, ou d'un document semi-structuré. Ainsi, toute modification de la base de donnée (resp. du document) est répercutée vers la vue, à travers la transformation définie. Mais également, il est nécessaire de répercuter les modification apportées à la vue, vers la base ou le document, à travers la transformation inverse. Une solution apportée dans ce domaine est la spécification de langages de transformation qui ne permettent de spécifier que des transformations réversibles [MHT04]. Il sera intéressant d'étudier l'utilisation de langages de transformation de programme avec de telles propriétés.

11.2.5 Co-transformation des méta-classes Java

La JVM offre des capacités de réflexion aux applications Java (à la fois l'introspection et l'intercession), à travers des méta-objets dont les classes sont standard (java.lang.Class, java.lang.reflect...). Or, des transformations de programmes compilés pour la JVM peuvent se traduire par des modifications, des suppressions ou des ajouts d'éléments, qui sont reflétés pendant l'exécution des programmes transformés à travers les méta-objets. Cette modification de l'état et du comportement des méta-objets, due à la transformation des classes de base, peut par exemple provoquer des erreurs pendant l'exécution du système transformé, ou violer des propriétés de sécurité. Comme décrit dans le chapitre 5, JMangler est le seul système qui limite les transformations des éléments « externes » des classes Java compilées (relations d'héritage, attributs, signatures de méthodes, etc.), pour conserver la *compatibilité binaire* des classes. Cependant, ce système permet tout de même d'ajouter des éléments dans les classes, ce qui est reflété à travers les méta-objets. Ce problème dépasse donc celui de la compatibilité binaire. En poursuivant la démarche de JMangler, une solution serait de contraindre les transformations implantables, pour interdire toute modification des éléments qui sont réifiés par les méta-classes de la JVM. Cependant, cette approche serait trop restrictive pour aboutir à un système de transformation utilisable et utile.

Nous proposons donc au contraire d'autoriser toutes les transformations possibles, dans un soucis de généralité, et de masquer les transformations aux applications transformées qui s'exécutent, en offrant à un système des « faux » méta-objets qui offrent une vue du système

avant transformation. Cette solution est la plus transparente et la plus générale pour traiter ce problème. Cette approche nécessite de transformer les utilisations des méta-classes de la JVM, telles que les appels de méthodes réflexifs, en utilisation de méta-classes générées spécifiquement pour chaque classe transformée. Les méta-classes générées correspondent au système avant transformation. Par exemple, un accès réflexif à un attribut d'un objet est implanté dans une telle méta-classe comme un accès effectif à cet attribut dans l'objet réifié. Il est ensuite nécessaire de transformer (*co-transformer*) les méta-classes générées, simultanément avec les classes de l'application, pour masquer ces transformations aux applications réflexives. Par exemple, dans la transformation de réification de l'état des objets décrite dans la section 7.3, il est nécessaire de transformer les méta-classes pour transformer les accès aux attributs extraits, en des accès aux objets état. Ainsi, pendant l'exécution du système, tout accès réflexif à un attribut d'un objet d'une classe décomposée se traduit en un accès à l'attribut de l'objet état correspondant, de manière transparente pour le système.

Implanter des co-transformations de programme est a priori très difficile dans le cas général. Nous ne connaissons pas de travaux qui traitent ce problème. Les seuls travaux sur la co-transformation sont effectués dans le domaine des bases de données, pour la maintenance des vues lors de l'évolution de schémas de base de données. Cependant, ces travaux ne sont pas directement applicables dans notre contexte. Une piste de travail est la définition d'un langage de transformation de programme limité qui permette d'effectuer automatiquement des co-transformations des méta-classes en fonction des transformations effectuées sur les classes de base spécifiées dans ce langage. Dans un premier temps, un tel langage pourra être spécifié spécifiquement pour la transformation de code compilé pour la JVM. Ensuite, l'approche pourra être généralisée, pour être applicable à la co-transformation de programme dans tout langage réflexif.

Une perspective est également d'étudier la combinaison des deux propriétés de réversibilité (cf. la section précédente) et de co-transformation automatique des méta-classes, dans la conception d'un seul langage de transformation de programme.

Chapitre 12

Bibliothèques d'implantations de transformateurs

Ce chapitre présente notre proposition pour la constitution de bibliothèques d'implantations de composants opérations de transformation Jabyce. Pour chaque composant, nous proposons classiquement de conserver son implantation (classes Java compilées), l'information des fichiers ADL Fractal pour ce composant, et de la documentation en langue naturelle tirée de la Javadoc, etc. Le stockage d'une telle information, dans une base de données, n'est généralement pas un problème. En revanche, la difficulté est de rechercher des implantations de composants dans une telle base. La *recherche d'information* (*information retrieval*) est la science pour trouver des objets dans n'importe quel medium, correspondant à la demande d'un utilisateur [Mit03]. Le domaine de la *recherche de composants* (*component retrieval*) est une application des travaux en recherche d'information, où les objets recherchés sont des implantations de composants. [LdPdA04] offre une étude relativement complète des travaux dans ce domaine.

Ce chapitre introduit brièvement notre proposition d'un système général et adaptable de recherche de composants adapté au modèle Fractal, puis les aspects plus spécifiques de la recherche de composants de transformation de programme Jabyce, et les perspectives de recherche dans cette problématique.

12.1 Un système général de recherche de composants Fractal

La recherche de composants est utilisée typiquement dans le contexte de la conception de systèmes, pour rechercher des implantations de composants qui doivent être composés pour former un système complet. Cependant, nous considérons que la recherche de composants est utilisable dans un contexte plus large que la conception de systèmes. Cet élargissement du cadre de la recherche de composants est motivé par la spécification de modèles de composants tels que Fractal, qui offrent un contrôle réflexif d'un système pendant son exécution, notamment de la configuration dynamique du système. Nous proposons donc d'étendre la recherche de composants à l'identification de composants dans un système en cours d'exécution, en fonction de requêtes exprimées par un utilisateur. L'évaluation de telles requêtes repose par exemple sur les capacités d'introspection de Fractal. C'est pourquoi nous qualifions ce mécanisme d'*introspection associative*. L'objectif est de construire des *vues* sur les systèmes, de

la même manière que dans les bases de données. Ces vues sont construites en considérant métaphoriquement chaque composant Fractal comme une *source de méta-données* (sur les sous-composants, sur les liaisons, etc.), qui peuvent être interrogées à travers les interfaces d'introspection offertes par Fractal. Dans ce point de vue original, nous avons proposé dans [ALCL04] un système de recherche de composants Fractal, qui permet d'interroger à la fois des bases d'implantations de composants, et des configurations de systèmes en cours d'exécution. Ce système de recherche peut être utilisé à la fois pour la conception et pour l'administration de systèmes. Cette proposition est à l'intersection des domaines de la recherche d'information, de l'architecture logicielle et des bases de données réparties. Ce système est décrit plus en détails dans [ALCL04].

12.2 Recherche d'implantations de transformateurs Jabyce

En ce qui concerne plus précisément la recherche d'implantations d'opérations de transformation Jabyce, nous ne considérons pas les aspects dynamiques de la composition d'opérations de transformation. En effet, nous considérons qu'une chaîne d'opérations de transformation Jabyce ne peut pas être reconfigurée pendant l'exécution de transformations. En revanche, la recherche d'implantations d'opérations de transformation est utile dans ce contexte, pour faciliter la réutilisation et la composition d'opérations complexes. La suite de ce chapitre aborde les caractéristiques spécifiques de la recherche d'implantations d'opérations Jabyce.

Parmi les systèmes de recherche de composant, les systèmes qui se basent sur le traitement automatique des langues offrent les résultats les plus pertinents et les plus riches. Par exemple, dans le contexte de la conception de composants Fractal, une grande quantité d'information est disponible en pratique pour les composants sous la forme de texte en langue naturelle, notamment sous la forme de Javadoc. La prise en compte de cette information permet d'améliorer significativement la qualité des résultats des requêtes. Le système ROSA est l'un des seuls systèmes complets de recherche de composants qui repose sur le traitement des langues naturelles [GI93]. ROSA offre la possibilité de formuler des requêtes en anglais, et de rechercher des composants en fonction de leur documentation en anglais. Cependant, dans ce système, seule l'information en langue naturelle est prise en compte dans les requêtes, et les informations de configuration et de type des composants ne sont pas prises en compte.

Dans le cas des implantations d'opérations de transformation Jabyce, nous considérons que l'information disponible est constituée 1) d'information en langue naturelle (par exemple des fichiers Javadoc), et 2) des informations de typage et de configuration tirées de fichiers ADL (*Architecture Description Language*) Fractal qui définissent les composants. Les informations de typage peuvent notamment prendre en compte les signatures d'interfaces (interfaces Java) et de méthodes. Afin d'offrir un système adaptable à de nombreux cas d'utilisation, le système de requête que nous proposons est extensible en ce qui concerne l'expression des requêtes [ALCL04], ce qui permet de combiner des opérateurs concernant la configuration des composants, et des opérateurs sur l'information en langue naturelle. Ce système de requête permet de définir de nouveaux opérateurs de requête. Dans notre contexte, nous nous intéressons principalement à la définition d'opérateurs liés à la recherche d'information en langue naturelle.

