Spécialisation de composants

Gustavo Bobeff

Université de Nantes

École Doctorale
Sciences et Technologies
de l'Information et des Matériaux

Année 2006
N° B.U. ED 366 - 328

Thèse de doctorat de l'Université de Nantes

spécialité Informatique
à présenter et à soutenir publiquement par

Gustavo Bobeff

le 14 Décembre 2006
à l'École Nationale Supérieure des
Techniques Industrielles et des Mines de Nantes

Spécialisation de composants

devant la commission d'examen composée de

Président	:	Mourad Oussalah	Professeur, Université de Nantes
Rapporteurs	:	Charles Consel	Professeur, ENSEIRB de Bordeaux
		Isabelle Borne	Professeur, Université de Bretagne-Sud
Examinateurs	:	Thomas Jensen	Chargé de recherche CNRS, IRISA-Rennes
	:	Jacques Noyé	Docteur, École des Mines de Nantes
Directeur de thèse	:	Pierre Cointe	Professeur, École des Mines de Nantes
Invité	:	Mufutau Gbadamosi	Responsable technique, PROXIAD Ouest

Équipe d'accueil : Objets, Aspects et Composants(OBASCO)
Laboratoire d'accueil : Département Informatique de l'École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler – 44307 Nantes cedex 3 – France

Résumé

La programmation à base de composants facilite l'encapsulation de logiciel générique qui peut ainsi être réutilisé dans différents contextes d'utilisation. Un composant est spécifié à partir d'une implémentation et d'une interface. Cette interface, utilisée pour la composition, peut être adaptée par le consommateur du composant. Les modèles existants ne permettent que l'adaptation au niveau de l'interface alors que leur implémentation reste inchangée (boîte noire), les applications résultantes conservent le degré de généralité des composants constituants. Pour aller au-delà de ces modèles, nous proposons MoSCo, un modèle qui permet une forme plus profonde d'adaptation où l'implémentation est aussi modifiée. Dans ce modèle, un composant est analysé et traduit, à l'aide des techniques de spécialisation de programmes, sous la forme d'un générateur de composants. Ce générateur produit finalement un composant spécialisé selon un contexte d'utilisation en respectant la notion de boîte noire.

Mots-clés : réutilisation de logiciel, programmation à base de composants, techniques de spécialisation de programmes, évaluation partielle, découpage, extension génératrice, générateur de programmes.

Abstract

Component-based programming allows developers to encapsulate generic pieces of code as components, which can be reused in different usage contexts. The component definition is based on an implementation and an interface. The interface, actually used for component composition, can be adapted by the component consumer. In existing models only the interfaces can be modified while the implementation remains unchanged (black box), the resulting applications keep the degree of genericity associated to the underlying components. To go beyond these models, we propose MoSCo, a model that allows a deeper adaptation of components where not only the interface can be adapted but also the implementation. In this model, a component is analyzed and afterwards translated into a component generator by using program specialization techniques. Finally, the generator generates a specialized component for a given usage context without breaking the black-box model.

Keywords : software reuse, component-based programming, program specialization techniques, partial evaluation, slicing, generating extensions, program generators.

À Carla, Santy et Igui

À tous mes pères et mères adoptifs

Avant de commencer les remerciements, que nous avons l'habitude de rencontrer dans les premières pages d'un manuscrit, dédiés aux personnes qui nous ont soutenu tout au long de l'aventure, je voudrais remercier la personne la plus importante sans laquelle je n'aurais pu achever cette thèse, ma femme : Carla. Je pense que toutes les personnes qui ont été confrontées à un défi similaire pourront comprendre l'importance d'avoir le soutien inconditionnel de la personne qui parcourre le chemin de la vie à notre coté. Il y a des moments où je m'interrogeais sur la nécessité de poursuivre un but pour lequel j'avais plus de doutes que de certitudes. C'est dans ces moments, où peu importait le soutien des autres, j'avais besoin de son soutien... et je l'ai eu. C'est pour cette raison que je considère que toi, Carla, tu s autant auteur de cette thèse que moi. Aussi, avec vos gestes, Santiago et Ignacio, vous m'avez donné tout ce dont une personne a besoin pour comprendre le sens de la vie. Merci.

Je remercie aussi ma famille : mon père Cristo, mon grand père " el abuelo Luis " et ma tante Maria Celia, " la tia ".

Finis mes remerciements personnels, dont j'ai voulu vous faire part, je voudrais maintenant remercier les gens que ont partagé avec moi cette expérience.

Je remercie Monsieur Mourad Oussalah qui m'a fait l'honneur d'être le président de mon jury. Je remercie également Madame Isabelle Borne et Monsieur Charles Consel d'avoir bien voulu accepter la charge de rapporteurs de cette thèse. Je remercie Monsieur Thomas Jensen d'avoir accepter le rôle d'examinateur. Je remercie aussi Monsieur Mufutau Gbadamosi pour avoir accepté mon invitation à participer du jury de cette thèse.

Je me dois de remercier également Pierre Cointe pour avoir eu confiance en moi en me permettant de réaliser cette thèse au sein du département d'Informatique de l'Ecole des Mines de Nantes.

Je tiens à remercier Monsieur Jacques Noyé, mon encadrant scientifique, pour m'avoir guidé et aidé à finir cette thèse. Je dois le remercier aussi pour avoir eu confiance en moi.

Je voudrais aussi remercier tous mes collègues de bureau : Yann-Gaël, Andrés, Marc et Pierre-Charles pour leur compagnie pendant ces nombreuses années.

J'ai quelques mots pour mon grand ami, le chilien Andrés. Bien que nous n'ayons jamais trouvé la réponse à la question " qui nous a obligé à faire une thèse ? ", je tiens à te remercier pour ces moments que vous, avec Sinagi, avez partagés avec nous. Vous faites désormais partie de notre famille.

Je voudrais remercier aussi tous les thésards avec lesquels j'ai partagé mes idées, Luc, Simon, Eric, Hervé, Sebastian Richard, ...

Merci Rémi pour ta sincérité.

Je remercie aussi les membres du département informatique ainsi que à tout le personnelle de l'Ecole des Mines de Nantes qui ont, d'une façon ou d'une autre, facilité mon séjour en France.

Je garde une place pour deux personnes très importantes pour ce qu'elles ont fait pour nous, Annya Romanczuk et Christine Violeau. Annya, j'ai beaucoup apprécié ton amitié et ta confiance. Christine, je tiens à te remercier pour tout ce que tu as fait pour moi et Carla. Tu es aussi devenu partie de notre famille.

Je remercie aussi Monsieur Gustavo Rossi qui a eu confiance en moi et m'a permis de réaliser le master EMOOSE, le point de départ de ma thèse.

Je me dois de remercier Jean-Gabriel Chévé simplement pour avoir eu confiance en moi. Je remercie également Monsieur Jérôme Gratien pour m'avoir permis de faire partie de l'entreprise ProxiAD.

Finalement, je tiens aussi à remercier tous ceux qui m'ont accompagné dans la dernière ligne droite de ma thèse, mes collègues du projet OASIS de la MAIF : Aurélie M., Alexandre B., Gaël B., Pierre-Henri D., Pierre B., Damien B., Alexandre L., Anis Z., Mikaël B., Simon L., Antoine R. Yann C., Stéphane R. et Thierry C..

Aurélie, pour être devenu ma correctrice personnelle et ainsi me permettre d'améliorer mon français, tu as gagné le droit de corriger ces remerciements. Merci.

Table des matières

1 Introduction **13**
1.1 Problématique . . . 13
1.2 Contribution . . . 14
1.3 Structure de la thèse . . . 16

I État de l'art **17**

2 Techniques de spécialisation de programmes **19**
2.1 Évaluation partielle . . . 20
2.1.1 Évaluation partielle en ligne . . . 20
2.1.2 Évaluation partielle hors ligne . . . 22
2.1.3 Approche génératrice : Spécialisation en deux étapes . . . 29
2.1.4 Évaluation partielle, interprétation et compilation . . . 32
2.1.5 Les évaluateurs partiels et leurs applications . . . 34
2.2 Découpage . . . 36
2.2.1 Découpage statique . . . 37
2.2.2 Découpage dynamique . . . 38
2.2.3 Analyse pour le découpage . . . 39
2.2.4 Applications . . . 39
2.2.5 Outils . . . 40
2.3 Intégration l'évaluation partielle et le découpage . . . 40
2.4 Bilan . . . 41

3 Développement d'applications à base de composants **43**
3.1 Des objets aux composants . . . 44
3.2 Notion de composant . . . 45
3.2.1 Réutilisation : boîte noire vs boîte blanche . . . 45
3.2.2 Interfaces . . . 46
3.3 Caractérisation d'un modèle de composants . . . 46
3.3.1 Composants . . . 46
3.3.2 Composition . . . 47
3.3.3 Processus de Développement . . . 48
3.4 Modèles industriels . . . 49

3.4.1 Enterprise Java Beans (EJB) . . . 49
3.4.2 Component Object Model (COM) . . . 53
3.4.3 CORBA Component Model (CCM) . . . 58
3.4.4 Autres modèles industriels . . . 63
3.5 Modèles académiques . . . 63
3.5.1 ComponentJ . . . 64
3.5.2 Java Layers . . . 66
3.5.3 Fractal . . . 70
3.5.4 ArchJava . . . 73
3.5.5 Autres modèles académiques . . . 76
3.6 Composants vs Modules . . . 76
3.7 Bilan . . . 79

4 Spécialisation modulaire 83
4.1 Évaluation partielle sensible aux modules . . . 83
4.1.1 Analyse symbolique de temps de liaison . . . 84
4.1.2 Extension génératrices polymorphiques . . . 86
4.1.3 Spécialisation de modules . . . 87
4.2 Scénarios de spécialisation . . . 88
4.2.1 Définition de scénarios . . . 88
4.2.2 Sous-spécialisation et sur-spécialisation . . . 89
4.2.3 Compilation et application des modules . . . 90
4.3 Analyse globale et séparée de programmes . . . 91
4.4 Bilan . . . 92

II Contribution 93

5 Introduction à la spécialisation dans les langages à objets 95
5.1 Problématique . . . 96
5.1.1 Annotation structurelle . . . 96
5.1.2 La spécialisation et l'héritage . . . 99
5.1.3 Sensibilité au polymorphisme . . . 106
5.2 Notre approche . . . 108
5.2.1 L'analyse . . . 108
5.2.2 La spécialisation . . . 111
5.3 Le langage : EFJ . . . 112
5.3.1 Syntaxe . . . 112
5.3.2 Un exemple : programme *Point2DStepping* . . . 115
5.3.3 Constructions annotables . . . 116
5.3.4 Règles de types déclarés . . . 116
5.4 Bilan . . . 118

6 Analyse et spécialisation **121**
6.1 Analyse de types concrets . . . 121
6.1.1 Règles de bonne annotation de type concret . . . 123
6.1.2 Analyse par contraintes . . . 125
6.2 Analyse de temps de liaison . . . 134
6.2.1 Règles de bonne annotation de temps de liaison . . . 136
6.2.2 Analyse par contraintes . . . 138
6.3 Analyse de temps d'évaluation . . . 145
6.3.1 Règles de bonne annotation de temps d'évaluation . . . 147
6.3.2 Analyse par contraintes . . . 151
6.4 Spécialisation . . . 159
6.4.1 Les extensions génératrices . . . 159
6.4.2 Générateur d'extensions génératrices pour EFJ . . . 166
6.5 Bilan . . . 168

7 Modèle minimal de composants **169**
7.1 Caractéristiques du modèle . . . 170
7.1.1 Composants . . . 170
7.1.2 Composition . . . 172
7.1.3 Processus de développement . . . 174
7.2 Instanciation de composants . . . 176
7.2.1 Instances partagées de composants . . . 176
7.2.2 Instances de composants avec état . . . 177
7.3 Description et implémentation de composants . . . 180
7.3.1 MoSCo-*CDL* . . . 180
7.3.2 Conformité de l'implémentation par rapport à la définition des composants . . . 181
7.4 Bilan . . . 187

8 Spécialisation de composants **189**
8.1 Scénarios de spécialisation . . . 189
8.1.1 Description des scénarios . . . 189
8.1.2 Définition des scénarios . . . 193
8.2 Production de composants spécialisables . . . 195
8.2.1 Des composants aux générateurs de composants . . . 196
8.2.2 Négociation entre les générateurs de composants . . . 198
8.2.3 Spécialisation de composants . . . 201
8.3 Bilan . . . 203