Dans le système ROSA, les composants considérés sont des *composants métier*. Ces composants sont donc décrits en utilisant une terminologie spécifique à un domaine, par exemple

avec les termes de « facture », de « compte », etc. dans le domaine de la facturation. L'avantage de supposer l'utilisation de terminologies spécifiques, comme dans ROSA, est qu'il est nécessaire de manipuler seulement des mots, pas les sens des mots. En effet, une terminologie étant idéalement non ambigüe, les mots (termes) sont monosémiques (ils n'ont qu'une seule acception). Cependant, dans le cadre de Jabyce, les opérations de transformation sont des *composants techniques*, et plus particulièrement spécifiques à un domaine de l'informatique. Or, comme évoqué dans la section 2.4, il n'y a pas de terminologie informatique à proprement parler : les termes utilisés en informatique sont généralement des utilisations métaphoriques de termes d'autres domaines. Par exemple, dans le cas des programmes Java compilés, le mot « instruction » est utilisé comme une métaphore de ce terme militaire. Nous considérons que les terminologies techniques introduisent essentiellement de nouvelles acceptions métaphoriques de mots existants, qui rendent ces mots polysémiques. Pour éviter l'ambigüité dans l'utilisation des termes, nous proposons de *raisonner sur les acceptions*, c'est-à-dire sur le sens des mots, et non pas sur les mots eux-mêmes comme dans les systèmes existants.

Le problème qui se pose généralement en traitement automatique des langues est la disponibilité des ressources lexicales (dictionnaires, etc.). Dans les systèmes de recherche existants tels que ROSA, la démarche proposée est de fournir à l'utilisateur le moyen de définir ses propres ressources qui seront utilisées pour les recherches. Cependant, la production d'une ressource lexicale (dictionnaire...) est un travail très difficile, et la qualité des résultats de recherche dépendent de la qualité et de la quantité d'information disponible dans les ressources lexicales utilisées. Nous proposons donc de permettre de réutiliser des ressources lexicales existantes, et d'offrir à l'utilisateur le moyen de personnaliser facilement ces ressources dans son contexte d'utilisation. Par exemple, dans le contexte de Jabyce, un cas d'utilisation est 1) la récupération d'un dictionnaire monolingue à usage général (par exemple, une version électronique du Petit Robert), et sa normalisation pour l'utilisation dans le système de recherche, puis 2) la personnalisation du dictionnaire dans le domaine spécifique considéré, en définissant de nouveaux mots ou de nouvelles acceptions, identifiées comme un *dialecte* de la langue du dictionnaire général. Une caractéristique originale de notre approche est de considérer la possibilité de définir plusieurs « dialectes techniques » en se basant sur les mêmes ressources lexicales générales. A notre connaissance, aucune approche de traitement des langues, et notamment de construction de ressources lexicales, ne prend en compte la capacité de définir des dialectes de langues générales de manière efficace, c'est-à-dire sans dupliquer complètement les ressources [Mit03]. Cette problématique nous apparaît pourtant essentielle dans le contexte du traitement de textes spécifiques au domaine de l'informatique.

La recherche d'information multilingue est également une fonctionnalité qui nous paraît essentielle, et qui n'est que peu traitée par les systèmes de recherche d'information existants [Mit03], et pas traitée par les systèmes de recherche de composants. Par exemple, ROSA supporte seulement la langue anglaise. Dans un environnement de développement fortement réparti, à un niveau international, la prise en compte de plusieurs langues permet à chaque développeur de documenter une implantation de composant dans sa langue maternelle, ce qui facilite la documentation et par conséquent augmente la documentation produite pour chaque implantation de composant. Le problème essentiel de la recherche d'information multilingue est la disponibilité de ressources lexicales multilingues, et leur qualité [Tee04a]. Parmi les projets de bases lexicales multilingues, l'un des plus significatif est le projet collaboratif Papillon [MLSL03], qui vise à produire une base lexicale qui comprend 9 langues, dont le français, l'anglais, et plusieurs langues asiatiques dont le japonais. La base Papillon est base d'accep-

tions, c'est-à-dire qu'elle relie des acceptions (sens de mots) de plusieurs langues, et non pas les mots entre eux. Ce type de base lexicale est donc bien adapté à notre contexte d'utilisation.

La construction d'une base lexicale multilingue est un problème difficile [Tee04b]. Cependant, le projet Papillon va fournir une base lexicale complète à usage général, et il sera possible de l'adapter à des besoins précis. Notamment, dans notre contexte, l'adaptation au « dialecte technique » de la transformation de code compilé se fera en définissant de nouvelles acceptions dans chaque langue, et les liens interlingues entre ces acceptions. Pour les nouveaux mots introduits dans le dialecte, cette activité est facile, car ces mots sont non polysémiques, ce qui rend la traduction directe entre les différentes langues. En ce qui concerne plus particulièrement les mnémoniques d'instructions (« ireturn », « aload », etc.), la construction des liens interlingues est triviale, parce que les mêmes mots sont utilisés dans les dialectes des différentes langues. Par exemple, le mot « ireturn » est utilisé aussi bien dans le dialecte « anglais pour le bytecode Java » que dans le dialecte « français pour le bytecode Java ». Il suffit donc de créer des liens interlingues entre les acceptions des mots qui représentent le même mnémonique d'instruction de code compilé Java.

Dans Papillon, les liens interlingues servent à représenter des relations de traduction entre sens de mots, mais des liens sémantiques d'autres types peuvent également être construits. Notamment, il est envisageable d'utiliser des ressources lexicales d'autres natures, telles que des thésaurii ou des dictionnaires de synonymes, pour établir des liens sémantiques entre mots d'une même langue. En ce qui concerne l'expression des requêtes, des opérateurs peuvent être définis pour exploiter les informations spécifiques à chaque ressource lexicale, par exemple un opérateur synonymeDe(...) peut être implanté pour explorer des liens construits à partir d'un dictionnaire de synonyme. Dans le contexte de Jabyce, l'information de synonymie peut être utilisée par exemple pour établir des liens de synonymie entre les types d'éléments du modèle conceptuel de Jabyce tels que ReturnInstruction, et les mnémoniques des instructions tels que ireturn, areturn, lreturn, etc.

12.3 Conclusion

Ce chapitre présente deux propositions pour élargir les cas d'utilisation de la recherche de composants, à l'administration de systèmes fortement répartis et dynamiques, et à la conception de tels systèmes à une échelle internationale. Ces propositions sont 1) l'*introspection associative*, c'est-à-dire la recherche de composants dans des systèmes en cours d'exécution, et 2) l'utilisation de ressources multilingues. Nous proposons d'introduire l'introspection associative comme une fonctionnalité générale utile pour tout système conçu avec le modèle de composants Fractal. L'utilisation de ressources multilingues est en revanche motivée par les propriétés spécifiques du langage naturel utilisé dans le domaine de la transformation de programmes Java compilés. Nous proposons de tirer parti de ces propriétés pour optimiser la qualité des résultats de recherche de composants de transformation Jabyce dans un contexte multilingue.

Parmi les systèmes de recherche de composants utilisés en pratique, à notre connaissance aucun n'est basé sur des techniques de traitement des langues avancés. De plus, à notre connaissance, aucun système de recherche de composants existant ne permet de prendre en compte des requêtes multilingues ou des informations multingues sur les composants. Or, ces informations

permettent d'améliorer significativement la qualité des résultats des recherches de composants. Cependant, les perspectives de ces travaux sont à très long terme. En effet, les avancées dans la conception de systèmes de recherche de composants en langue naturelle multilingues dérivent des avancées dans le domaine de la recherche d'information multilingue, qui est un domaine de recherche récent [Mit03]. A leur tour, les avancées dans la recherche d'information multilingue dérivent des avancées dans la construction de bases lexicales multilingues, qui est un domaine de recherche également récent et actif. Les « temps de propagation » des résultats de recherche entre ces différents domaines impliquent que des systèmes de recherche de composants en langue naturelle multilingues ne seront opérationnels et utilisés qu'à très long terme.

Bibliographie

[ABG+01] Anders Andersen, Gordon S. Blair, Vera Goebel, Randi Karlsen, Tage Stabell-Kulø, et Weihai Yu. Arctic Beans : Flexible and Open Enterprise Component Architectures. Dans *Proceedings of the Norsk Informatikkonferanse Conference (NIK 2001)*, Tromsø, Norway, novembre 2001.

[ABV92] Mehmet Akşit, Lodewijk M.J. Bergmans, et Sinan Vural. An Object-Oriented Language-Database Integration Model : The Composition-Filters Approach. Dans *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615, pages 372–395. Springer–Verlag, juin 1992.

[ACBD+04] Mourad Alia, Sébastien Chassande-Barrioz, Pascal Déchamboux, Catherine Hamon, et Alexandre Lefebvre. A Middleware Framework for the Persistence and Querying of Java Objects. Dans Martin Odersky, éditeur, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, volume 3086 de *Lecture Notes in Computer Science*, pages 291–315, Oslo, Norway, juin 2004. Springer-Verlag.

[AK02] Parker Abercrombie et Murat Karaorman. jContractor : Bytecode Instrumentation Techniques for Implementing Design by Contract in Java. Dans *Proceedings of the 2nd Workshop on Runtime Verification (RV 02)*, Copenhagen, Denmark, juillet 2002.

[AKWN] Parker Abercrombie, Murat Karaorman, David P. White, et Christopher Niederauer. jContractor – `http://jcontractor.sf.net/`.

[ALCL04] Mourad Alia, Romain Lenglet, Thierry Coupaye, et Alexandre Lefebvre. Querying reflexive component-based architectures. Dans *Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), track on Component-Based Software Engineering*, pages 127–134, Rennes, France, septembre 2004. IEEE Computer Society Press.

[Ale79] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* The Royal Institute of Technology, Stockholm, Sweden, décembre 2003. Doctor of Technology Dissertation.

[BBMW02] Jean-Louis Bénard, Laurent Bossavit, Régis Médina, et Dominic Williams. *L'eXtreme Programming.* Référence - Technologies objet. Eyrolles, Paris, France, mai 2002.