9 MoSCo*suite* **205**
9.1 Architecture . . . 205
9.1.1 *ComProM* (*Component Producer Module*) . . . 205
9.1.2 *ComCoM* (*Component Consumer Module*) . . . 207
9.1.3 Dépôt de composants . . . 207

9.2 Implémentation 208
9.2.1 Conception 208
9.2.2 Points d'extension 210
9.3 Bilan 212

III Conclusions **215**

10 Conclusions **217**
10.1 Analyse et spécialisation dans les langages à objets 217
10.2 Modèle minimal de composants *MoSCo* 218
10.3 Spécialisation de composants 218

11 Perspectives **221**
11.1 Analyse et spécialisation dans les langages à objets 221
11.1.1 Preuve formelle de la correction de la phase d'analyse 221
11.1.2 Langage cible de la spécialisation 221
11.1.3 Résolution intégrée de contraintes 223
11.1.4 Sensibilité des analyses 223
11.2 Modèle minimal de composants *MoSCo* 223
11.2.1 MoSCo-*CDL* 223
11.2.2 Adaptateurs de composants 224
11.3 Spécialisation de composants 225
11.3.1 Quantification de la spécialisation des services 225
11.3.2 Fusion de composants 225
11.3.3 Applications réalistes 225

Table des figures

2.1 Structure d'un évaluateur partiel en ligne . . . 21
2.2 Evaluation partielle en ligne de la méthode `add` . . . 21
2.3 Structure d'un évaluateur partiel hors ligne . . . 23
2.4 Sensibilité au flot de contrôle . . . 25
2.5 Sensibilité au contexte d'appel . . . 26
2.6 Spécialisation avec une analyse sensible au contexte d'appel . . . 27
2.7 Insensibilité au retour . . . 27
2.8 Sensibilité au retour . . . 28
2.9 Sensibilité à l'utilisation . . . 28
2.10 Approche indirecte : auto-application. . . . 31
2.11 Approche directe : écriture à la main. . . . 31
2.12 Découpage statique . . . 37
2.13 Découpage dynamique . . . 38

3.1 Processus de développement à base de composants. . . . 49
3.2 EJB : utilisation de composants EJB dans une architecture à trois niveaux. 50
3.3 EJB : exemple "Bonjour, le monde". . . . 51
3.4 EJB : Contrats de l'architecture. . . . 52
3.5 EJB : processus de développement et ses rôles. . . . 53
3.6 COM : utilisation de composants COM dans une architecture 3-niveaux. . . . 54
3.7 COM : interface du point de vue binaire. . . . 55
3.8 COM : un composant fournissant plusieurs interfaces. . . . 55
3.9 COM : Description d'une interface `IUnknown`. . . . 56
3.10 COM : Composition des composants. . . . 57
3.11 CCM : modèle de Conteneur. . . . 59
3.12 CCM : modélisation d'un distributeur de boissons. . . . 60
3.13 CCM : Définition IDL3 du composant `Distributeur`. . . . 61
3.14 CCM : définition CIDL de la maison du composant `Distributeur`. . . . 61
3.15 CCM : Relations entre la maison, le composant et l'exécuteur. . . . 62
3.16 ComponentJ : un composant et ses interfaces. . . . 64
3.17 ComponentJ : Composant composé. . . . 65
3.18 ComponentJ : Les composant comme des entités de premier ordre. . . . 66
3.19 ComponentJ : Instanciation de composants. . . . 67

3.20 Java Layer : Définition d'une couche. 68
3.21 Java Layer : Composition de couches. 68
3.22 Java Layer : Contraintes de Composition. 69
3.23 Java Layer : Processus de développement. 70
3.24 Fractal : Définition du composant `Server`. 71
3.25 Fractal : Langage Fractal ADL. 72
3.26 Fractal : Composant `ClientServer`. 72
3.27 Fractal : Description du composant `ClientServer`. 73
3.28 ArchJava : Les composants et la composition. 74
3.29 ArchJava : Patron de connexion. 75
3.30 L'application `Clock` en ArchJava. 78
3.31 L'application `Clock` dans le MIL INTERCOL. 79
3.32 Implémentation en Pascal de l'application `Clock`. 80
3.33 Composants vs Modules : Architecture résultante. 81

4.1 Analyse symbolique de temps de liaison. 85
4.2 Extension génératrice de la fonction `Power`. 86
4.3 Spécialisation de modules. 87
4.4 Modules de spécialisation : Spécialisation de la fonction `dot`. 89
4.5 Scénarios de spécialisation : Processus de spécialisation. 90

5.1 Annotation structurelle sur la classe `Point2D`. 97
5.2 Instanciation de la classe `Point2D`. 98
5.3 Analyse de la classe `Point2D` par rapport à l'instance `point1`. 99
5.4 Analyse de la classe `Point2D` par rapport à l'instance `point2`. 100
5.5 Spécialisation depuis le site invoqué. 102
5.6 Héritage et redéfinition de méthodes. 104
5.7 Critère d'insertion de la méthode spécialisée. 105
5.8 Héritage vs. évaluation partielle 106
5.9 Sensibilité au polymorphisme. 107
5.10 Approche pour la spécialisation 108
5.11 REQS : Syntaxe des équations. 111
5.12 EFJ : Syntaxe des programmes. 113
5.13 EFJ : Définitions auxiliaires. 114
5.14 Exemple EFJ : Programme *Point2DStepping* 116
5.15 Exemple EFJ : Implémentation du programme *Point2DStepping*. 117
5.16 EFJ : Relation de sous-typage. 118
5.17 Type déclaré : règles sur les programmes EFJ. 119

6.1 Type concret : règles sur les programmes EFJ. 124
6.2 Variables de contraintes : annotation sur la méthode `Point2D2.dec`. 126
6.3 Type concret : syntaxe des contraintes. 126
6.4 Type concret : sémantique des contraintes. 126
6.5 Type concret : générateur de contraintes. 127

6.6 Type concret : règles de résolution des contraintes. 128
6.7 Treillis d'ensemble de parties des types du programme *Point2DStepping*. . 129
6.8 Type concret : règles de conversion vers REQS. 129
6.9 REQS : opérateur conditionnel `IF`. 130
6.10 Type concret : conversion des inégalités en équations REQS. 131
6.11 Type concret : conversion d'inégalités en égalités. 131
6.12 Réplication de membres hérités : membres de la classe `Point2DX` 132
6.13 Réplication de membres hérités : perspective de l'analyse. 133
6.14 Type concret : annotation de l'expression initiale E_{main}. 135
6.15 Temps de liaison : définitions auxiliaires. 135
6.16 Temps de liaison : constructions de type objet. 136
6.17 Temps de liaison : règles sur les programmes EFJ. 137
6.18 Temps de liaison : syntaxe des contraintes. 139
6.19 Temps de liaison : sémantique des contraintes. 140
6.20 Temps de liaison : Générateur de contraintes. 140
6.21 Temps de liaison : règles de résolution des contraintes. 141
6.22 Temps de liaison : règles de conversion vers REQS. 142
6.23 Temps de liaison : conversion des inégalités en égalités. 143
6.24 Temps de liaison : conversion d'inégalités en égalités. 144
6.25 Type concret : annotations des classes `Point2DX` et `Point2D2`. 145
6.26 Temps de liaison : annotations des classes `Point2DX` et `Point2D2`. 148
6.27 Temps d'évaluation : annotations de constructions de type primitif. 149
6.28 Temps d'évaluation : annotations de constructions de type objet. 149
6.29 Temps d'évaluation : règles sur les constructions de type primitif. 149
6.30 Temps d'évaluation : règles sur les programmes EFJ. 150
6.31 Temps d'évaluation : syntaxe des contraintes. 152
6.32 Temps d'évaluation : sémantique des contraintes. 152
6.33 Temps d'évaluation : générateur des contraintes. 153
6.34 Temps d'évaluation : conversion d'inégalités en égalités. 155
6.35 Temps de liaison : annotation de l'expression initiale e_{main}. 156
6.36 Temps d'évaluation : annotations des classes `Point2DX` et `Point2D2`. 159
6.37 Générateur d'extensions génératrices pour les opérations binaires. 164
6.38 Générateur d'extensions génératrices pour le langage EFJ. 167
6.39 Générateur et constructeur pour l'acces aux champs : $cogen_{Field}$. 168
6.40 Analyse et spécialisation à l'aide d'extensions génératrices. 168

7.1 *MoSCo* : Définition des composants *Adder* et *Multiplier*. 171
7.2 *MoSCo* : Définition du composant *ComputationUnit*. 172
7.3 *MoSCo* : Implémentation du composant *ComputationUnit*. 173
7.4 *MoSCo* : Descripteur de développement du composant *Adder* 174
7.5 *MoSCo* : Descripteur de développement du composant *ComputationUnit* . . 175
7.6 *MoSCo* : Descripteur de déploiement du composant *ComputationUnit* . . . 176
7.7 $CMname$: Composant `ComputationUnit` avec deux instances du composant `Adder` 177

7.8 *MoSCo* : Définition des composants *Counter* et *DCounter*. 178
7.9 *MoSCo* : Description du composant *Clock*. 179
7.10 *MoSCo*-CDL : Grammaire 181
7.11 MoSCo-*CDL* : Définitions auxiliaires. 182
7.12 MoSCo-*CDL* : Règles de conformité. 183
7.13 MoSCo-*CDL* : Règles de conformité (suite). 185
7.14 *MoSCo* : Transformation des interfaces. 186
7.15 MoSCo-*CDL* : Fonction auxiliaire *interfaces*. 186

8.1 Description de spécialisation `MultiplierS`. 191
8.2 Description de spécialisation `MultiplierS` avec d'hypothèses. 192
8.3 Découpage du service `divide_o`. 193
8.4 *MoSCo*-CDL : scénarios de spécialisation. 194
8.5 MoSCo-*CDL* : Règles de conformité sur les scénarios de spécialisation. . . . 194
8.6 Interfaces de ports de générateur `GenAdderI` et `GenMultiplierI`. 198
8.7 `GenMultiplier` : Générateur du composant `Multiplier` (version simplifiée). 199
8.8 Négociation entre les générateurs de composants `GenAdder` et `GenMultiplier`. 200
8.9 Composant `Power`. 201
8.10 Négociation des scénarios pour la spécialisation de composants. 202
8.11 Spécilisation des composants `Adder`, `Multiplier` et `Power`. 202

9.1 Architecture de MoSCo*suite*. 206
9.2 Module *ComProM*. 206
9.3 Module *ComCoM*. 207
9.4 Module *CoRe*. 208
9.5 Infrastructure de MoSCo*suite*. 209
9.6 Analyseur de type concret dans MoSCo*suite*. 211
9.7 Générateur de contraintes `CGMethodDeclaration`. 212
9.8 Adaptateur du solveur REQS dans MoSCo*suite*. 213

11.1 EFJ : Définition initiale des méthodes. 222

Liste des tableaux

6.1 Type concret : génération des contraintes sur l'expression initiale E_{main}. . . 134
6.2 Temps de liaison : génération des contraintes à partir des méthodes `dec` des classes `Point2DX` et `Point2D2`. 146
6.3 Temps de liaison : conversion vers REQS de la contrainte conditionnelle *(d)* et *(n)* du tableau 6.2 147
6.4 Temps d'évaluation : génération des contraintes à partir de l'expression initiale (E_{main}). 157
6.5 Temps d'évaluation : génération des contraintes à partir de la classe `Point2DX`.158

Chapitre 1

Introduction

1.1 Problématique

Depuis quelques années le but essentiel des nouvelles techniques de développement de logiciel a été la génération de logiciel réutilisable à grande échelle (*in the large*). La programmation par objets a permis, au départ, le développement de logiciels complexes en utilisant, par exemple, les schémas de conception (*design patterns*) [GHJV94] pour améliorer les conditions d'extensibilité du logiciel. L'approche à objets, démontrant une bonne couverture des besoins dans le cadre de la programmation à petite échelle (*in the small*), a démontré parallèlement des limitations importantes. En premier lieu, la réutilisation basée sur l'héritage met en cause l'évolution du logiciel car l'implémentation à réutiliser doit être largement accessible. On l'appelle réutilisation de type *boîte blanche*. Un problème bien connu lié à la réutilisation par héritage est la dépendance forte envers l'implémentation initiale, mentionné par Mikhajlov et al. en [MS98] comme le *problème de la classe de base fragile*. En second lieu, les objets se sont montrés insuffisants pour résoudre, de façon naturelle pour les développeurs, des problèmes associés à l'accroissement en complexité des environnements des applications. C'est le cas de la solution fournie par les *objets distribués* [OHE96] aux besoins demandés par le réseau des réseaux. Il a donc été nécessaire d'envisager des techniques permettant la réutilisation de façon modulaire de pieces génériques de logiciel encapsulant des propriétés fonctionnelles. Dans ce contexte, les objets ont évolué en développant des notions nouvelles comme les *composants* et les *aspects*. Cette évolution, appelée la *période post-objets* par Cointe et al. [CND+04], à été motivée principalement par le besoin d'architectures de logiciels facilement maintenables et adaptables. D'une part, la programmation à base de composants facilite l'encapsulation et, par conséquence, la réutilisation de logiciel, qui implique une simplification aussi de la maintenance. Tandis que, d'autre part, les aspects contribuent à l'extensibilité des architectures des logiciels plus particulièrement en ce qui concerne les propriétés non fonctionnelles comme la persistance, la concurrence, la tolérance aux fautes,... Ici, nous nous concentrons sur la programmation à base de composants, et plus particulièrement sur l'adaptation de ces composants à leur contexte d'utilisation.

La programmation à base de composants [HC01, Szy02] cherche à développer des logi-

ciels de qualité de manière rapide en assemblant des composants préfabriqués. Par ce fait, la programmation par composant a été appliquée comme une forme de livraison de pièces de logiciels génériques et exécutables pouvant donc être réutilisées dans différents environnements et applications. Durant le processus de développement par composants nous distinguons clairement deux rôles, d'une part le *producteur de composants* qui produit des composants réutilisables et d'autre part le *consommateur de composants* qui développe des applications en réutilisant les composants. Cette séparation entre le producteur et le consommateur s'appuie sur la notion de composant vu comme une boîte noire (*black box*) fournie par le producteur dont l'*implémentation* reste hors de portée du consommateur. Toutefois, le producteur met à disposition du consommateur une *interface* qui n'est plus qu'une abstraction des fonctionnalités du composant. Dans la plupart des modèles de composants l'interface exprime les *conditions requises* pour l'exécution du composant ainsi que les *fonctionnalités fournies* par le composant. On peut classifier cette information en *services requis* et *services fournis* par la boîte noire mentionnée auparavant. Cette interface a pour but de guider le processus de *composition* (assemblage) de composants en rejetant, par exemple, les compositions incorrectes. Du point de vue syntaxique, l'interface d'un composant est censée conduire la composition de composants d'une part en exploitant l'information structurelle disponible pour la possibilité d'*interconnection* des composants, et d'autre part en faisant intervenir l'information sur le comportement du composant pour vérifier la compatibilité des *interactions* entre les composants composés [CH01, Szy02]. En revanche, en plus de ce rôle syntaxique, l'interface est assujettie aussi à un rôle sémantique où elle est transformée, au moment de la composition (au moins de façon conceptuelle), en une enveloppe (*wrapper*). Cette enveloppe peut être modifiée pour rendre possible la réutilisation de l'implémentation encapsulée dans un contexte particulier. Contrairement à l'implémentation, l'interface étant adaptable par le consommateur du composant, peut être vue comme une boîte blanche. Cette approche, partagée par plusieurs infrastructures de composants telles que les *Entreprise JavaBeans* (EJB) [DeM03] et COM+ [Ses00], produit des applications à base de composants dont seule l'interface des composant est adaptée, alors que leur implémentation reste inchangée. Les possibilités d'adaptation des composants dans les modèles de composant existants sont superficielles. En conséquence, les applications résultantes conservent le degré de généralité initial des composants constituants.

Le but de cette thèse est de contribuer à la définition d'un modèle de composants qui permette une forme plus profonde d'adaptation où l'implémentation des composants est aussi modifiée selon le contexte d'application.

1.2 Contribution

Pour résoudre les problèmes évoqués ci-dessus, nous nous sommes attaqués à la définition d'un modèle de composants *profondément* adaptables. En tant que code source, l'implémentation des composants peut être adaptée à un contexte visé en utilisant des techniques de spécialisation de programmes. En reprenant l'idée initialement proposée par Schultz [Sch99], nous nous sommes servis aussi des techniques de spécialisation de pro-

grammes, à savoir l'*évaluation partielle* [JGS93] et le *découpage* (*slicing*) [Wei84], pour la mise en œuvre de la spécialisation de composants. Nous nous sommes concentrés sur la combinaison de techniques de spécialisation qui, en général, considèrent les applications de manière monolithique et ouverte (boîte blanche) avec la programmation à base de composants qui, par définition, considère les applications comme un composition de composants dont l'accessibilité et l'adaptabilité reste limitée (boîte noire). Concrètement, l'idée a été de conserver la notion de réutilisation de boîtes noires intrinsèque aux composants en introduisant dans la définition du composant les éléments nécessaires pour rendre son implémentation *spécialisable* par les consommateurs. Dans la contribution proposée nous identifions essentiellement les point suivants :

Spécialisation dans des langages à objets Nous définissons un spécialiseur pour l'évaluation partielle et le découpage de programmes Java. La phase d'analyse des programmes est effectuée en combinant différents types d'analyses, à savoir une analyse de type concrets, une analyse de temps de liaison pour finir par une analyse de temps d'évaluation. En fait, le spécialiseur est appliqué sur des programmes écrits dans un sous-ensemble de Java, appelé EFJ, une extension de la version de FJ [IPW01] proposée par Schultz [Sch00]. Cela permet principalement de nous concentrer sur les caractéristiques intrinsèques aux langages à objets.

Langage de Composants Nous proposons un langage minimal de composants, appelé *MoSCo* (*Model of Specializable Components*), qui permet le développement et l'assemblage de composants de manière simple. L'interface des composants est décrite à l'aide de ce langage tandis que l'implémentation est spécifiée par un programme EFJ (voir ci-dessus). Le langage inclut aussi des règles qui permettent de vérifier la cohérence entre l'interface et l'implémentation.

Spécialisation de Composants Le modèle *MoSCo* est étendu afin d'ajouter aux composants la description des opportunités de spécialisation associées à leurs services. Dans ce cas, le producteur détermine comment l'implémentation du composant peut être spécialisée si certaines conditions sont satisfaites. En effet, le producteur étend la description initiale de l'interface avec les *scénarios de spécialisation* [LMCE02] possibles par rapport à l'implémentation associée au composant. Cette information est utilisée pour la génération d'un générateur du composant qui produit finalement une version spécialisée du composant par rapport à un contexte d'utilisation donné par le consommateur.

MoSCo*suite* À l'aide de l'outil de développement Eclipse [Ecl], nous avons développé un ensemble de *plugins*, appelé MoSCo*suite*, qui fournissent les fonctionnalités nécessaires aux producteurs et consommateurs de composants pour mettre en œuvre notre approche. D'une part, le plugin *ComProM* (*Component Producer Module*) permet aux producteurs la définition et la génération des composants spécialisables. D'autre part, le plugin *ComCoM* (*Component Consumer Module*) est utilisé par le consommateur pour la recherche, l'assemblage des composants (spécialisables) ainsi que pour la génération d'une application, à base de composants, spécialisée. Les fonctionnalités des deux modules mentionnés sont accessibles à travers une interface de l'utilisateur (*GUI*) qui permet, entre autres fonctionnalités, d'accéder au dépôt

de composants (*CoRe*).

1.3 Structure de la thèse

Le contenu de la présente thèse est structuré en trois parties : l'état de l'art, la contribution et les conclusions.

Tout d'abord nous présentons l'état de l'art par rapport au contexte de notre approche. Dans le chapitre 2, nous introduisons les concepts clés des techniques de spécialisation considérées : l'évaluation partielle et le découpage. Le chapitre 3 résume les principales caractéristiques des modèles de composants ainsi que les implémentations existantes tant dans le contexte industriel qu'académique. Finalement, les approches qui ont essayé de conjuguer les deux mondes, en proposant une spécialisation modulaire, sont présentées dans le chapitre 4.

Dans la seconde partie, nous présentons notre proposition pour l'adaptation d'applications à base de composants en utilisant les techniques de spécialisation mentionnées. Dans les chapitres 5 et 6, nous nous intéressons au langage d'implémentation des composants. Dans le chapitre 5, nous introduisons la problématique de la spécialisation dans les langages à objets et présentons un langage à objets minimal et une approche de spécialisation nous permettant de couvrir cette problématique. La spécialisation est décrite en détail dans le chapitre 6.

Dans le chapitre 7, nous proposons un modèle minimal de composants pour la description et l'implémentation de composants spécialisables. Le chapitre 8 conclut la description de notre approche en décrivant le processus de développement de composants spécialisables dans le cadre du modèle minimal précédent. Ensuite, dans le chapitre 9, nous décrivons notre environnement de développement pour la construction et la réutilisation de composants spécialisables.

Finalement, le chapitre 10 résume notre travail et le chapitre 11 en décrit les perspectives.

Première partie

État de l'art

Chapitre 2

Techniques de spécialisation de programmes

Sommaire

2.1 Évaluation partielle **20**
2.1.1 Évaluation partielle en ligne 20
2.1.2 Évaluation partielle hors ligne 22
2.1.3 Approche génératrice : Spécialisation en deux étapes 29
2.1.4 Évaluation partielle, interprétation et compilation 32
2.1.5 Les évaluateurs partiels et leurs applications 34
2.2 Découpage **36**
2.2.1 Découpage statique 37
2.2.2 Découpage dynamique 38
2.2.3 Analyse pour le découpage 39
2.2.4 Applications 39
2.2.5 Outils 40
2.3 Intégration l'évaluation partielle et le découpage **40**
2.4 Bilan **41**

Les techniques de spécialisation de programmes permettent l'adaptation automatique de programmes à un contexte d'exécution particulier. Elles produisent un programme spécialisé qui se comporte de manière équivalente dans le contexte d'exécution considéré, mais qu'on espère plus efficace. Plusieurs techniques de spécialisation basées sur différentes méthodes ont été définies, à savoir l'*évaluation partielle* [JGS93], le *découpage*(*slicing*) [Wei84] et la *bifurcation* [Mog89], basées essentiellement sur l'analyse et transformation de code source ainsi que la *super-compilation* [Sør94, Tur86], qui évalue les programmes de manière symbolique pour imiter leur comportement. Dans ce chapitre nous nous concentrons sur l'évaluation partielle et le découpage (*slicing*), qui permettent la spécialisation des programmes en fonction de leurs paramètres d'entrée et de sortie, respectivement. Autrement dit, lors de la spécification du contexte de spécialisation le programme peut être vu comme une *boîte noire* où nous n'identifions que les entrées et

les sorties. De ce fait, notre intérêt pour ces deux techniques deviendra claire dans les chapitres suivants où nous abordons la spécialisation de composants.

2.1 Évaluation partielle

L'évaluation partielle est une technique de transformation automatique de programmes en fonction d'un contexte d'utilisation donné. Le résultat de la transformation est un programme spécialisé ou résiduel dans lequel toutes les expressions basées sur des valeurs connues du contexte, appelées aussi *statiques* et dénotées $V_{\mathcal{S}}$, ont été précalculées lors de la transformation. Le programme résiduel ne contient que les calculs dépendant des valeurs d'entrée inconnues, appelées aussi *dynamiques* et dénotées $V_{\mathcal{D}}$, au moment de la transformation. Du point de vue fonctionnel, l'évaluation partielle peut se comparer avec la *currification* de fonctions où, en fixant la valeur de certains arguments, on obtient une nouvelle fonction ne prenant en compte que les arguments non fixés.

Parfois utilisée comme un synonyme de *spécialisation de programmes*, l'évaluation partielle a comme but principal l'optimisation des programmes en réduisant le temps d'exécution et/ou la mémoire utilisée. L'évaluation partielle a été étudiée dans le contexte des langages fonctionnels [Bon90, Con93], impératifs [And94, BGZ94, CHN+96], logiques [LS91] et à objets [Sch00, SLC03]. Dans le cadre de notre travail nous avons utilisé Java comme le langage de programmation cible de la spécialisation (voir section 5.3). Par contre, dans cette section, pour des raisons de simplicité (et d'homogénéité) nous ne considérons que le sous-ensemble impératif de Java [Cas97] dans l'illustration du fonctionnement de différentes approches d'évaluation partielle.

Quand l'évaluation partielle s'applique lors de l'exécution on utilise un évaluateur partiel *en ligne*, tandis que pour la spécialisation effectuée pendant l'étape de compilation on utilise un évaluateur partiel *hors ligne*. Le choix de l'approche adéquate dépendra du moment où les valeurs connues deviendront disponibles.