[BC02] Mikaël Beauvois et Thierry Coupaye. Composition comportementale d'aspects techniques. Dans *Proceedings of the ASF (ACM SIGOPS France) Journées*

Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems), Grenoble, France, novembre 2002.

[BCL+04] Eric Bruneton, Thierry Coupaye, Mathieu Leclercq, Vivien Quéma, et Jean-Bernard Stefani. An Open Component Model and Its Support in Java. Dans *Proceedings of the 7th Component-Based Software Engineering International Symposium (ICSE2004-CBSE7)*, volume 3054 de *Lecture Notes in Computer Science*, pages 7–22, Edinburgh, Scotland, UK, mai 2004. Springer-Verlag.

[BCM+01] Pierre Bieber, Jacques Cazin, Abdellah El Marouani, Pierre Girard, Jean Louis Lanet, Virginie Wiels, et Guy Zanon. The PACAP Prototype : A Tool for Detecting Java Card Illegal Flow. Dans Isabelle Attali et Thomas Jensen, éditeurs, *Proceedings of the 1st International Workshop on Java on Smart Cards : Programming and Security*, volume 2041 de *Lecture Notes in Computer Science*, pages 25–37, Cannes, France, 2001. Springer-Verlag.

[BCS02] Eric Bruneton, Thierry Coupaye, et Jean-Bernard Stefani. Recursive and Dynamic Software Composition with Sharing. Dans *Proceedings of the 7th International Workshop on Component-Oriented Programming (WCOP'02, ECOOP 2002)*, Malaga, Spain, juin 2002.

[BCS04] Eric Bruneton, Thierry Coupaye, et Jean-Bernard Stefani. The Fractal Component Model Specification 2.0, `http://fractal.objectweb.org/specification/index.html`, février 2004.

[Bea03] Mikaël Beauvois. Brenda : Towards a Composition Framework for Non-orthogonal Non-functional Properties. Dans Jean-Bernard Stefani, Isabelle Demeure, et Daniel Hagimont, éditeurs, *Proceedings of the 4th IFIP WG6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS'03)*, Lecture Notes in Computer Science, pages 29–40, Paris, France, novembre 2003. Springer-Verlag.

[BFB+] Bill Burke, Marc Fleury, Adrian Brock, Claude Hussenet, Kabir Khan, Andrew Oswald, Marshall Culpepper, et Honain Khan. JBoss Aspect Oriented Programming – `http://www.jboss.org/products/aop`.

[BFG03] David Basin, Stefan Friedrich, et Marek Gawkowski. Bytecode Verification by Model Checking. *Journal of Automated Reasoning*, 30(3–4) :399–444, 2003.

[BHvdMW] Jethro Bakker, Wilke Havinga, Peter van der Meer, et Johan Wiskerke. JMunger – `http://wwwhome.cs.utwente.nl/~bakkerj/jmunger/`.

[BJNR98] Kathy Bohrer, Verlyn Johnson, Anders Nilsson, et Bradley Rubin. Business process components for distributed object applications. *Communications of the ACM*, 41(6) :43–48, juin 1998.

[BLC02] Eric Bruneton, Romain Lenglet, et Thierry Coupaye. ASM : a code manipulation tool to implement adaptable systems. Dans *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Grenoble, France, novembre 2002.

[BNRB] Juozas Baliuka, Chris Nokleberg, Matas Ramoska, et Wes Biggs. Code Generation Library – `http://cglib.sf.net/`.

[BP03] Cees-Bart Breunesse et Erik Poll. Verifying JML specifications with model fields. Dans *Proceedings of the 5th ECOOP'2003 Workshop on Formal Techniques for Java-like Programs (FTfJP'2003)*, pages 51–60, Darmstadt, Germany, juillet 2003. Technical Report 408, ETH Zurich.

[BS98] Boris Bokowski et André Spiegel. Barat – A Front-End for Java. Rapport Technique B-98-09, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, décembre 1998.

[BSD] Boris Bokowski, André Spiegel, et Markus Dahm. Barat – `http://barat.sf.net/`.

[BSL01] Noury M. N. Bouraqadi-Saâdani et Thomas Ledoux. Le point sur la programmation par aspects. *Techniques et sciences informatiques*, 20(4) :505–528, 2001.

[CC04] Geoff A. Cohen et Jeff S. Chase. An Architecture for Safe Bytecode Insertion. *Software : Practice and Experience*, 2004. To appear.

[CCK98] Geoff A. Cohen, Jeff S. Chase, et David L. Kaminsky. Automatic Program Transformation with JOIE. Dans *Proceedings of the 1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[CES86] Edmund M. Clarke, E. Allen Emerson, et A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications : a practical approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[Cha99] Jean-Marie Chauvet. *Composants et transactions : Corba/OTS, EJB/JTS, COM/MTS*. Eyrolles, Paris, France, 1999.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. Dans *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 285–299, Austin, Texas, USA, octobre 1995.

[Chi00] Shigeru Chiba. Load-Time Structural Reflection in Java. Dans *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 de *Lecture Notes in Computer Science*, pages 313–336, Sophia Antipolis and Cannes, France, juin 2000. Springer–Verlag.

[CHL+98] Charles Consel, Luke Hornof, Julia Lawall, Renaud Marlet, Gilles Muller, Jacques Noyé, Scott Thibault, et Eugen-Nicolae Volanschi. Tempo : Specializing Systems Applications and Beyond. *ACM Computing Surveys*, 30(3es), septembre 1998. Article 19.

[CLB+01] Thierry Coupaye, Romain Lenglet, Mikael Beauvois, Eric Bruneton, et Pascal Déchamboux. Composants et composition dans l'architecture des systèmes répartis. Dans *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2001 : flexibilité du système au langage (flexibility from systems to languages)*, Besançon, France, octobre 2001.

[CLSE03] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, et Stephen Edwards. Model Variables : Cleanly Supporting Abstraction in Design By Contract. Rapport Technique 03-10, Department of Computer Science, Iowa State University, Ames, Iowa, USA, avril 2003.

[CO03] Philippe Collet et Alain Ozanne. Spécification d'intégration du modèle de contrats dans la plateforme Fractal. Rapport Technique F-3.1, version 1.0,

Collaboration Contract France Telecom – CNRS – I3S, number 422721832I3S, Sophia Antipolis, France, septembre 2003.

[Coh97] Richard M. Cohen. The Defensive Java Virtual Machine Specification. Rapport technique, Computational Logic Inc., mai 1997. Draft version Alpha 1.

[CR03] Philippe Collet et Roger Rousseau. Modèle de contrats. Rapport Technique F-2.2, version 2.0, Collaboration Contract France Telecom – CNRS – I3S, number 422721832I3S, Sophia Antipolis, France, juin 2003.

[Cyta] Ron K. Cytron. Clazzer – `http://www.cs.wustl.edu/~doc/RandD/PCES/PCESjava/Clazzer.html`.

[Cytb] Ron K. Cytron. PCESjava – `http://www.cs.wustl.edu/~doc/RandD/PCES/PCESjava/`.

[Deu89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Systems. Dans T. J. Biggerstaff et A. J. Perlis, éditeurs, *Software Reusability*, volume II, pages 57–71. ACM Press, New York, 1989.

[Dij68] Edsger W. Dijkstra. The structure of ¨THE¨ multiprogramming system. *Communications of the ACM*, 11(5) :341–346, mai 1968.

[Dij83] Edsger W. Dijkstra. The structure of ¨THE¨ multiprogramming system. *Communications of the ACM*, 26(1) :49–52, janvier 1983.

[DJC94] Michel Diaz, Guy Juanole, et Jean-Pierre Courtiat. Observer – A Concept for Formal On-Line Validation of Distributed Systems. *IEEE Transactions on Software Engineering*, 20(12) :900–913, décembre 1994.

[dJO04] Hayco A. de Jong et Pieter A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59 :35–61, 2004.

[dJVV01] Merijn de Jonge, Eelco Visser, et Joost Visser. XT : a bundle of program transformation tools. *Electronic Notes in Theoretical Computer Science*, 44, 2001.

[dlLF] Office Québécois de la Langue Française. Le grand dictionnaire terminologique – `http://www.granddictionnaire.com/`.

[DMM+] Markus Dahm, Conor MacNeill, Costin Manolache, Jason van Zyl, David Dixon-Peugh, et Enver Haase. Byte Code Engineering Library (BCEL) – `http://jakarta.apache.org/bcel/`.

[DR96] Laura K. Dillon et Y. Srinivas Ramakrishna. Generating oracles from your favorite temporal logic specifications. Dans *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 106–117, San Francisco, California, USA, octobre 1996. ACM Press.

[DV04] Arie Van Deursen et Joost Visser. Source Model Analysis Using the JJTraveler Visitor Combinator Framework. *Software : Practice and Experience*, 34(14) :1345–1379, novembre 2004.

[DWE98] Sophia Drossopoulou, David Wragg, et Susan Eisenbach. What is Java binary compatibility ? Dans *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 341–361, Vancouver, British Columbia, Canada, octobre 1998. ACM Press.

[DÜYK01] Linda G. DeMichiel, L. Ümit Yalçinalp, et Sanjeev Krishnan. Enterprise Java Beans Specification Version 2.0 Proposed Final Draft 2. Sun Microsystems Inc., avril 2001.

[ea99] BEA Systems et al.. CORBA Component Model Joint Revised Submission. Object Management Group, OMG Document orbos/99-07-01 ed., juillet 1999.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, et James O'Toole Jr.. Exokernel : An Operating System Architecture for Application-Level Resource Management. Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 251–266, Copper Mountain, Colorado, USA, décembre 1995. ACM Press.

[FBH+] Bernard Farrell, Tim Buchalka, Jochen Hoenicke, Jonathan Nash, et Tobias Rademacher. JODE, Java Optimize and Decompile Environment – `http://jode.sf.net/`.

[FM98] Stephen N. Freund et John C. Mitchell. A type system for object initialization in the Java bytecode language. Dans *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 310–327, Vancouver, British Columbia, Canada, octobre 1998. ACM Press.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern – `http://www.martinfowler.com/articles/injection.html`, janvier 2004.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, et Gilles Muller. Think : A software framework for component-based operating system kernels. Dans *Proceedings of the Usenix Annual Technical Conference 2002*, Monterey, California, USA, juin 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object–Oriented Software*. Addison Wesley, 1994.

[GI93] M. Rosario Girardi et Bertrand Ibrahim. A Software Reuse System Based on Natural Language Specifications. Dans *Proceedings of the 5th International Conference on Computing and Information (ICCI'93)*, pages 507–511, Sudbury, Ontario, Canada, mai 1993. IEEE Computer Society Press.

[GK03] Thomas Genssler et Volker Kuttruff. Source-to-Source Transformation In The Large. Dans *Proceedings of the Joint Modular Languages Conference (JMLC 2003)*, volume 2789 de *Lecture Notes in Computer Science*, pages 254–265, Klagenfurt, Austria, août 2003. Springer-Verlag.

[GKB01] Michael Golm, Jürgen Kleinöder, et Frank Bellosa. Beyond Address Spaces – Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. Dans *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 3–8, Elmau, Germany, mai 2001. IEEE Computer Society Press.

[Gro] Christian Grothoff. JAMIT, the Java Access Modifier Inference Tool – `http://www.ovmj.org/jamit/`.

[Gro03] Java Data Objects Expert Group. Java Data Objects Specification (JDO), Java Specification Request 12, Version 1.0.1, mai 2003.

[HB99] Galen Hunt et Doug Brubacher. Detours : Binary Interception of Win32 Functions. Dans *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, Washington, USA, juillet 1999.

[Hei] Thorsten Heit. JavaGuard – `http://sf.net/projects/javaguard/`.

[HHT] Paul Hammant, Aslak Hellesoy, et Jon Tirsen. PicoContainer – `http://www.picocontainer.org/`.

[HHW+00] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, et Steve Byrne. Document Object Model (DOM) Level 2 Core Specification – `http://www.w3.org/TR/DOM-Level-2-Core/`, novembre 2000.

[HL95] Walter L. Hürsch et Cristina Videira Lopes. Separation of Concerns. Rapport Technique NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, USA, février 1995.

[Hof82] Douglas Hofstadter. *Ma thémagie*, Chapitre Des variations sur un thème considérées comme l'essence de la créativité, pages 220–248. InterEditions, octobre 1982.

[Hou] Arnd Houben. JoGa – `http://www.nq4.de/webSite/JoGa/joga.html`.

[ISO95a] ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 2 : Foundations, International Standard ISO/IEC 10746–2, ITU–T Recommendation X.902, 1995, 1995.

[ISO95b] ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 3 : Architecture, International Standard ISO/IEC 10746–3, ITU–T Recommendation X.903, 1995, 1995.

[Izw03] Sattar Izwaini. A corpus-based study of metaphor in information technology. Dans John Barnden, Sheila Glasbey, Mark Lee, Katja Markert, et Alan Wallington, éditeurs, *Proceedings of the 2nd Conference on Corpus Linguistics (CL 2003) Interdisciplinary Workshop on Corpus-Based Approaches to Figurative Language*, volume 18, Lancaster, UK, mars 2003. UCREL Technical Paper.

[JAF] Jakarta Avalon Framework. Inversion of Control – `http://avalon.apache.org/framework/cop/guide-patterns-ioc.html`.

[JFL04] Tahar Jarboui, Jean-Philippe Fassino, et Marc Lacoste. Applying Components to Access Control Design : Towards a Framework for OS Kernels. Dans *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, juin 2004.

[Joh97] Ralph E. Johnson. Frameworks = (Components + Patterns) : How frameworks compare to other object-oriented reuse techniques. *Communications of the ACM*, 40(10) :39–42, octobre 1997.

[KCA01] Günter Kniesel, Pascal Costanza, et Michael Austermann. JMangler – A Framework for Load-Time Transformation of Java Class Files. Dans *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, novembre 2001.

[KH98] Ralph Keller et Urs Hölzle. Binary Component Adaptation. Dans *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*,

volume 1445 de *Lecture Notes in Computer Science*, pages 307–329, Brussels, Belgium, juillet 1998. Springer–Verlag.

[KHB98] Murat Karaorman, Urs Hölzle, et John Bruno. jContractor : A Reflective Java Library to Support Design By Contract. Rapport Technique TRCS98-31, Computer Science Department, University of California, Santa Barbara, California, USA, décembre 1998.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, et William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10) :59–65, octobre 2001.

[KK] Hannes Kegel et Ingo Kegel. jclasslib – `http://www.ej-technologies.com/products/jclasslib/overview.html`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, et John Irwin. Aspect-Oriented Programming. Dans Mehmet Akşit et Satoshi Matsuoka, éditeurs, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[Kni] Konstantin Knizhnik. JavaGO, Java Global Optimizer – `http://www.garret.ru/~knizhnik/javago/ReadMe.htm`.

[Kra] Reto Kramer. iContract – `http://www.reliable-systems.com/tools/iContract/iContract.htm`.

[Kra98] Reto Kramer. iContract – The Java Design by Contract tool. Dans *Proceedings of the 26th Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA'26)*, pages 295–307, Los Alamitos, California, USA, août 1998.

[KV01] Tobias Kuipers et Joost Visser. Object-oriented Tree Traversal with JJForester. *Electronic Notes in Theoretical Computer Science*, 44, 2001.

[Laf] Eric P. F. Lafortune. ProGuard – `http://proguard.sf.net/`.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, mai 1994.

[Lam99] Leslie Lamport. Specifying Concurrent Systems with TLA+. Dans *Calculational System Design*, Amsterdam, 1999. IOS Press.

[Lam02] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers.* Addison Wesley, juillet 2002.

[Lap96] Jean-Claude Laprie. *Guide de la sûreté de fonctionnement.* Cépaduès, deuxième édition, 1996.

[LCB04] Romain Lenglet, Thierry Coupaye, et Eric Bruneton. Composing Transformations of Compiled Java programs with Jabyce. *Computer Science and Information Systems (ComSIS)*, 1(2), septembre 2004.

[LdPdA04] Daniel Lucrédio, Antonio Francisco do Prado, et Eduardo Santana de Almeida. A Survey on Software Components Search and Retrieval. Dans *Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), track on Component-Based Software Engineering*, pages 152–159, Rennes, France, septembre 2004. IEEE Computer Society Press.

[Ler03] Xavier Leroy. Java bytecode verification : algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4) :235–269, 2003.

[LH00] Andreas Ludwig et Dirk Heuzeroth. Metaprogramming in the Large. Dans *Proceedings of the 2nd International Conference on Generative and Component-based Software Engineering (GCSE)*, numéro 2177. Springer–Verlag, janvier 2000.

[LLLT] Johan Lind, Jimmy Larsson, Robert Lillsjo, et Jon Tirsen. Nanning – `http://nanning.codehaus.org/`.

[LMS+] Chris Laffra, Lisa Martin, David Streeter, Peter Sweeney, Frank Tip, et Sarvamangala Jagadeesh. JAX, Java Application eXtractor – `http://www.research.ibm.com/jax/`.

[LPC+04] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, et Joseph Kiniry. *JML Reference Manual*, mars 2004. Draft revision 1.68.

[LvHKBO84] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brueckner, et Olaf Owe. ANNA : a language for annotating Ada programs. Rapport Technique CSL-TR-84-261, Computer Systems Laboratory, Stanford University, Stanford, California, USA, juillet 1984.

[LW94] Barbara H. Liskov et Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, novembre 1994.

[LY99] Tim Lindholm et Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, deuxième édition, 1999.

[LZ97] Han Bok Lee et Benjamin G. Zorn. BIT : A Tool for Instrumenting Java Bytecodes. Dans *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*, Monterey, California, USA, décembre 1997.

[Mar96a] Robert C. Martin. Granularity. *C++ Report*, 8(10) :57–62, novembre-décembre 1996.

[Mar96b] Robert C. Martin. The Interface Segregation Principle. *C++ Report*, août 1996.

[Mar03] Robert C. Martin. *Agile Software Development.* Prentice-Hall, 2003.

[MC95] Microsoft Corporation. The Component Object Model Specification, mars 1995.

[MCM+00] Gilles Muller, Charles Consel, Renaud Marlet, Luciano Porto Barreto, Fabrice Mérillon, et Laurent Réveillère. Towards Robust OSes for Appliances : A New Approach Based on Domain-Specific Languages. Dans *Proceedings of the 9th workshop on ACM SIGOPS European workshop : beyond the PC : new challenges for the operating system (EW2000)*, pages 19–24, Kolding, Denmark, septembre 2000.

[Mey96] Jonathan Meyer. *Jasmin Instruction Syntax*, juillet 1996.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, deuxième édition, 1997.

[MGM] Tanmay Mohapatra, Alex Guardiet, et Abdul Mannan. Java Class File Editor – `http://classeditor.sf.net/`.

[MH02a] Jerome Miecznikowski et Laurie J. Hendren. Decompiling Java Bytecode : Problems, Traps and Pitfalls. Dans *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, pages 111–127, Grenoble, France, avril 2002. Springer-Verlag.