Notation On considère un programme comme ayant deux facettes, en premier lieu comme une fonction qui prend une entrée pour générer la sortie correspondante et en second lieu comme des données qui peuvent être prises comme une entrée ou le résultat généré par un autre programme. Pour distinguer les deux facettes nous dénotons par l'expression $[\![p]\!]$ l'exécution d'un programme dont le code source du programme est représenté par p, ou $[\![p]\!]_L$ dans le cas où il est nécessaire d'expliciter le langage d'implémentation du programme p. Le résultat de l'exécution du programme p par rapport aux données d'entrée d dans un environnement d'exécution de programmes L, appelé aussi machine-L, s'exprime par l'expression $[\![p]\!]_L(d)$.

2.1.1 Évaluation partielle en ligne

Un évaluateur partiel en ligne dont la structure est montrée dans la figure 2.1 génère le programme résiduel en une étape [Ruf93]. Ce type de spécialisation peut être vu comme une interprétation non standard où le programme est évalué en fonction des valeurs connues des

variables et le programme résiduel est généré à partir de la représentation textuelle des variables dont les valeurs restent inconnues lors la spécialisation ainsi que de la représentation textuelle des valeurs des expressions précalculées [JGS93, Mey91, WCRS91]. Ce type de spécialisation est connu aussi comme *evaluation partielle à l'exécution*[CN96, NHCL96]. L'équation 2.1 exprime le processus d'évaluation partielle en ligne [JGS93].

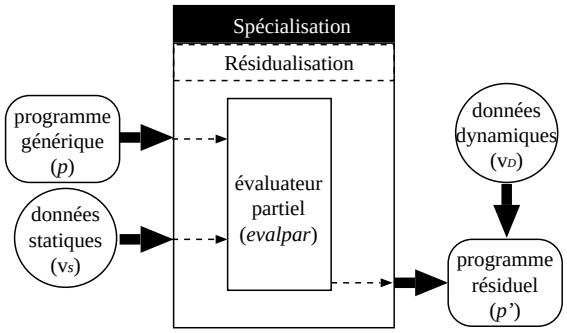


FIG. 2.1 – Structure d'un évaluateur partiel en ligne

$$[\![[\![evalpar]\!](p, v_{\mathcal{S}})]\!](v_{\mathcal{D}}) = [\![p']\!](v_{\mathcal{D}}) = [\![p]\!](v_{\mathcal{S}}, v_{\mathcal{D}}) \tag{2.1}$$

Dans l'équation 2.1, l'évaluateur partiel (*evalpar*) génère une version spécialisée du programme *p*, *p'*, par rapport aux valeurs d'entrée connues $v_{\mathcal{S}}$. Parallèlement, le résultat de l'exécution du programme *p'* appliqué aux valeurs inconnues $v_{\mathcal{D}}$ est équivalent à celui obtenu à partir de l'exécution du programme original et générique *p*.

(a) Méthode `add`.

```
(1) int add (int x, int y){
(2)     int out, aux;
(3)     out = 0;
(4)     aux = x;
(5)     out = out + aux;
(6)     out = out + y;
(7)     return out;
(8) }
```

(b) Spécialisation en ligne.

```
Contexte de spécialisation: x = 3
(1) int add_x3 (, int y){
(2)     int out;
(3)
(4)
(5)
(6)     out = 3 + y;
(7)  return out;
(8) }
```

FIG. 2.2 – Evaluation partielle en ligne de la méthode `add`

Pour illustrer cette approche considérons la méthode `add` de la figure 2.2(a). En supposant que le paramètre d'entrée `x` soit statique et égal à 3 l'évaluation partielle en ligne produit le programme résiduel de la figure 2.2(b).

L'approche en ligne quant à elle produit une spécialisation avec une précision plus importante que dans l'approche hors ligne puisqu'elle manipule des valeurs concrètes, lesquelles sont toujours disponibles au temps d'exécution. Par exemple, dans une expression

conditionnelle, si les valeurs de variables qui interviennent dans le test sont connues alors l'évaluateur sera capable d'évaluer les instructions concernées dans la branche choisie, après l'évaluation du test. Dans le cas où le test ne peut pas être évalué alors toute l'expression conditionnelle est résidualisée. Cependant, cette approche devient souvent coûteuse parce qu'elle ne permet pas de partager le travail d'analyse des temps de liaison. Par conséquent, lors de la spécialisation de la même partie de programme et pour la même configuration de valeurs connues mais avec des valeurs concrètes différentes la même analyse se répète.

2.1.2 Évaluation partielle hors ligne

À la différence de l'approche en ligne, un évaluateur hors ligne réalise la spécialisation d'un programme en deux étapes : une étape d'*analyse* suivie d'une étape de *spécialisation* proprement dite, appelée aussi *résidualisation*. Dans ce cas, la spécialisation à lieu avant même que l'exécution ne débute, par conséquent les valeurs des paramètres connus seront disponibles également avant l'exécution. En sachant que la fonction sera appelée de nombreuses fois avec le même ensemble de paramètres statiques mais avec des valeurs concrètes différentes il est plus efficace d'effectuer une analyse qui soit partagée pour les différentes valeurs au moment de la spécialisation. Cette analyse, appelée *analyse de temps de liaison*, consiste en une interprétation abstraite du code à spécialiser sur deux valeurs abstraites associées aux paramètres : statique ($\mathcal{S}$) pour représenter les paramètres connus et dynamique ($\mathcal{D}$) pour représenter les paramètres inconnus. Lors de l'analyse, la description abstraite des paramètres est propagée à travers le programme en annotant les constructions avec un temps de liaison indiquant quels fragments devront être évalués (calculs qui dépendent uniquement de paramètres statiques) où quels fragments devront être reconstruits (calculs qui dépendent pour une part de quelques paramètres dynamiques et effet de bords).

Finalement, dans l'étape de spécialisation l'évaluateur partiel effectue la spécialisation du programme à partir des valeurs concrètes des paramètres statiques. La définition 2.2 décrit ces deux étapes.

$$[\![evalpar]\!]([\![analyser]\!](p, tdl_{\mathcal{S}\&\mathcal{D}}), v_{\mathcal{S}}) = [\![evalpar]\!](p_{annote}, v_{\mathcal{S}}) = p' \tag{2.2}$$

$$[\![p']\!](v_{\mathcal{D}}) = [\![p]\!](v_{\mathcal{S}}, v_{\mathcal{D}}) \tag{2.3}$$

L'évaluateur partiel hors ligne génère un programme spécialisé *p'* en prenant le programme original annoté (p_{annote}) produit par l'analyseur à partir de l'information de temps de liaison initial ($tdl_{\mathcal{S}\&\mathcal{D}}$) plus les valeurs statiques ($v_{\mathcal{S}}$). Finalement, et exactement de la même manière que pour l'évaluation partielle en ligne, l'exécution de ce programme spécialisé en lui donnant les valeurs d'entrée dynamiques ($v_{\mathcal{D}}$) produit un résultat équivalent à celui du programme original p avec les valeurs des paramètres correspondants. La figure 2.3 expose la structure d'un évaluateur partiel hors ligne.

L'un des intérêts de l'approche hors ligne est que le résultat de la phase d'analyse reste valide pour les mêmes descriptions de temps de liaison des paramètres d'entrée ($tdl_{\mathcal{S}\&\mathcal{D}}$). Pour cette raison la spécialisation devient plus rapide car le spécialiseur n'a qui à interprété l'information de transformation calculée à l'avance pour l'analyse, c'est-à-dire à évaluer les constructions statiques et résidualiser les constructions dynamiques.

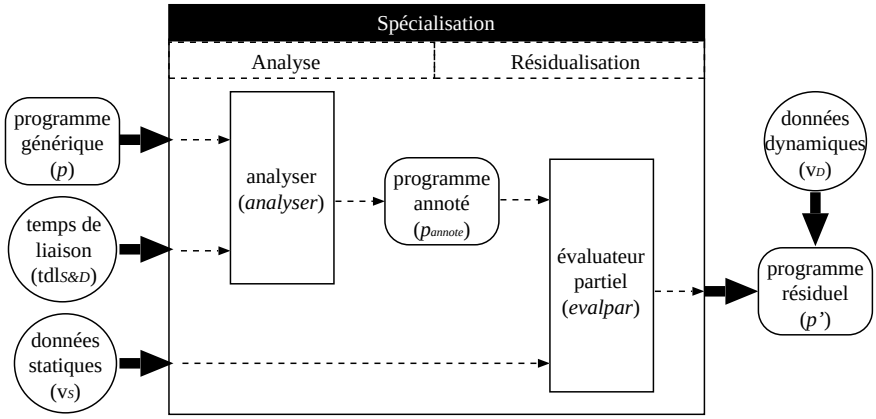


FIG. 2.3 – Structure d'un évaluateur partiel hors ligne

2.1.2.1 Précision de l'analyse de temps de liaison

La propagation de l'information des temps de liaison des paramètres d'entrée d'un programme divise le programme en deux sous-ensembles : les constructions annotées comme statiques d'un côté, et les constructions annotées comme dynamiques de l'autre. De la méthode employée pour implémenter cette propagation dépendra la précision de l'analyse. Une construction de programme est statique si elle ne dépend que de constructions statiques, sinon elle est dynamique. On dit donc que l'analyse est *sûre* (*safe*) si toutes les constructions qui ont été annotées comme statiques ne sont associées à aucune construction annotée comme dynamique. En effet, puisque la valeur $\mathcal{D}$ est une *approximation* sûre de la valeur $\mathcal{S}$ il est toujours cohérent d'annoter une construction comme dynamique même quand la construction ne dépend pas de constructions dynamiques. Indépendamment de l'approche choisie, le calcul du temps de liaison se base sur une relation d'ordre $\leq$ sur les valeurs abstraites $\{\mathcal{S}, \mathcal{D}\}$ définit comme suit :

$$\forall\ (t,t')\ \in\ \{\mathcal{S},\mathcal{D}\}_2,\ t \leq t' sii\ t = \mathcal{S}\ ou\ t = t'\ \forall\ t,t' \in \{\mathcal{S},\mathcal{D}\}\ tel\ que\ \mathcal{S} < \mathcal{D} \quad (2.4)$$

L'opérateur $\leq$ peut être interprété comme "*moins dynamique que*" ou "*plus statique que*". Pendant l'analyse, une construction peut être décrite à travers plusieurs valeurs du temps de liaison. Afin de trouver une description commune et sûre pour cette construction on calcule le *plus petit majorant* (*least upper bound*) défini comme suit :

$$t \sqcup t' = t"|\exists\ t"|\ t" est\ le\ plus\ petit\ élément\ de\ \{\mathcal{S},\mathcal{D}\} tel\ que\ t \leq t" \wedge t' \leq t" \quad (2.5)$$

L'expression $t \sqcup t'$ correspond à la plus petite (moins dynamique) valeur abstraite qui est au moins aussi dynamique que t et t'. Par exemple, $\mathcal{S} \sqcup \mathcal{D} = \mathcal{D}$, $\mathcal{S} \sqcup \mathcal{S} = \mathcal{S}$,... L'opérateur $\sqcup$ est comparable à l'union entre ensembles : l'union $A \cup B$ est le plus petit ensemble qui est plus grand ou égal à A et B.

Quand l'analyse tend à généraliser la valeur abstraite des constructions et par conséquence à rendre la division du programme moins précise on parle d'une analyse *monovariante*. Au contraire, une analyse *polyvariante* essaie de fournir une division plus précise en annotant, par exemple, avec des valeurs de temps de liaison différentes une même construction qui intervient à plusieurs points du programme [JGS93]. Nous parlons aussi d'une analyse *insensible* ou *sensible* selon le cas. Pour différencier les constructions statiques des constructions dynamiques, elles seront annotées avec le temps de liaison correspondant. Si le temps de liaison affecté à la construction `const` est statique alors la construction sera annotée $\underset{\mathcal{S}}{\texttt{const}}$, autrement elle sera annotée $\underset{\mathcal{D}}{\texttt{const}}$. Dans le cas des expressions composées (e.x. affectations de variables, opérations binaires, ...), les annotations sont associées au symbole correspondant, par exemple, l'expression d'affectation d'une variable est annotée sur le symbole égal (`=`). Durant la spécialisation une construction statique `const` est évaluée en une valeur v tandis que une construction annotée comme dynamique est évaluée en une construction résiduelle `const'`. Cependant, une construction statique devra être résidualisée si elle se trouve dans un contexte dynamique. C'est le cas des opérations binaires où interviennent des constantes, qui par définition sont des constructions statiques, et des variables dynamiques. Afin de résidualiser une construction statique dans le programme spécialisé le spécialiseur doit *"faire monter"* (*lift*) le temps de liaison de la construction sur le treillis mentionné auparavant. Pour cela, l'analyseur annote la construction `const` comme $\underset{lift}{\texttt{const}}$ pour indiquer au spécialiseur qu'il s'agit d'une construction statique qui ne doit pas être évaluée au moment de la spécialisation [Mog88].