[MH02b] Jerome Miecznikowski et Laurie J. Hendren. A Graph-Free Approach to Data-Flow Analysis. Dans *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, pages 46–61, Grenoble, France, avril 2002. Springer-Verlag.

[MHT04] Shin-Cheng Mu, Zhenjiang Hu, et Masato Takeichi. An Injective Language for Reversible Computation. Dans *Proceedings of the 7th International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 de *Lecture Notes in Computer Science*, pages 289–313, Stirling, Scotland, UK, juillet 2004. Springer-Verlag.

[Mit03] Ruslan Mitkov, éditeur. *The Oxford Handbook of Computational Linguistics.* Oxford University Press, Oxford, UK, 2003.

[MK90] John D. McGregor et Timothy D. Korson. Understanding Object-Oriented : A Unifying Paradigm. *Communications of the ACM*, 33(9) :40–60, 1990.

[MLSL03] Mathieu Mangeot-Lerebours, Gilles Sérasset, et Mathieu Lafourcade. Construction collaborative d'une base lexicale multilingue – Le projet Papillon. *Traitement Automatique des Langues*, 44(2) :151–176, 2003.

[Moh02] Markus Mohnen. An open framework for data-flow analysis in Java : extended abstract. Dans *Proceedings of the Inaugural Conference on the Principles and Practice of programming and Proceedings of the 2nd Workshop on Intermediate Representation Engineering for virtual machines (IRE'02)*, pages 157–161, Dublin, Ireland, juin 2002. National University of Ireland.

[Mon96] Jean-François Monin. *Comprendre les Méthodes formelles, panorama et outils logiques.* Collection Technique et Scientifique des Télécommunications. Masson, 1996. Préface de Gérard Huet.

[MRBM+] Peter Murray-Rust, Tim Bray, David Megginson, Jon Bosak, et al.. SAX, Simple API for XML – `http://www.saxproject.org/`.

[MRV03] Pierre-Etienne Moreau, Christophe Ringeissen, et Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. Dans *Proceedings of the 12th Conference on Compiler Construction (CC 2003)*, volume 2622 de *Lecture Notes in Computer Science*, pages 61–76, Warsaw, Poland, avril 2003. Springer-Verlag.

[MS99] Gilles Muller et Ulrik Pagh Schultz. Harissa : A Hybrid Approach to Java Execution. *IEEE Software*, 16(2) :44–51, mars 1999.

[MT97] Nenad Medvidovic et Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. Dans M. Jazayeri et H. Schauer, éditeurs, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 60–76. Springer–Verlag, 1997.

[NW] Nathaniel Nystrom et David Whitlock. BLOAT – `http://www.cs.purdue.edu/s3/projects/bloat/`.

[Nys98] Nathaniel Nystrom. *Bytecode-level analysis and optimization of Java class files.* Purdue University, West Lafayette, Indiana, USA, mai 1998. Master's Dissertation.

[OB99] Alexandre Oliva et Luiz Eduardo Buzato. The Design and Implementation of Guaraná. Dans *Proceedings of the 5th USENIX Conference on Objected-Oriented Technologies and Systems (COOTS'99)*, San Diego, California, USA, mai 1999.

[Obja] ObjectWeb. ASM – `http://asm.objectweb.org/`.

[Objb] ObjectWeb. The Fractal component model – `http://fractal.objectweb.org/`.

[Objc] ObjectWeb. JORM – `http://jorm.objectweb.org/`.

[Objd] ObjectWeb. Speedo – `http://speedo.objectweb.org/`.

[OHBS94] Harold Ossher, William Harrison, Franck Budinsky, et Ian Simmonds. Subject-Oriented Programming : Supporting Decentralized Development of Objects. Dans *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, 1994.

[Ohu] Hidetoshi Ohuchi. Jarg – `http://jarg.sf.net/`.

[OJY+] Brian S. O'Neill, Scott Jappinen, Jonathan Yam, Reece Wilton, Joshua Yockey, et Jonathan Colwell. Tea Trove Class File – `http://teatrove.sf.net/trove.html`.

[OMG03] OMG. UML 2.0 OCL Specification, Final Adopted Specification, novembre 2003.

[OSM+00] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, et Yasunori Kimura. OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java. Dans *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 de *Lecture Notes in Computer Science*, pages 362–387, Sophia Antipolis and Cannes, France, juin 2000. Springer-Verlag.

[Pan] Nicos Panayides. jclassinfo – `http://jclassinfo.sf.net/`.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, décembre 1972.

[PGA01] Andrei Popovici, Thomas Gross, et Gustavo Alonso. Dynamic Homogenous AOP with Prose. Rapport Technique TR01, Department of Computer Science, ETH Zürich, Zürich, Switzerland, mars 2001.

[Pra02] Michele Prandi. La métaphore : de la définition à la typologie. *Langue Française*, 134 :6–20, mai 2002.

[Proa] Compose Project. JSpec – `http://compose.labri.fr/prototypes/jspec/`.

[Prob] Compose Project. Tempo – `http://compose.labri.fr/prototypes/tempo/`.

[Proc] The SAX Project. Events vs. Trees – `http://www.saxproject.org/?selected=event`.

[Pro03] Marek Prochazka. Jironde : A Flexible Framework for to Make Components Transactional. Dans *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, Paris, France, novembre 2003.

[PS83] Helmuth Partsch et Ralf Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys (CSUR)*, 15(3) :199–236, 1983.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, et Laurent Martelli. JAC : an aspect-based distributed dynamic framework. *Software : Practice and Experience*, 34(12) :1119–1148, octobre 2004.

[PSDF01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, et Gérard Florin. JAC : A Flexible Solution for Aspect-Oriented Programming in Java. Dans *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscuting Concerns (Reflection 2001)*, volume 2192 de *Lecture Notes in Computer Science*, Kyoto, Japan, septembre 2001.

[PTB+97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, et Scott A. Watterson. Toba : Java for Applications – A Way Ahead of Time (WAT) Compiler. Dans *Proceedings of the 3rd USENIX Conference on Objected-Oriented Technologies and Systems (COOTS'97)*, pages 41–53, Portland, Oregon, USA, juin 1997.

[Ró3] Mickaël Rémond. *Erlang*. Eyrolles, 2003.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, et T. Owen O'Malley. Specification-based test oracles for reactive systems. Dans *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 105–118, Melbourne, Australia, mai 1992. ACM Press.

[RC03] Nicolas Rivierre et Thierry Coupaye. Observing component behaviours with TLA. Dans *Proceedings of the International Workshop on Correctness of Model-based Software Composition (CMC'03, ECOOP 2003)*, Darmstadt, Germany, juillet 2003.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4) :314–335, avril 1995.

[SLCM98] Ulrik Pagh Schultz, Julia L. Lawall, Charles Consel, et Gilles Muller. Towards Automatic Specialization of Java Programs. Rapport Technique 1216, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France, décembre 1998.

[SS96] Tony Savor et Rudolph E. Seviora. An Architectural Overview of a Software Supervisor. Dans *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 52–56, L'Aquila, Italy, juin 1996.

[SSY00] Takahiro Sakamoto, Tatsurou Sekiguchi, et Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. Dans *Proceedings of the ASA/MA Conference*, pages 16–28, 2000.

[Ste03] Jean-Bernard Stefani. Éléments d'architecture pour la supervision de systèmes répartis de grande taille. Document interne - Projet INRIA SARDES, décembre 2003.

[Sto] Jesse Stockall. GenJar – `http://genjar.sf.net/`.

[Sul01] Gregory T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10) :95–97, octobre 2001.

[Tat02] Michiaki Tatsubori. *A Class-Object Model for Program Transformation.* Graduate School of Engineering, University of Tsukuba, Ibaraki, Japan, janvier 2002. Doctor of Philosophy in Engineering Dissertation.

[TCKI99] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, et Kozo Itano. OpenJava : A Class-Based Macro System for Java. Dans *Papers from the 1st OOPSLA Workshop on Reflection and Software Engineering (OORaSE 1999)*, volume 1826 de *Lecture Notes in Computer Science*, pages 117–133, Denver, Colorado, USA, novembre 1999. Springer–Verlag.

[Tee04a] Aree Teeraparbseree. Qualitative Evaluation of Automatically Calculated Acception Based MLDB. Dans *Proceedings of the COLING Workshop on Multilingual Linguistic Resources (MLR 2004)*, Geneva, Switzerland, août 2004.

[Tee04b] Aree Teeraparbseree. Un système adaptable pour l'initialisation automatique d'une base lexicale interlingue par acceptions. Dans *Proceedings of Rencontre des Etudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RECITAL 2004)*, Fès, Morocco, avril 2004.

[TFL+04] Frédéric Dang Tran, Jean-Philippe Fassino, Olivier Lobry, Jacques Pulou, et Nicolas Rivierre. Toward a component-based embedded Java-oriented Operating Systems. Dans *Proceedings of the 2nd Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'04)*, Lecture Notes in Computer Science, Larnaca, Cyprus, octobre 2004. Springer-Verlag.

[TLSS99] Frank Tip, Chris Laffra, Peter F. Sweeney, et David Streeter. Practical experience with an application extractor for Java. Dans *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 292–305, Denver, Colorado, USA, novembre 1999. ACM Press.

[Tor97] Seved Torstendahl. Open Telecom Platform. *Ericsson Review*, 17(1) :14–17, 1997.

[VB04] Ankush Varma et Shuvra S. Bhattacharyya. Java-through-C Compilation : An Enabling Technology for Java in Embedded Systems. Dans *Proceedings of the conference on Design, automation and test in Europe (DATE'04)*, volume 3, pages 30161–30167, Paris, France, février 2004. IEEE Computer Society Press.

[vdBdJKO00] Mark G. J. van den Brand, Hayco de Jong, Paul Klint, et Pieter Olivier. Efficient Annotated Terms. Rapport Technique SEN-R0003, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, février 2000.

[vdBKV03] Mark G. J. van den Brand, Paul Klint, et Jurgen J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2) :152–190, avril 2003.

[vDV02] Arie van Deursen et Joost Visser. Building Program Understanding Tools using Visitor Combinators. Dans *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 137–146, Paris, France, juin 2002. IEEE Computer Society Press.

[Vis] Eelco Visser. The Online Survey of Program Transformation – `http://www.program-transformation.org/survey.html`.

[Vis01] Eelco Visser. A survey of rewriting strategies in program transformation systems. Dans Bernhard Gramlich et Salvador Lucas, éditeurs, *Proceedings of the 1st Workshop on Reduction Strategies in Rewriting and Programming (WRS'2001)*, volume 57 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, mai 2001.

[Vis02] Eelco Visser. Meta-Programming with Concrete Object Syntax. Dans Don Batory, Charles Consel, et Walid Taha, éditeurs, *Proceedings of the 1st International Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 de *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, Pennsylvania, USA, octobre 2002. Springer-Verlag.

[Vis04] Eelco Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, juin 2004. Accepted for publication.

[Vli96] John Vlissides. Protection. I. The Hollywood Principle. *C++ Report*, 8(2) :14, 16–19, février 1996.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, et Phong Co. Soot - a Java Optimization Framework. Dans *Proceedings of CASCON 1999*, pages 125–135, 1999.

[Whi] A. Abram White. Serp – `http://serp.sf.net/`.

[YL] University of Tokyo Yonezawa Laboratory, Department of Information Science. JavaGO, transparent migration for Java programs – `http://www.taplas.org/amo/JavaGo/doc/index-e.html`.

[Zha] Gang Zhang. JCMP, Java class compressor – `http://viscomp.utdallas.edu/FACADE/jcmp/`.

[Zi99] Sun Zi. *L'Art de la Guerre*. Bibliothèque Stratégique. Economica, Paris, France, 1999. Introduction de Maurice Prestat.

[ZX04] Yingzhou Zhang et Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, 39(3) :14–30, mars 2004.

Annexe A

Format des classes Java compilées

Ce chapitre présente le format des classes Java compilées, comme décrit dans la spécification de la JVM [LY99]. Le contenu de ce chapitre est tiré et adapté de [BLC02].[1]

A.1 Structure générale

La structure générale (grammaire) d'une classe Java au format binaire est une structure arborescente. La racine de cette structure contient les nœuds suivants (« * » signifie « 0 ou plus ») :

- ConstantPool : table des constantes (cf. la section A.2) ;
- FieldInfo* : descriptions des champs (*fields*) de la classe ;
- MethodInfo* : descriptions des méthodes de la classe ;
- Attribute* : nœuds optionnels supplémentaires.

Un nœud FieldInfo contient le nom d'un champ, son type, ainsi que des drapeaux indiquant si le champ est public, privé, statique... Un nœud FieldInfo peut également contenir des *attributs* optionnels supplémentaires. Un attribut est un nœud dont le type (le nom) et la taille sont indiqués explicitement. Un attribut « ConstantValue » permet d'indiquer la valeur d'un champ statique constant. Les attributs « Deprecated » et « Synthetic » permettent d'indiquer respectivement qu'un champ est obsolète, ou qu'il a été ajouté par le compilateur (et ne correspond donc pas un champ dans le code source de la classe).

Un nœud MethodInfo contient le nom d'une méthode, les types de ses paramètres et de sa valeur de retour, ainsi que des drapeaux indiquant si la méthode est publique, prive, statique, abstraite... Un nœud MethodInfo peut galement contenir des attributs optionnels supplémentaires. Le plus important est l'attribut « Code » qui contient le code des méthodes non abstraites (cf. la section A.3). L'attribut « Exceptions » contient la liste des exceptions que peut lancer la méthode. Les attributs « Deprecated » et « Synthetic » ont le même rôle que pour les champs.

[1] Le document à l'URL `http://murrayc.com/learning/java/java_classfileformat.shtml` est également une description simple du format des classes compilées.

Les attributs optionnels du nœud racine (pour une classe) sont l'attribut « SourceFile », pour indiquer le nom du fichier source à partir duquel une classe a été compilée, l'attribut « InnerClasses », pour stocker les noms des classes internes d'une classe, et enfin l'attribut « Deprecated ».

Les attributs permettent de rendre extensible le format des classes Java. On peut en effet ajouter des informations dans une classe Java en lui ajoutant des attributs non standard, sans que cela affecte la possibilité de charger cette classe dans une machine virtuelle ne connaissant pas ces attributs. Les machines virtuelles doivent en effet simplement ignorer les attributs qu'elles ne reconnaissent pas, ce qui est possible grâce au fait que chaque attribut contient son type (son nom) et son taille.

A.2 La table des constantes

La table des constantes d'une classe, ou ConstantPool, est un tableau contenant un ensemble de constantes de différentes natures : valeur numérique, chaîne de caractère, ou *type*. Une constante peut être référencée, via son index dans ce tableau, par d'autres nœuds de la structure arborescente d'une classe.

Le but de cette table est d'éviter les redondances. Ainsi, par exemple, un nœud FieldInfo ne contient pas directement le nom d'un champ, mais l'index de ce nom dans la table des symboles. Autre exemple, les instructions getfield et putfield ne contiennent pas directement le nom du champ qui doit être lu ou modifié, mais l'index de ce nom. Ainsi le nom du champ est stocké une et une seule fois, même si de nombreuses instructions y font référence.

A.3 Le code des méthodes

Le code d'une méthode est stocké dans un attribut de nom « Code ». Ce code est une suite linéaire d'instructions élémentaires. La JVM est une machine à pile : les instructions manipulent généralement les données sur la pile, bien qu'une notion de *variable locale*, similaire à la notion de registre, soit également disponible. La plupart des instructions sont codées sur un seul octet et n'ont pas d'argument en plus de leur identificateur, appelé *opcode*. Ce codage sur un octet est à l'origine du l'appellation *bytecode* pour les instructions.

L'attribut « Code » contient une table des traitants d'exceptions, qui correspondent aux blocs try...catch...finally du langage Java, ainsi qu'un entier indiquant la taille maximale que peut atteindre la pile lors de l'excution du code de la méthode. Enfin, l'attribut « Code » peut également contenir des sous attributs optionnels supplémentaires, qui contiennent des informations de *debugging*. Ainsi l'attribut « LineNumberTable » stocke des informations permettant de retrouver, pour chaque instruction de bytecode, le numéro de ligne correspondant dans le fichier source. L'attribut « LocalVariableTable » stocke les noms des arguments formels de la méthode ainsi que les noms des variables locales, tels que déclarés dans le code source.

Annexe B

Patrons de conception

Le chapitre 2, qui présente notre problématique, introduit l'idée d'une évolution cyclique du domaine de l'ingénierie logicielle qui alterne l'identification de bonnes pratiques, leur formulation sous la forme de principes de conception, et la spécification de nouveaux outils d'ingénierie (langages de programmation, etc.). Dans la catégorie des outils d'ingénierie logicielle, un *patron de conception* est un guide de conception qui permet de répéter facilement une bonne pratique de conception éprouvée. Les patrons de conception sont généralement utilisés dans le cadre de la programmation par objets. Le terme de patron en ingénierie logicielle est une acceptation métaphorique du terme de patron dans le domaine de l'architecture des bâtiments [Ale79]. En général, les patrons de conception sont suffisamment simples pour permettre d'être appliqués facilement, ce qui a fait le succès de cette approche.

En programmation par objets, un patron de conception décrit une partie d'une architecture, en terme d'objets et de leurs interactions. L'inconvénient des patrons de conceptions est qu'ils ne considèrent la conception d'une architecture que localement, sur un sous-partie de cette architecture. Les patrons de conception ne forment pas un ensemble suffisamment complet et cohérent pour guider la conception complète d'applications. En revanche, cette limitation de la portée d'un patron de conception permet de décrire un patron de manière facilement compréhensible. De plus, la description d'un patron de conception se base généralement sur l'utilisation de métaphores projectives. Cette utilisation des métaphores se reflète de manière flagrante dans les noms des patrons de conception, qui font référence à des concepts aussi variés que l'armée (Instruction, Commande, etc.), l'humain (Observateur, Médiateur, etc.) ou l'architecture des bâtiments (Façade, Pont, etc.). Cette utilisation de métaphores permet de mieux appréhender la conception d'une architecture. En effet, comme décrit dans la section 2.4, les métaphores permettent de « glisser mentalement » de manière efficace pour appréhender des idées nouvelles. C'est pourquoi les patrons de conception sont avant tout des outils de communication très efficaces, et très utilisés.

[GHJV94] est une collection de patrons de conception bien connus, qui sont le résultat d'un travail collectif d'identification et de description. Cette collection a eu un tel succès que les noms de ces patrons de conception, qui sont des utilisations métaphoriques de termes d'autres domaines, sont entrées dans le « vocabulaire courant » en ingénierie logicielle. Ce chapitre donne une description succinte d'un sous-ensemble des patrons de cette collection, qui sont utilisés dans ce mémoire pour décrire des parties d'architectures, à partir des description données dans [GHJV94] : Fabrique Abstraite, Décorateur, et Stratégie. En revanche, les exemples

donnés sont originaux.

B.1 Fabrique abstraite

Dans le patron Fabrique abstraite (*Abstract factory*), une application utilise une interface abstraite (*fabrique abstraite*) pour la création d'objets d'un type abstrait (*produit abstrait*). Différentes classes (*fabriques concrètes*) peuvent implanter l'interface de fabrique abstraite, pour créer des objets (*produits concrets*) de classes qui héritent du type de produit abstrait. L'application ne dépend pas directement des classes de fabriques concrètes et de produits concrets, ce qui permet de changer le type de produit concret créé sans modifier le code de l'application.