Sensibilité au flot de contrôle. Dans une analyse sensible au flot nous considérons les espaces de mémoire susceptibles d'être affectés qui sont gérés par un programme donné. Car une analyse associe une valeur de temps de liaison pour chaque affectation alors une variable pourrait avoir plusieurs annotations de temps de liaison dans des points de programmes différents. Dans l'exemple de la figure 2.4 la méthode `add` est analysée en considérant le paramètres `x` comme étant statique et le paramètre `y` comme dynamique.

Dans le cas d'une analyse insensible au flot, toutes les affectations de la variable `out` vont avoir le même temps de liaison en reprenant le principe qu'une construction de programme sera dynamique s'il existe au moins une construction dont elle dépend qui était annotée comme dynamique. Ceci est illustré par l'exemple 2.4(a) où on observe que même quand la variable `out` reste statique jusqu'à la ligne (5) elle devient dynamique à cause de l'affectation de la ligne (6) où le paramètre `y` intervient. Par contre, si on applique une analyse sensible au flot, comme montré par l'exemple 2.4(b), la même variable `out` est annotée comme statique jusqu'à la ligne (5) et comme dynamique après et, par conséquent, la spécialisation trouvera plus de constructions à évaluer.

Sensibilité au contexte d'appel. L'utilisation de variables globales dans la définition d'une fonction implique que l'analyse ne dépendra pas seulement de ses paramètres mais aussi de variables globales, définissant une *contexte d'appel*. Une analyse sensible au contexte, dite aussi polyvariante, produit une version spécialisée de la fonction par rapport

(a) Analyse insensible au flot de contrôle.

```
// Temps de liaison: [x_S, y_D]
(1) int add (int x_S, int y_D){
(2)     int out_D, aux_D;
(3)     out_D = 0_S;
(4)     aux_S = x_S;
(5)     out_D = out_D +_D aux_S;
(6)     out_D = out_D +_D y_D;
(7)     return_D out_D;
(8) }
```

(b) Analyse sensible au flot de contrôle.

```
// Temps de liaison: [x_S, y_D]
(1) int add (int x_S, int y_D){
(2)     int out_D, aux_D;
(3)     out_S = 0_S;
(4)     aux_S = x_S;
(5)     out_S = out_S +_S aux_S;
(6)     out_D = out_S +_D y_D;
(7)     return_D out_D;
(8) }
```

FIG. 2.4 – Sensibilité au flot de contrôle

aux différents contextes d'appel.

La figure 2.5(a) montre l'analyse de la méthode `addAccum` par rapport au contexte d'appel décrit dans la figure 2.5(b). En plus du temps de liaison des paramètres, le contexte inclut aussi l'information des variables globales, dans ce cas le champ `accum` défini dans la classe `Adder`. Pour le premier appel, la méthode est analysée avec une description initiale de temps de liaison spécifiant que le premier paramètre `x` et le champ `accum` sont statiques, alors que le paramètre `y` est dynamique. Noter qu'à cause de l'affectation qui a lieu la ligne `(11)` le champ `accum` devient dynamique après le premier appel. Ceci justifie de distinguer les deux appels même si les temps de liaison des paramètres restent les mêmes. Comme les deux appels sont analysés séparément, lors de la spécialisation deux versions spécialisées sont générées. Dans les figures 2.5(c) et (d), nous observons le résultat de l'analyse sensible au contexte pour les deux appels. Le fait que la variable globale (champ) `accum` soit dynamique rend l'analyse radicalement différente, particulièrement à partir de la ligne `(22)` de la figure 2.5(d) où la variable locale `out` est affectée. Par contre, une analyse insensible au contexte d'appel n'aurait fourni que l'analyse de la figure 2.5(d) pour les deux appels (lignes `(18)` et `(19)` de la figure 2.5(b)). Finalement, nous illustrons dans la figure 2.6 la spécialisation d'un programme dont l'analyse est sensible au contexte en reprenant le résultat des analyses des figures figure 2.5 (c) et (d) et en affectant des valeurs concrètes aux paramètres de la méthode.

Sensibilité au retour. Maintenant nous expliquons comment une fonction pourrait retourner une valeur statique même si elle doit être residualisée à cause des effets de bord dynamiques inclus dans son corps. Pour illustrer cette approche nous prenons la méthode `addCounter` qui contrairement à la méthode `addAccum` modifie la variable globale `counter` pour compter les appels de la méthode en plus de l'addition des deux valeurs entières. La figure 2.7(a) montre le résultat de l'analyse de la méthode en considérant les paramètres `x`

(a) Implémentation de la classe `Adder`.

```
(1) class Adder {
(2)   int accum;
(3)  Adder(int initValue){
(4)     this.accum = initValue;
(5)  }
(6)  int addAccum (int x, int y){
(7)     int out;
(8)     out = this.accum;
(9)     out = out + x;
(10)    out = out + y;
(11)    this.accum = out;
(12)    return out;
(13)  }
(14) ...
(15) }
```

(b) Contexte d'appel.

```
// Temps de liaison: [a, b]
                      S  D

(16) int add1, add2, a, b;
(17) Adder adder = new Adder(0);
(18) add1 = adder.add(a,b);
(19) add2 = adder.add(a,b);
     ...
```

(c) Analyse pour le premier appel.

```
// Contexte : [accum]
                S

(20) int addAccum (int x, int y){
                       S      D
(21)     int out;
           D
(22)     out = this.accum;
          S S      S
(23)     out = out + x;
          S    S    S
(24)     out = out + y;
          D   D  S  D  D
(25)     this.accum = out;
            D       D  D
(26)     return out;
            D      D
(27) }
```

(d) Analyse pour deuxième appel.

```
// Contexte : [accum]
                D

(20) int addAccum (int x, int y){
                       S      D
(21)     int out;
           D
(22)     out = this.accum;
          D D      D
(23)     out = out + x;
          D D  D   D  D
(24)     out = out + y;
          D D  D   D  D
(25)     this.accum = out;
            D       D  D
(26)     return out;
            D      D
(27)   }
```

FIG. 2.5 – Sensibilité au contexte d'appel

et `y` comme statiques et `counter` comme dynamique. La figure 2.7(b) montre un exemple de spécialisation.

Une analyse sensible au retour calcule un temps de liaison pour les valeurs de retour et un temps de liaison pour les effets de bord. En prenant la méthode `addAccount` de l'exemple 2.7(a), nous illustrons dans l'exemple 2.7(c) et (d) une analyse insensible au retour en considérant les paramètres `x` et `y` comme statiques et le champ `counter` comme statique par rapport au contexte d'appel (i.e. valeurs concrètes) de l'exemple 2.7(b). L'approche insensible au retour impose la résidualisation de l'instruction de retour (`return`) même quand l'expression associée est annotée comme statique. Ceci empêche, par exemple, d'effectuer l'expansion en ligne (*inlining*) de l'appel de la méthode. La figure 2.7(c) montre le résultat de la spécialisation de la méthode `addCounter` par rapport aux valeurs `2` et `3` affectées aux paramètres `x` et `y` respectivement, où l'instruction de retour est résidualisée (ligne `(20)` de la figure 2.7(b)), malgré l'expression statique retournée (la constante `5`).

(a) Spécialisation pour le premier appel.

```
// Valeurs: [x = 2, accum = 0]
(20) int addAccum_x2 (int y){
(21)     int out;
(22)     [          ]
(23)     [          ]
(24)     out = 2 + y;
(25)     this.accum = out;
(26)     return out;
(27) }
```

(b) Spécialisation pour le deuxième appel.

```
// Valeurs: [x = 2]
(20) int addAccum (int y){
(21)     int out;
(22)     out = this.accum;
(23)     out = out + 2;
(24)     out = out + y;
(25)     this.accum = out;
(26)     return out;
(27) }
```

FIG. 2.6 – Spécialisation avec une analyse sensible au contexte d'appel

(a) Analyse insensible au retour.

```
// Contexte : [x, y, counter]
                S  S  D
(1) int addCounter(int x, int y){
                       S      S
(2)     this.counter = this.counter + 1;
           D       D       D     D S
(3)     return x + y;
           D    S S S
(4) }
```

(b) Spécialisation de la méthode.

```
// Valeurs: [x = 2, y = 3]
(1) int addCounter_x2_y3(){
(2)     this.counter = this.counter + 1;
(3)    return 5;
(4) }
```

(c) Analyse du contexte d'appel.

```
// Contexte : [a, b, counter]
                S  S  D
...
(5)   int add = this.addCounter(a,b) + 5;
           D   D       D          S S D S
 ...
```

(d) Spécialisation de l'appel.

```
// Valeurs: [a = 2, b = 3]
...
(5) int add = this.addCounter_x2_y3() + 5;
 ...
```

FIG. 2.7 – Insensibilité au retour

Par contre, dans le cadre d'une analyse sensible au retour l'instruction de retour est annotée comme statique (voir ligne `(3)`, figure 2.8(a)). Ceci impacte sur le résultat de l'analyse dans le site d'appel (voir figure 2.8(b)). Par conséquent, la méthode résiduelle résultante de la spécialisation ne contiendra que les effets de bord et la valeur statique retournée remplacera l'appel de la méthode originale dans le site d'appel (voir figures 2.8(c) et (d)).

Sensibilité à l'utilisation. La sensibilité à l'utilisation a été introduite dans [HN00] dans le contexte de programmes systèmes pour le traitement des pointeurs et des structures. Elle permet de traiter des utilisations d'une même variable indifféremment, ce qui n'empêche pas de résidualiser et d'évaluer l'utilisation en fonction du temps de liaison associé en chaque cas. Dans le cadre des objets, il est possible d'avoir une référence sur un objet utilisé pour atteindre un champ statique et un champ dynamique. Dans ce cas,

(a) Analyse insensible au retour.

```
// Contexte : [x, y, counter]
(1) int addCounter(int x, int y){
(2)     this.counter = this.counter + 1;
(3)     return x + y;
(4) }
```

(b) Spécialisation de la méthode.

```
// Valeurs: [x = 2, y = 3]
(1) void addCounter_x2_y3(){
(2)      this.counter = this.counter + 1;
(3)      [          ]
(4) }
```

(c) Analyse du contexte d'appel.

```
// Contexte : [a, b, counter]
...
(5)  int add = this.addAccount(a, b) + 5;
      ...
```

(d) Spécialisation de l'appel.

```
// Valeurs: [a = 2, b = 3]
     ...
(5)   addCounter_x2_y3();
(6)   int add = 10;
     ...
```

FIG. 2.8 – Sensibilité au retour

une analyse insensible à l'utilisation rendra la référence dynamique et par conséquent la déclaration du champ sera residualisées dans tous les cas. Par contre, si on considère une analyse sensible à l'utilisation, la référence est annotée comme dynamique et statique ce qui conduit à une résidualisation de la déclaration et de tous les accès au champ dynamique tandis que les accès au champ statique peuvent être évalués. En reprenant l'exemple de la classe `Adder` de la figure 2.5(a), nous illustrons respectivement l'analyse insensible et sensible à l'utilisation.

(a) Insensible à l'utilisation.

```
// Contexte : [initAccum = 15, initCounter]
(1) int averageAdder1;
(2) ...
(3) Adder adder1 = new Adder(initAccum,
(4)                 initCounter);
(5) averageAdder1 = adder1.accum /
(6)                       adder1.counter;
    ...
```

(b) Sensible à l'utilisation.

```
// Contexte : [initAccum = 15, initCounter]
(1) int averageAdder1;
(2) ...
(3) Adder adder1 = new Adder(initAccum,
(4)                 initCounter);
(5) averageAdder1 = 15 /
(6)                       adder1.counter;
    ...
```

FIG. 2.9 – Sensibilité à l'utilisation

Dans la figure 2.9(a), le champ `accum`, lorsqu'il est utilisé dans un contexte dynamique, devrait être résidualisé en utilisant sa représentation textuelle lors de la spécialisation. Par contre, si on considère une approche sensible à l'utilisation l'accès au champ `adderAccum`

peut être évalué malgré le contexte dynamique (voir figure 2.9(b)).