Par exemple, une application qui crée des objets modélisant des animaux de type Animal peut utiliser l'interface de fabrique abstraite FabriqueDAnimaux, comme illustré dans la figure B.1. Différentes classes peuvent implanter FabriqueDAnimaux, pour créer des objets Animal de types spécifiques, par exemple de type Souris ou Elephant.

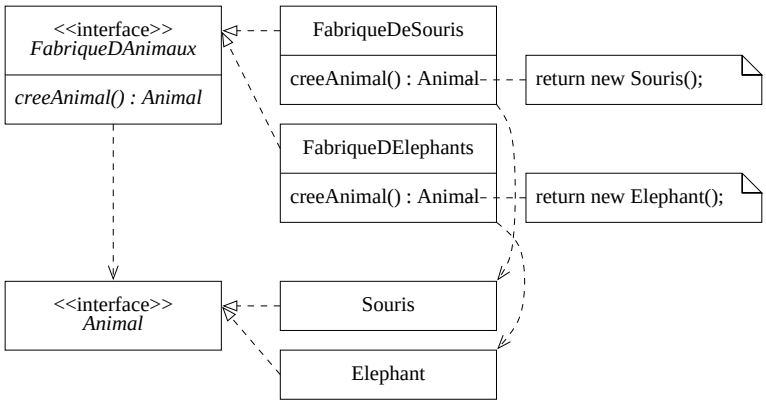


Fig. B.1 : Patron de conception Fabrique abstraite

B.2 Décorateur

L'objectif de l'application du patron Décorateur (*Decorator*) est de permettre d'ajouter dynamiquement des responsabilités supplémentaires à un objet, sans modifier le code de sa classe. Dans ce patron, un objet à étendre est enveloppé (contenu) dans un autre objet appelé *décorateur*. Le décorateur offre la même interface que l'objet décoré, ce qui permet de remplacer un objet par un décorateur de manière transparente pour les classes clientes. Le décorateur reçoit les appels de méthodes à la place de l'objet décoré, qu'il transfère à l'objet décoré. Le décorateur peut effectuer des traitements additionnels avant ou après chaque appel. Ce patron peut être appliqué récursivement, en décorant un objet décorateur, pour composer arbitrairement les traitements additionnels qui sont effectués.

La figure B.2 illustre le diagramme de classes pour des objets qui effectuent pour traitement un « arrosage de plantes ». Les classes implantent la même interface ArroseurDePlantes, qui

définit la méthode arrose(). La classe ArroseurSimple est une implantation complète d'arroseur simple. La classe abstraite DecorateurDArroseur est héritée par toutes les classes d'objets décorateurs d'objets ArroseurDePlantes. L'objet décoré par un décorateur est référencé dans l'attribut decore. L'implantation par défaut de la méthode arrose() dans DecorateurDArroseur se limite à retransmettre l'appel à l'objet décoré. La méthode arrose() de la classe ArroseurAvecEngrais effectue un traitement (« l'ajout d'engrais ») *après* la retransmission de l'appel. La méthode arrose() de la classe ArroseurNettoyeur effectue un traitement (« le nettoyage des feuilles ») *avant* la retransmission de l'appel. Il possible d'assembler les objets, pour qu'un objet ArroseurAvecEngrais décore un objet ArroseurNettoyeur, qui décore à son tour un objet ArroseurSimple, pour composer les traitements additionnels (« ajout d'engrais » et « nettoyage des feuilles »).

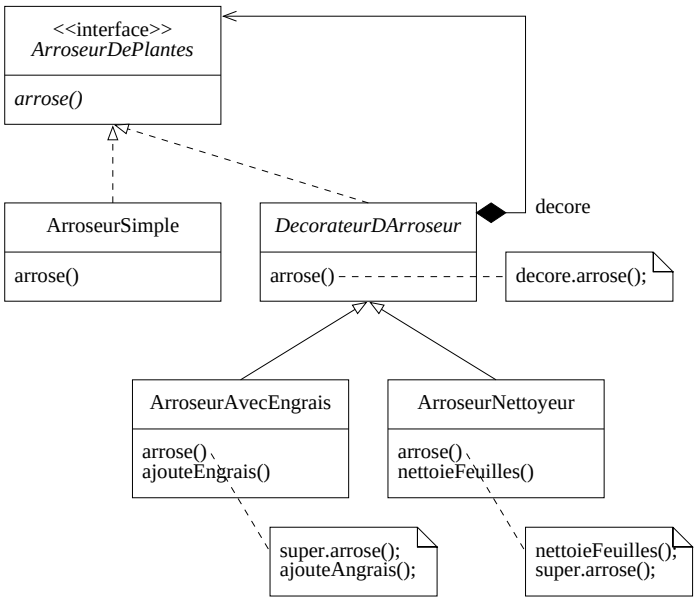


Fig. B.2 : Patron de conception Décorateur

B.3 Stratégie

Le terme de *stratégie* provient du domaine militaire. La stratégie est l'art du général d'une armée : c'est *l'art de l'action* [Zi99]. La stratégie s'oppose à la *politique*. La politique est l'art du gouvernement (du *souverain*), c'est-à-dire *l'art de la décision*. L'art du général consiste à élaborer un plan d'action (également qualifié de stratégie) pour atteindre des objectifs militaires stratégiques décidés par le gouvernement, tels que l'invasion d'un pays par exemple (« le souverain clairvoyant projette les attaques, le bon général les prépare » [Zi99]). Alors que la stratégie embrasse l'ensemble du plan d'action, la *tactique* regarde son application concrète dans les cas particuliers. Les responsabilités du général et du gouvernement sont indépendantes : le gouvernement ne doit pas intervenir dans la stratégie militaire, (« celui dont le général est capable et dont le souverain n'intervient pas dans les affaires militaires remporte la victoire » [Zi99]), et vice-versa.

Ces termes sont utilisés métaphoriquement en ingénierie logicielle, et en particulier dans le patron de conception Stratégie (*Strategy*), bien que ce ne soit pas explicite. D'après [GHJV94], dans ce patron, un objet *contexte* délègue l'exécution d'un algorithme (un « plan d'action ») à un objet *stratégie* qui hérite d'une interface abstraite de *stratégie*. Les opérations définies dans l'interface abstraite correspondent aux objectifs qui doivent être atteints par une stratégie. En rapport avec la métaphore militaire, nous considérons qu'il est impropre de qualifier ces objets de *contextes* et de *stratégies*. Nous proposons donc une dénomination différente de celle de [GHJV94]. Notamment, nous proposons plutôt de qualifier l'objet stratégie de *général*, car la stratégie (un plan d'action) n'est pas l'objet lui-même, mais une exécution de l'algorithme implanté dans la classe de l'objet. De même, nous proposons de qualifier l'objet contexte de *souverain*, car son rôle est de *décider* d'exécuter un plan d'action pour atteindre son objectif, en déléguant le choix et l'exécution du plan d'action à l'objet *général*. Il est possible d'implanter différentes classes de *généraux*, qui héritent de cette interface abstraite pour implanter différents algorithmes. Le remplacement d'un objet *général* par un autre est transparent pour un objet *souverain*.

La figure B.3 illustre un exemple d'application du patron de conception Stratégie pour modéliser le comportement d'une personne qui déjeune. L'objet *souverain* (de type GestionAlimentation) décide notamment de l'heure du déjeuner, de l'endroit et du menu, et délègue à un objet *général* (de type DeroulementRepas) le choix du plan d'action pour manger les différents éléments du menu. Plusieurs algorithmes peuvent être implantés, dans différentes classes d'objets *généraux*, pour spécifier par exemple une stratégie où le déjeuner commence par le fromage, ou une stratégie où le dessert est mangé en parallèle avec l'entrée, etc.

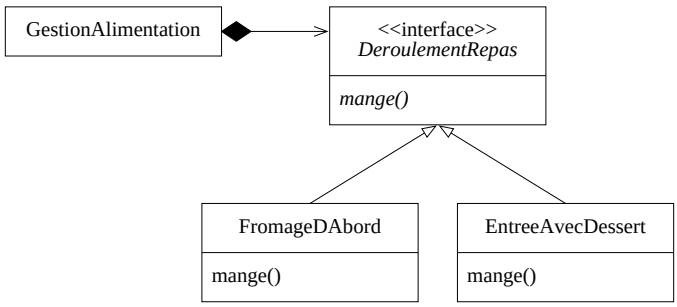


Fig. B.3 : Patron de conception Stratégie

Annexe C

Concepts de ISO RM-ODP

Cette annexe résume les concepts généraux définis dans la partie II du standard ISO Reference Model of Open Distributed Processing [ISO95a]. Cette partie du standard est utile pour interpréter les constructions des langages de spécification de RM-ODP. Ces concepts sont répartis dans différentes catégories, dont les principales sont présentées dans cette annexe.

C.1 Concepts d'interprétation de base

Cette catégorie définit des méta-concepts, qui peuvent être utilisés dans toute démarche de modélisation :

entité (*entity*) une chose concrète ou abstraite qui a de l'intérêt.

abstraction la démarche qui consiste à ignorer les détails inutiles pour établir un modèle simplifié ; ou le résultat de cette démarche.

atomicité (*atomicity*) une entité est atomique à un certain niveau d'abstraction si elle ne peut pas être subdivisée dans ce niveau d'abstraction.

système (*system*) une chose qui a de l'intérêt comme un tout, ou comme composée de ces parties. Un système peut être considéré comme une entité. Un système peut comprendre d'autres systèmes : dans ce cas, ce sont des sous-systèmes (*subsystems*).

architecture un ensemble de règles qui définissent la structure d'un système et les relations entre ses parties.