2.1.2.2 Analyse de temps d'évaluation

L'analyse du temps de liaison calcule les temps de liaison des variables en propageant en avant l'information de la définition vers les utilisations des variables. L'analyse du temps d'évaluation, au contraire, propage l'information en arrière, des utilisations vers les définitions [HN00]. Les utilisations statiques de variables statiques qui n'ont pas une représentation textuelle (e.x. références) sont traitées comme dynamiques ainsi que leurs définitions. Une analyse de temps d'évaluation évite la perte d'opportunités de spécialisation en annotant les définitions de telles variables à la fois comme dynamiques et statiques [HNC97]. Pour cette raison une analyse sensible à l'utilisation aura besoin de l'information du temps d'évaluation.

2.1.2.3 Analyse d'action

Connue aussi comme analyse de transformation [HN00], l'analyse d'action peut être vue comme la compilation du temps de liaison [CD90]. Elle permet d'annoter les constructions de programme avec l'action à effectuer lors de la spécialisation. À la base, une construction est annotée *Rd* (*Réduire*) si elle peut être éliminée du programme résiduel. Elle est annotée *Rc* (*Reconstruire*) si elle doit apparaître dans le programme résiduel. Par exemple, si le test d'une instruction conditionnelle est statique, l'instruction sera annoté *Rd* et, finalemente, seule la branche sélectionnée sera résidualisée. En conséquence, l'instruction conditionnelle n'apparaît pas dans le programme résiduel. Pour améliorer l'efficacité au moment de la spécialisation, les annotations de base, *Rd* et *Rc*, peuvent être affinées. Si un sous arbre représentant un fragment du programme est totalement dynamique alors il est annoté *Id* (*Identique*). Par contre, un fragment totalement statique, qui peut donc être évalué en sa totalité, est annoté *Ev* (*Evaluer*). Dans les deux cas, le spécialiseur n'a pas besoin d'analyser récursivement toute la structure de l'arbre.

2.1.3 Approche génératrice : Spécialisation en deux étapes

De la même façon qu'on ajoute l'étape d'analyse dans le processus de spécialisation d'évaluation partielle en ligne pour rendre la spécialisation proprement dite plus efficace, il est possible, pour la même raison, de diviser l'étape de spécialisation trouvée dans l'évaluation partielle hors ligne. Un évaluateur hors ligne réutilise le résultat de l'analyse (i.e. la division du programme en constructions statiques et constructions dynamiques) pour générer des versions spécialisées du programme par rapport à différentes valeurs statiques d'entrée. De l'équation 2.2 on déduit qu'un évaluateur partiel hors ligne interprète à chaque fois la division résultant de l'analyse par rapport à des valeurs statiques données. En conséquence, si on considère l'évaluateur partiel comme un programme ordinaire, une version plus efficace de ce programme peut être obtenue en le spécialisant par rapport à ses paramètres statiques d'entrée, qui dans ce cas est la division (annotation) donnée du programme. Le programme résultant, un spécialiseur spécialisé, est appelé *extension génératrice* (*generating extension*) [JGS93].

Il existe deux façon d'obtenir l'extension génératrice, l'*approche indirecte* et l'*approche directe* [BW94] que nous décrivons ci-après.

2.1.3.1 Approche indirecte : auto-application

Cette méthode correspond à l'explication développée auparavant (i.e. *spécialiseur spécialisé*). Cette approche est étudiée en détail dans [And94, Bon90, JGS93, Lau91]. Les équations 2.6 et 2.7 représentent la définition de la spécialisation de l'évaluateur partiel ci-avant, tandis que l'équation 2.8 représente l'application de l'extension génératrice par rapport aux valeurs statiques (i.e. $v_{\mathcal{S}}$) qui génère finalement le programme spécialisé p'. La figure 2.10 illustre le processus de l'approche indirecte.

$$[\![evalpar]\!](spec, [\![analyser]\!](p, tdl_{\mathcal{S}\&\mathcal{D}})) = [\![evalpar]\!](spec, p_{annote}) \tag{2.6}$$

$$[\![evalpar]\!](spec, p_{annote}) = extGenInd_p \tag{2.7}$$

$$[\![[\![extGenInd_p]\!](v_{\mathcal{S}})]\!](v_{\mathcal{D}}) = [\![p']\!](v_{\mathcal{D}}) = [\![p]\!](v_{\mathcal{S}}, v_{\mathcal{D}}) \tag{2.8}$$

Cette approche souffre d'un problème d'encapsulation des valeurs et du programme. Les valeurs manipulées par le spécialiseur sont représentées par un environnent qui lie noms et valeurs. Les programmes quand ils sont manipulés à travers leur arbre de syntaxe abstrait sont considerés comme des valeurs. Lors de la spécialisation, le spécialiseur prend le programme comme donnée et, par conséquent, il est encapsulé dans l'environnement correspondant. Cela implique que les données du programme sont doublement encapsulées. Une solution proposée dans [And94, Lau91] consiste à introduire un nouveau type, programme, qui permet de traiter un programme au même niveau que les types primitifs au lieu d'être considéré comme un type composé. Néanmoins, les solutions mentionnées peuvent introduire des codifications inutiles ce qui réduit l'efficacité lors de la spécialisation. Pour résoudre ce problème [And94] propose l'implémentation d'une analyse de *codification* qui permet de détecter si le code résiduel associé à l'extension génératrice conserve encore des calculs exprimés par encapsulations répétitives pouvant ralentir le code résultant. D'autre part, le fait que le spécialiseur doit être capable de s'auto-appliquer implique une forte dépendance sur le langage.

2.1.3.2 Approche directe : écriture à la main

Un générateur d'extensions génératrices peut être construit, contrairement à l'approche indirecte, de manière *ad hoc*. L'équation 2.9 montre que le générateur, appelé *cogen* [JGS93], prend le programme annoté comme entrée pour la génération de l'extension génératrice correspondante. Concrètement, l'extension génératrice extériorise une partie de l'arbre de syntaxe abstrait du programme vers le programme résiduel. L'équation 2.10 devient identique à l'équation 2.8 en utilisant une extension génératrice qui n'est pas forcément la même.

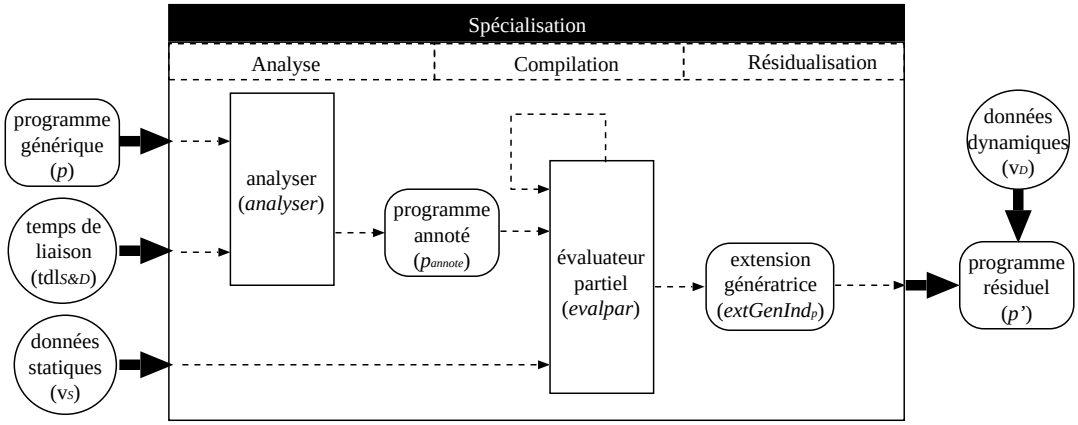


FIG. 2.10 – Approche indirecte : auto-application.

$$[\![cogen]\!]([\![analyser]\!](p, tdl_{\mathcal{S}\&\mathcal{D}})) = [\![cogen]\!](p_{annote}) = extGenDir_p \tag{2.9}$$

$$[\![[\![extGenDir_p]\!](v_{\mathcal{S}})]\!](v_{\mathcal{D}}) = [\![p']\!](v_{\mathcal{D}}) = [\![p]\!](v_{\mathcal{S}}, v_{\mathcal{D}}) \tag{2.10}$$

L'extension génératrice générée par *cogen* doit résidualiser les parties dynamiques du programme et exécuter les parties annotées comme statique. Pour cela *cogen* utilise des *fonctions génératrices* qui permettent d'évaluer les parties statiques et de les *reconstruire* lors de l'exécution de l'extension génératrice. La figure 2.11 montre les étapes correspondant à l'approche directe.

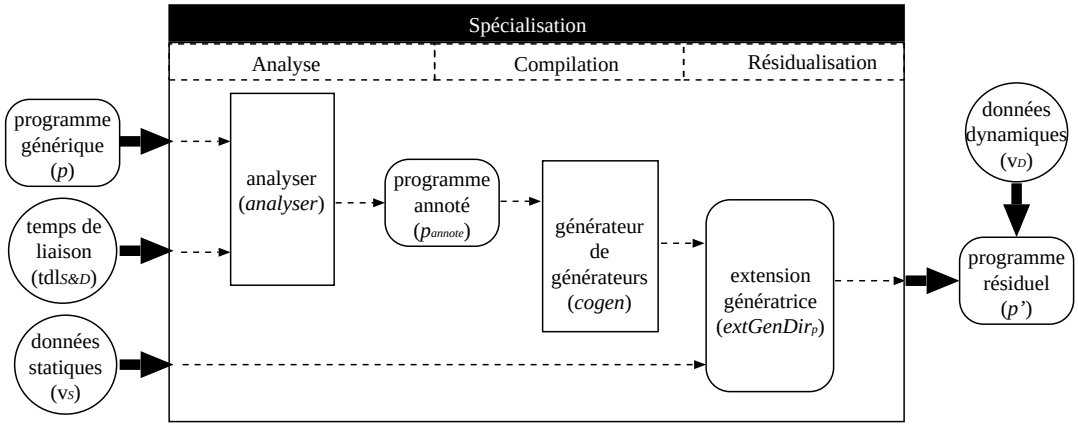


FIG. 2.11 – Approche directe : écriture à la main.

Puisque le *cogen* ne manipule que des arbres de syntaxe abstraits, la codification réalisée dans l'approche indirecte n'est plus nécessaire. Pour cette même raison, un générateur généré par l'approche directe devrait être plus efficace que celui généré par auto-application

car il n'a pas besoin d'accéder à un environnement lors de la construction de l'extension génératrice. Finalement, dans l'approche directe, la restriction existant sur le langage d'implémentation du spécialiseur et celui du programme à spécialiser n'existe plus.

Dans [BW94], Birkedal et al. montrent que l'implémentation à la main de *cogen* n'implique pas un effort supérieur à l'implémentation d'un évaluateur partiel auto-applicable.

2.1.4 Évaluation partielle, interprétation et compilation

Nous rappelons ici brièvement le rapport entre évaluation partielle, interprétation et compilation en nous inspirant de [MS97].

2.1.4.1 Compilation par évaluation partielle

Un interprète *Sint* implémenté dans un langage L, qui permet d'interpréter des programmes écrits dans un langage S, satisfait, pour tous les programmes p et un ensemble de valeurs d'entrée v, l'équation suivante :

$$[\![p]\!]_S(v) = [\![Sint]\!]_L(p, v) \tag{2.11}$$

L'équation 2.11 exprime simplement l'équivalence existant entre le résultat produit par l'exécution du programme p sur une S-machine en prenant v comme entrée et le résultat produit par l'exécution de l'interprète *Sint* sur une L-machine en prenant p, le programme « interprété », et v comme entrée.

Dans le cas le programme p est exécuté d'une façon répétitive il devient logique de spécialiser l'interprète par rapport à l'entrée statique p et l'entrée dynamique v. En réutilisant l'équation 2.1 de l'évaluation partielle nous pouvons exprimer ce processus comme suit :

$$[\![[\![evalpar]\!]_L(Sint, p)]\!](v) = [\![Sint_p]\!]_L(v) = [\![Sint]\!]_L(p, v) \tag{2.12}$$

Alors, en appliquant la définition de l'interprète (équation 2.11) nous obtenons :

$$[\![Sint_p]\!]_L(v) = [\![p]\!]_S(v) \tag{2.13}$$

Même si le programme résiduel $Sint_p$ est équivalent au programme original p, leur implémentation est différente. En particulier, alors que p est écrit dans le langage S, *Sint* est écrit dans le langage L. Concrètement, nous montrons ici la modélisation du processus de compilation (transformation) d'un programme p écrit en S, le langage interprété par l'interprète, vers le langage L, le langage d'implémentation de l'interprète.