C.2 Concepts de modélisation de base

Cette catégorie définit des concepts de modélisation, indépendamment du langages d'implantation :

objet (*object*) un modèle d'une entité. Un modèle est caractérisé par son comportement et son état. Les objets sont tous distincts.

environnement (*environment*) (d'un objet) la partie du modèle qui n'est pas l'objet.

action une chose qui arrive. Il y a deux types d'actions : les actions internes et les actions externes (interactions). Une *action interne* n'implique qu'un seul objet, sans la participation de son environnement. Une *interaction* implique la participation de l'environnement de l'objet.

interface abstraction du comportement d'un objet qui consiste en un ensemble d'interactions de cet objet, avec des contraintes pour l'occurrence de ces interactions. Toute interaction d'un objet fait partie d'une interface unique : les interfaces d'un objet forment donc une partition des interactions de ce objet.

activité (*activity*) séquence d'actions.

comportement (*behaviour*) (d'un objet) une collection d'actions avec un ensemble de contraintes sur leur occurrence. Ces contraintes peuvent être par exemple des contraintes de séquentialité, de non-déterminisme, de concurrence ou de temps-réel.

état (*state*) (d'un objet) à un instant donné, les conditions qui déterminent l'ensemble de toutes les séquences d'actions auquelles l'objet peut participer. Les objets sont encapsulés : l'état d'un objet ne peut pas être modifié directement par l'environnement, mais seulement par les actions auquelles il participe.

communication transmission d'information entre deux objets ou plus, comme résultat d'une ou plusieurs interactions.

localité (*location*) un intervalle arbitraire dans l'espace (ou dans le temps), dans lequel une action peut se produire.

point d'interaction une localisation où il existe un ensemble d'interfaces.

C.3 Concepts de spécification

composition (d'objets) la combinaison de deux objets ou plus, formant un nouvel objet, à un niveau d'abstraction différent. Les caractéristiques du nouvel objet sont déterminées par les objets combinés, et la façon dont ils sont combinés. Le comportement du nouvel objet est la composition des comportements des objets combinés.

composition (de comportements) la combinaison de deux comportements ou plus, formant un nouveau comportement. Les caractéristiques du nouveau comportement sont déterminées par les comportements combinés, et la façon dont ils sont combinés.

objet composite (*composite object*) un objet exprimé comme une composition.

décomposition la spécification d'un objet ou d'un comportement donné par une composition. C'est l'activité duale de la composition.

compatibilité comportementale (*behavioural compatibility*) un objet est comportementalement compatible (*behaviourally compatible*) avec un second objet, respectivement à un ensemble de critères, si le premier objet peut remplacer le second objet sans que l'environnement puisse détecter de différence dans le comportement des objets selon l'ensemble de critères.

raffinement (*refinement*) démarche de transformation d'une spécification en une spécification plus détaillée.

trace un enregistrement des interactions auquelles participe un objet, depuis son état initial jusqu'à un autre état. Le comportement d'un objet détermine de façon unique l'ensemble de toutes ses traces possibles, mais l'inverse n'est pas vrai.

type (d'un <X>) un prédicat caractérisant une collection de <X>s. Un <X> "est de ce type", ou "satisfait ce type", si le prédicat est vérifié pour ce <X>. Dans RM-ODP, des types sont nécessaires pour les objets, les interfaces et les actions.

classe (*class*) (de <X>s) l'ensemble des <X>s satisfaisant un type. Les éléments de cet ensemble sont membres de cette classe.

sous-type / super-type (*subtype/supertype*) un type A est sous-type d'un type B, et B est un super-type de A, si tout <X> qui satisfait A satisfait aussi B.

sous-classe / super-classe (*subclass/superclass*) une classe A est une sous-classe d'une classe B, et B est une super-classe de A, si le type associé à A est un sous-type du type associé à B.

patron de <X> (*<X> template*) la spécification des fonctionnalités communes d'une collection de <X>s, suffisamment détaillée pour pouvoir instancier un <X>. C'est une abstraction de la collection de <X>s. <X> peut être n'importe quoi aillant un type. Peut spécifier des paramètres à renseigner lors d'une instanciation.

signature d'interface (*interface signature*) l'ensemble des patrons d'actions associés aux interactions d'une interface.

instanciation (*instantiation*) (d'un patron de <X>) un <X> produit à partir d'un patron de <X>, et d'autres informations nécessaires à cette instanciation. Le <X> produit exhibe les mêmes fonctionnalités que celles spécifiées dans le patron de <X>.

création (*creation*) (d'un <X>) instanciation d'un <X> par une action impliquant des objets du modèle.

introduction (d'un <X>) instanciation d'un <X> non réalisée par une action impliquant des objets du modèle.

suppression (*deletion*) (d'un <X>) l'action de destruction d'un <X> instancié.

rôle (*role*) identifiant d'un comportement, qui peut apparaître comme paramètre dans un patron d'un objet composite, et qui est associé à l'un des objets composants l'objet composite.

instance (d'un type) un <X> qui satisfait le type.

type de patron (*template type*) (d'un <X>) un prédicat défini dans un patron, qui est vérifié pour toutes les instanciations du patron.

type de classe (*class template*) (d'un <X>) l'ensemble de tous les <X>s satisfaisant un type de patron de <X>, c'est-à-dire l'ensemble des <X>s qui sont les instances du patron de <X>.

invariant un prédicat, requis par une spécification, qui doit être vérifié pendant toute la durée de vie de l'ensemble d'objets.

précondition (*precondition*) un prédicat, requis par une spécification, qui doit être vérifié pour qu'une action puisse se produire.

postcondition un prédicat, requis par une spécification, qui doit être vérifié immédiatement après l'occurrence d'une action.

C.4 Concepts d'organisation

configuration (d'objets) une collection d'objets capables de communiquer via leurs interfaces. Une configuration détermine l'ensemble des objets impliqués dans chaque interaction. Sa spécification peut être statique ou dynamique (par attachement et détachement).

domaine <X> (*<X> domain*) un ensemble d'objets, dont chaque objet est associé à un objet contrôleur (*controlling object*) par une relation caractéristique <X> (*<X> characterizing relationship*). Tout domaine a un objet contrôleur associé. Exemple : domaine d'administration (*administration domain*).

époque (*epoch*) une période de temps pendant laquelle l'objet expose un comportement donné. Chaque objet est dans une seule époque à un moment donné. Plusieurs objets qui interagissent peuvent être dans des époques différentes.

point de référence (*reference point*) un point d'interaction défini dans une architecture , pouvant être choisi comme point de conformité dans une spécification à laquelle se conforme cette architecture.

point de conformité (*conformance point*) un point de référence où un comportement peut être observé pour tester la conformité d'une architecture.

C.5 Propriétés des systèmes et objets

contrat un accord réglant une partie du comportement commun à un ensemble d'objets. Un contrat spécifie des obligations, permissions et interdictioins pour les objets impliqués.

contrat d'environnement (*environment contract*) un contrat entre un objet et son environnement. Les contraintes d'environnement décrivent à la fois les contraintes sur l'environnement pour un comportement correct de l'objet, et les contraintes sur le comportement de l'objet dans un environnement correct.

persistance (*persistence*) la propriété qu'un objet continue d'exister à travers des changements de contexte contractuel.

C.6 Concepts pour les comportements

comportement d'établissement (*establishing behaviour*) le comportement par lequel un contrat donné est mis en place entre des objets donnés. Il peut être explicite (résultat des interactions entre les objets prenant part au contrat) ou implicite (mis en place par un objet externe, ou dans une époque précédente).

comportement permis (*enabled behaviour*) le comportement caractérisant un ensemble d'objets, rendu possible par un comportement d'établissement.

contexte contractuel (*contractual context*) la connaissance qu'un contrat donné est en place. Un objet peut être dans un nombre quelconque de contextes contractuels simultanément ; son comportement est contraint par l'intersection des comportements prescrits par chaque contexte contractuel.

liaison la relation entre des objets, qui résulte d'un comportement d'établissement.

comportement de terminaison (*terminating behaviour*) le comportement qui brise une liaison, et rejete le contexte contractuel et le contrat correspondants.

objet initiateur (*initiating object*) un objet qui produit une communication.

objet répondeur (*responding object*) un objet participant à une communication, qui n'en est pas l'objet initiateur.

objet producteur (*producer object*) un objet qui est la source de l'information transmise, dans une communication.

objet consommateur (*consumer objetct*) un objet qui est une destination de l'information transmise, dans une communication.

objet client (*client object*) un objet qui requiert l'exécution d'une fonction par un autre objet.

objet serveur (*server object*) un objet qui exécute une fonction à la demande d'un objet client.

comportement de ligature (*binding behaviour*) un comportement d'établissement entre deux interfaces ou plus.

lien (*binding*) un contexte contractuel, résultant d'un comportement de ligature. Un binding est un cas particulier de liaison.

comportement de détachement (*unbinding behaviour*) un comportement qui termine un lien.

défaillance (*failure*) violation d'un contrat. Un contrat exprimant le comportement correct des objets, une défaillance est un comportement incorrect.

erreur (*error*) partie de l'état d'un objet qui peut conduire à des défaillances. La manifestation d'une faute dans un objet.

faute (*fault*) une situation qui peut causer des erreurs dans un objet.

stabilité (*stability*) la propriété qu'a un objet, respectivement à un mode de défaillance donné, s'il ne peut pas exhiber ce mode de défaillance.