2.1.4.2 Génération d'un compilateur par auto-application

En outre, un compilateur *STcomp* implémenté dans un langage L et défini pour traduire des programmes écrits dans un langage source S vers des programmes écrits dans un langage cible T, est tel que :

$$[\![STcomp]\!]_L(p) = p' \Rightarrow [\![p']\!]_T(v) = [\![p]\!]_S(v) \tag{2.14}$$

C'est-à-dire qu'un programme p transformé en un programme p' et exécuté dans une T-machine avec les valeurs d'entrée v génère un résultat équivalent à l'interprétation du programme p avec les mêmes données d'entrée v dans une machine-S.

Dans la section 2.1.4.1, nous avons présenté la compilation d'un programmes comme étant équivalente au résultat de l'évaluation partielle d'un interprète de ce programme. La compilation de différents programmes implique la spécialisation d'un même interprète par rapport aux différents programmes cibles de la compilation. Puisque l'entrée restant statique dans le processus de compilation est l'interprète *Sint*, il est donc possible de spécialiser l'évaluateur partiel par rapport à cette entrée. Le résultat de l'auto-application, exprimé par la définition 2.15, est un compilateur des programmes interprétables par *Sint*.

De la même façon que la compilation par l'évaluation partielle d'un interprète permet la *réutilisation* du programme résiduel (programme compilé),

D'après la définition 2.1 :

$$[\![evalpar]\!](evalpar, Sint) = evalpar_{Sint} \Rightarrow [\![evalpar_{Sint}]\!](p) = [\![evalpar]\!](Sint, p) \quad (2.15)$$

En appliquant une nouvelle fois la définition d'*evalpar* à la partie droite de l'implication 2.16 :

$$[\![evalpar_{Sint}]\!](p) = Sint_p \Rightarrow [\![Sint_p]\!](v) = [\![p]\!](v) \quad (2.16)$$

En reprenant la définition d'un compilateur (2.14), nous observons que $evalpar_{Sint}$ satisfait les conditions requises pour être considéré comme un compilateur de programmes S vers des programmes T, où T est le langage de sortie de l'évaluateur partiel. Si le langage d'entrée et de sortie de l'évaluateur partiel est le même, alors le langage d'implémentation du compilateur est implémenté et le langage cible de la compilation sont égaux au langage L, le langage d'implémentation de l'interprète *Sint*.

Le compilateur $evalpar_{Sint}$ est une extension génératrice de l'interprète *Sint* selon la définition 2.8. De manière générale, il est possible d'obtenir une extension génératrice p_{gen} d'un programme p en évaluant partiellement un évaluateur partiel *evalpar* par rapport au programme p, tel que $p_{gen} = evalpar_p$.

2.1.4.3 Génération d'un générateur de compilateurs

À partir de l'auto-application d'un évaluateur partiel expliquée dans la section précédente, il est possible d'envisager l'*auto-auto-application* exprimée par la définition suivante :

$$[\![evalpar]\!](evalpar, evalpar) = evalpar_{evalpar} \Rightarrow [\![evalpar_{evalpar}]\!](p) = [\![evalpar]\!][evalpar, p] \quad (2.17)$$

Car $[\![evalpar]\!](evalpar, p) = evalpar_p$ étant une extension génératrice de p, $evalpar_{evalpar}$ devient un générateur d'extensions génératrices. Le programme $evalpar_{evalpar}$ est une extension génératrice de l'évaluateur partiel : $evalpar_{gen} = evalpar_{evalpar}$. Donc, dans le cas

où le programme p est un interprète, l'extension génératrice p_{gen} est un compilateur et e_{gen} est un générateur de compilateurs capable de produire un compilateur à partir d'un interprète.

Toutes les définitions mentionnées ci-dessus, qui expriment l'application d'un évaluateur partiel par rapport à un interprète, de manière directe ou par auto-application, sont connues comme les projections de Futamura et définies comme suit :

1ère projection : Compilation

$$[\![evalpar]\!](\overline{interprete}^{Prg}, \overline{source}^{Val}) = \overline{programme}^{Prg} \tag{2.18}$$

2ème projection : Génération de compilateurs

$$[\![evalpar]\!](\overline{evalpar}^{Prg}, \overline{\overline{interprete}^{Prg}}^{Val}) = \overline{compilateur}^{Prg} \tag{2.19}$$

$$[\![compilateur]\!](\overline{source}^{Val}) = \overline{compilateur}^{Prg} \tag{2.20}$$

3ème projection : Génération de générateurs de compilateurs

$$[\![evalpar]\!](\overline{evalpar}^{Prg}, \overline{\overline{evalpar}^{Prg}}^{Val}) = \overline{genCompilateur}^{Prg} \tag{2.21}$$

$$[\![genCompilateur]\!](\overline{\overline{interprete}^{Prg}}^{Val}) = \overline{compilateur}^{Prg} \tag{2.22}$$

Les deux premières projections ont été développées par Futamura [Fut71] tandis que la dernière a été abordée par Beckman [BHOS76] et Turchin [Tur77] de manière indépendante. Les définitions représentant les projections de Futamura ont été proposées par Birkedal et al. [BW94] dans le contexte de langages statiquement typés, où ^{Val} type des données et ^{Prg} type des programmes. Cette notation permet d'identifier la manière dont sont traités les éléments des définitions, soient les entrées et sorties à chaque étape des projections. Cela permet également d'identifier les problèmes existant dans l'approche directe car le fait qu'un argument de type ^{Prg} soit également du type ^{Val} introduit le problème du double codage mentionné dans la section 2.1.3.2.

2.1.5 Les évaluateurs partiels et leurs applications

Cette section présente un survol des implémentations des évaluateurs partiels définies dans le contexte de langages impératifs, fonctionnels ou logiques. Nous nous occuperons dans le chapitre 5 du contexte des langages à objets. Nous détaillons, ensuite, quelques applications de l'évaluation partielle.

2.1.5.1 Évaluateurs partiels

Langages impératifs Parmi les premiers évaluateurs partiels pour langages impératifs nous trouvons le travail d'Ershov [Ers77, Ers78], tandis que Bulyonkov *et al.* présentent [BB88] pour un langage décrivant des organigrammes comme celui décrit par Gomard et Jones dans [GJ89]. Pour la spécialisation de programmes Fortran,

Baier et al. décrivent en [BGZ94] un spécialiseur utilisé pour l'optimisation d'un ensemble de problèmes de calculs numériques. Andersen définit deux implémentations pour la spécialisation de programmes C : un évaluateur partiel auto-applicable pour un sous-ensemble de C [And94, And93], avec procédures, pointeurs et vecteurs, et un évaluateur partiel pour tout ANSI C [And94]. Tempo est un évaluateur partiel hors ligne du langage C qui permet de spécialiser des programmes à la compilation mais aussi à l'exécution [CHL+98, CHN+96].

Langages fonctionnels Le premier évaluateur pour un sous-ensemble de Lisp, appelé `Redfun`, a été défini par Beckman et al. dans [BHOS76]. Beckman explique l'utilisation de l'évaluation partielle aux différentes étapes du développement de programmes. Jones et al. [JSS85] ont conçu un évaluateur auto-applicable pour programmes Lisp (premier ordre) pour lequel l'utilisateur est sensé fournir les annotations de temps de liaison. Une version automatisée du même évaluateur est décrite en [JSS89]. En ce qui concerne la spécialisation de programmes Scheme, Weise et al. [WCRS91] ont travaillé sur la définition d'un évaluateur partiel en ligne qui permet de spécialiser de manière complètement automatique. En outre, Consel définit Schism, un évaluateur partiel auto-applicable pour un sous-ensemble de Scheme qui manipule des structures partiellement statiques et réalise une analyse de temps de liaison polyvariante [Con88, Con90, Con93]. Similix, un évaluateur partiel auto-applicable sur un sous-ensemble de Scheme a été défini par Bondorf en [Bon90, Bon91]. Pour ce qui est de ML, d'une part, la spécialisation de programmes ML a été abordée par Malmkjær et al. en [MHD94], d'autre part le travail de Birkedal et al. mentionné auparavant se base sur la description d'un générateur d'extensions génératrices sur un sous-ensemble de SML.

Langages logiques La majorité des travaux sur la spécialisation de programmes logiques a été développée autour du langage Prolog. Une caractéristique particulière des langages logiques est leur capacité à être exécutés en prenant un ensemble des paramètres d'entrée incomplet. Ce qui à la simple vue pourrait être confondu avec l'évaluation partielle, ils se distinguent principalement par les résultats obtenus car de l'évaluateur en attend plus qu'une simple liste de faits. Le pionnier de l'évaluation partielle (où *déduction partielle* dans le contexte logique) pour Prolog est Komorowski [Kom81, Kom82]. Pour les langages logiques, l'évaluation en ligne présentent plus d'opportunités que l'évaluation hors ligne puisque lors du processus d'unification des variables dynamiques sont souvent instanciées. Toutefois, Gurr [Gur94] et Mogensen et al. [MB93] ont proposé des stratégies hors ligne ayant l'auto-application comme objectif. En outre, Sahlin a défini en [Sah91] un spécialiseur pour des programmes Prolog réalistes, mais dans ce cas il n'est pas auto-applicable. Finalement, nous mentionnons le travail de Jørgensen et al. [JL96] qui présente un générateur d'extensions génératrices pour Prolog.

2.1.5.2 Applications

Comme mentionné auparavant l'application la plus courante de l'évaluation partielle a été, pour un grand nombre de langages, la compilation de programmes ainsi que la

génération de compilateurs. En dehors des applications dans le cadre de langages de programmation, l'évaluation partielle a été appliquée pour résoudre des problèmes d'efficacité dans des domaines spécifiques, à savoir les systèmes d'exploitation, le graphisme, la simulation et l'intelligence artificielle, entre autres.

La généralité et modularité intrinsèque aux systèmes d'exploitation sont la cause pour laquelle ces applications sont de bonne cibles de spécialisation, cela à cause, d'une part, de l'utilisation des bibliothèques génériques et, d'autre part, de leur structuration modulaire. Dans le premier cas, les bibliothèques qui implémentent les fonctionnalités associées au protocole d'appel de procédure distante (RPC pour *Remote Procedure Call*) ont été spécialisées par Muller et al. [MVM97, MMV+98]. Dans le deuxième cas, McNamee a spécialisé les signaux utilisés dans le système d'exploitation Unix pour la communication inter-processus [MWP+01].

Dans le domaine du graphisme par ordinateur, la spécialisation des surfaces cachées a été abordée par [Goa82] avec une application réelle dans l'optimisations des simulateurs de vol. En outre, l'évaluation partielle a aidé à la reconstruction des objets d'une scène virtuelle (*ray-tracing*), qu'il est possible d'optimiser par rapport à ses éléments restant fixes [Mog86, And95].

En ce qui concerne la simulation, quand une partie du modèle reste fixe tandis que l'autre varie, l'évaluation partielle peut être utilisée pour l'optimisation de la simulation par rapport à la partie fixe comme l'a montré [Ber90] dans la simulation du mouvement d'objets quelconques.

Dans le domaine de l'intelligence artificielle, la relation de cette discipline et l'évaluation partielle a été discutée initialement dans [Kah84]. Par exemple, le processus d'apprentissage défini sur des réseaux sémantiques implique généralement de multiples parcours du même espace de recherche. Le processus d'apprentissage peut donc être optimisé par rapport à quelques paramètres qui restent fixes comme la topologie du réseau [Jac90, vHB88]. Andersen :94

2.2 Découpage

Le *découpage* (*slicing*) permet au développeur d'analyser un programme par rapport à un sous-ensemble de ses comportements. Le découpage calcule la portion (*slice*) minimale du programme qui reproduit le résultat choisi et qui représente elle-même le programme original par rapport au domaine de ce résultat [Wei84]. Les éléments constituant la portion identifiée consistent en des blocs du programme qui n'apparaissent pas forcément de manière adjacente.

L'intérêt de la technique de découpage était initialement le débogage et la compréhension de programmes. À la différence des techniques existantes appliquées pour la conception de programmes, comme par exemple l'encapsulation d'information (*information hiding*) [Par72], l'abstraction de données (*data abstraction*) [LZ74] ou des techniques de conception (HIPO [Sta76]), le découpage étant appliqué sur des programmes complètement implémentés est donc une technique qui permet d'améliorer leur maintenance. Le fait d'utiliser le code source du programme fait du découpage une technique

plus précise et susceptible d'automatisation.

Le découpage requièrt tout d'abord l'identification d'une portion déterminée du programme. Cette identification se fait à travers la spécification d'une instruction déterminée et un ensemble (parfois un singleton) de variables. L'ensemble des caractéristiques qui identifie une portion du programme s'appelle *critère de découpage*. Une portion est découpée du programme en éliminant les instructions que n'affectent pas les variables incluses dans le critère de découpage.

Il existe également différentes stratégies de découpage selon que les entrées du programme sont ignorées ou pas. Dans le premier cas il s'agit d'un *découpage statique* tandis que dans le deuxième cas on parle d'un *découpage dynamique*.

2.2.1 Découpage statique

(a) Méthode `divide`.

```
...
(1)  DivResult divide(int x, int y){
(2)    int quotient = 0;
(3)    int remainder = 0;
(4)    boolean error = true;
(5)    if (y!=0) {
(6)      while (x > = y) {
(7)        quotient = quotient + 1;
(8)         x = x - y;
(9)      }
(10)     remainder = x;
(11)   else {
(12)     error = true;
(13)   }
(14)   return new DviResult(error,
(15)                        quotient,
(16)                        remainder);
(17) }
...
```

(b) Découpage par le critère `(remainder,16)`.

```
...
(1)  DivResult divide(int x, int y){
(2)
(3)    int remainder = 0;
(4)
(5)    if (y!=0) {
(6)      while (x > = y) {
(7)
(8)        x = x - y;
(9)      }
(10)     remainder = x;
(11)   else {
(12)
(13)   }
(14)   return new DviResult(  ,
(15)                          ,
(16)                        remainder);
(17) }
...
```

Fig. 2.12 – Découpage statique

La figure 2.12(a) montre la méthode `divide` qui calcule la division entière de deux valeurs entières positives passées à travers des paramètres `x` et `y` respectivement. Le résultat est encapsulé dans un objet de type `DivResult` contenant le résultat de la division proprement dit représenté par la variable `quotient`, le reste présenté par la variable `remainder` et la condition de faisabilité de la division (i.e. division par une valeur nulle) exprimée dans le résultat par la variable `error`. Nous pourons maintenant vouloir obtenir la portion de la méthode `divide` dans laquelle, par exemple, nous conservons les instructions et variables qui affectent le calcul d'une partie du résultat. Si nous ne sommes intéressés que par le

reste de la division, le critère de découpage `(remainder,16)` doit être fourni. Le critère de découpage est, comme mentionné auparavant, formé de la variable (`remainder`) et du point de programme à prendre en compte (ligne `16`). Le résultat du découpage montré dans la figure 2.12(b) inclut les instructions concernant les instructions où la valeur de la variable `remainder` est utilisé. Autrement dit, les instructions pour le calcul du résultat et la condition de faisabilité sont éliminées de la portion résultante.

La technique de découpage expliquée ci-dessous est connue comme une technique de *découpage en arrière* (*backward slicing*) car l'analyse du programme est faite en parcourant *en arrière* le code du programme à partir du critère donné. Il est aussi possible d'identifier la portion souhaitée en gardant les instructions et les variables dépendant du critère spécifié. Une instruction (ou variable) est dit *dépendant* du critère si les valeurs calculées lors de l'exécution de l'instruction dépendent des valeurs intervenant dans le critère donné. Cette variante du découpage, appelée *découpage en avant*, a été mentionnée par Bergetti et Carré [BC85] et expliquée ensuite par Reps et Bricker [RB89].

2.2.2 Découpage dynamique

(a) Méthode `divide`.

```
...
(1)  DivResult divide(int x, int y){
(2)   int quotient = 0;
(3)   int remainder = 0;
(4)   boolean error = true;
(5)   if (y!=0) {
(6)     while (x > = y) {
(7)       quotient = quotient + 1;
(8)        x = x - y;
(9)     }
(10)    remainder = x;
(11)  else {
(12)    error = true;
(13)  }
(14)  return new DviResult(error,
(15)                       quotient,
(16)                       remainder);
(17) }
...
```

(b) Découpage par le critère `([x = 3, y = 2],16`[1]`,remainder)`.

```
...
(1)  DivResult divide(int x, int y){
(2)   [          ]
(3)   int remainder = 0;
(4)   [                  ]
(5)   [      ]
(6)     while (x > = y) {
(7)       [          ]
(8)        x = x - y;
(9)     }
(10)    remainder = x;
(11)  [      ]
(12)      [      ]
(13)  [ ]
(14)  return new DviResult([ ],
(15)                       [ ],
(16)                       remainder);
(17) }
...
```

FIG. 2.13 – Découpage dynamique

Pendant le débogage d'un programme, il arrive parfois d'obtenir durant l'exécution des valeurs qui ne correspondent pas aux valeurs attendues. Dans ce cas, pour savoir pourquoi les valeurs inattendues apparaissent, le programmeur devrait pouvoir identifier l'ensemble des instructions intervenant dans le calcul de ces valeurs. Cette technique, connue comme

une analyse de *flot arrière* (*flow back*) [Bal69], peut être implémentée par découpage où les entrées spécifiques de l'exécution interviennent dans le processus d'indentification de la portion. Cette technique de découpage, appelé *découpage dynamique*, est guidé lui-même par un *critère de découpage dynamique*. Dans ce contexte, le critère doit distinguer, lors de l'exécution, entre des occurrences différentes d'une instruction déterminée.

La figure 2.13 (b) montre le résultat du découpage de la méthode `divide` dont le corps original est montré dans la figure 2.13 (a), par rapport au critère `([x = 3, y = 2],16`[1]`,remainder)`. La portion correspondante est calculée en considérant les valeurs `3` et `2` pour les paramètres, la première occurrence de l'instruction `16` et en fonction de la valeur associée à la variable `remainder`. L'instruction conditionnelle (5) vérifiant la faisabilité de la division a été éliminée parce que la valeur connue de `y` permet d'identifier la branche exécutée. Noter que le même résultat peut être obtenu si la valeur d'entrée affectée au paramètre `x` est omise. Cela montre que la spécification partielle des entrées peut être possible comme c'est le cas des approches hybrides décrites dans [CMN91, DGS92, Kam93].

2.2.3 Analyse pour le découpage

Pour déterminer l'appartenance d'une instruction à une portion déterminée il faut connaître la relation existante entre l'instruction et le critère de découpage. La solution générale se base sur la construction du *graphe de dépendance* (*program dependence graph*) [FOW87, KKP$^+$81] dont les nœuds représentant les variables et les instructions tandis que les arcs représentent leurs relations. Ottensteint et al. définissent le découpage de programmes par un algorithme qui calcule l'accessibilité des nœuds dans le graphe correspondant : les nœuds accessibles à partir des nœuds qui représentent le critère choisi appartiennent à la portion résultante [OO84]. Sur la base du graphe de dépendance, Weiser [Wei84] calcule une portion en définissant des équations de flot de données sur le programme. Elle permettent de connaître la pertinence de chaque nœud du graphe et ainsi de pouvoir obtenir une portion *exécutable* du programme. Pour le découpage de programmes contenant des communications inter-procédurales, manipulation de types composés (vecteurs, pointeurs, ...) ou flot de contrôle arbitraire, différentes solutions ont été proposées : programmes récursifs [HDC88, HRB90], flot de contrôle arbitraire [BH93a, CF94], découpage avec vecteurs et pointeurs [Lyl84, ADS91], entre autres.

2.2.4 Applications

Des nombreuses applications utilisent le découpage comme une technique de base dans différents domaines. C'est le cas de Gallagher et al. [GL91] qui appliquent cette technique dans la maintenance de logiciel. Dans ce contexte, le découpage est utilisé pour la décomposition de logiciel, et donc face à une modification il est possible de délimiter l'impact d'un tel changement. De façon générale, le découpage a été utilisé comme une technique d'analyse de programmes. En particulier, il s'applique dans la compréhension de programmes en permettant, par exemple, l'implémentation des outils pour l'identification de différences entre plusieurs versions du même programme (*differencing*) [HPR89, Hor90].

En outre, le découpage a été largement appliqué dans le débogage d'applica-

tions [WL86, ADS93] car il permet d'identifier les parties (portions) qui ne retournent pas les résultats attendus, ainsi que dans la détection du code mort, c'est-à-dire de code qui n'affecte aucun résultat possible du programme analysé [BC85].

La vérification des résultats attendus d'un programme peut être effectuée en utilisant le découpage dans la modélisation des cas de tests (*test-cases*) [BH93b, Bin98]. Des portions obtenues il est possible de savoir si la définition et l'utilisation d'une variable interviennent dans un cas de test déterminé. Duesterwald et al. [DGS92] appliquent le découpage dans le domaine de la vérification de logiciel où le critère de découpage requièrt aussi que les variables incluses dans les cas de tests fassent partie du résultat, c'est-à-dire que elles aient une influence directe ou indirecte sur le résultat attendu de la portion concernée.

Un autre domaine d'application des techniques de découpage qui a été exploité dernièrement est la conception d'architectures logicielles. C'est le cas de Zhao [Zha98] où le découpage intervient dans la compréhension de l'architecture proprement dite, appelé *découpage architectural*, ainsi que dans la généralisation qui permet finalement la réutilisation de différentes parties formant l'architecture analysée. La généralisation par découpage est présente aussi dans le processus de reingénierie de programmes. Les portions obtenues peuvent être considérées comme des sous-composants en y ajontant une interface pour leur réutilisation ultérieure [NEK94].

2.2.5 Outils

La plupart des outils de découpage disponibles n'offrent pas une couverture complète du langage cible et implémentent généralement des variantes simples du découpage, typiquement du découpage statique.

L'outil le plus largement utilisé a été le *Wisconsin Program Slicer* [Wis]. Il permet le découpage statique en avant et en arrière de programmes C. Même si ce projet est terminé, le code de base a été intégré dans l'implémentation commerciale multi-plateforme *CodeSurfer* [Cod] qui permet aussi le découpage de programmes C++.

Unravel [Unr] a été développé pour le découpage de programmes C. Il utilise le découpage statique en arrière dans la détection de portion de programmes qui s'exécutent dans plusieurs endroits du programme analysé. Cette information est extrêmement importante car le dysfonctionement d'une portion caractérisée de cette manière affecterait plusieurs composants logiques du programme.

Finalement, nous mentionnons le projet Bandera [Ban] un vérificateur de modèles (*model checker*) qui utilise le découpage afin de déterminer les dépendances entre composants dans le domaine des applications à base de composants. Une dérivation de ce projet est le projet Indus [Ind] qui a donné comme résultat un plugin Eclipse [Ecl] appelée Kaveri [Kav] qui permet le découpage en arrière et en avant sur des programmes Java.

2.3 Intégration l'évaluation partielle et le découpage

Le découpage peut être vu comme un complément de l'évaluation partielle car le découpa en arrière propage l'information de spécialisation dés valeurs de retour tandis

que l'évaluation partielle propage le même type d'information dés les paramètres d'entrée du programme [RT96].

L'intégration de l'évaluation partielle et le découpage en arrière s'avère naturelle à condition de que l'analyse soit sensitive à l'utilisation [HN00]. Comme mentionné auparavant (voir section 2.1.2.1), l'intérêt d'une analyse sensible à l'utilisation est de permettre aux différents utilisations d'une variable avoir différents annotations de temps de liaison. Cette annotation peut être calculée en divisant l'analyse de temps de liaison en deux étapes. Lors de la premier étape l'analyse propage le *temps de liaison de l'utilisation* (*use binding times*) qui correspondent, concrètement, aux valeur de temps de liaison (i.e., $\mathcal{S}$, $\mathcal{D}$). Lors de la deuxième étape, par contre, on propage en arrière le temps de liaison de l'utilisation afin de calculer le *temps de liaison de la définition* (*definition binding times*). Dans ce cas, la définition d'un emplacement de mémoire (une variable, un champ, ...) est vu comme la somme de ses utilisations. En conséquence, la domaine associé à ces annotations est l'ensemble de parties du domaine de temps de liaison classique dont l'élément plus grand du treillis est $\{\mathcal{S}, \mathcal{D}\}$ et correspond aux emplacements de mémoire qui sont autant statique que dynamique. L'élément plus petit $\{\}$, par contre, représente à un emplacement qui n'est pas utilisé lors de l'exécution du programme. La définition d'un tel emplacement peut être considéré comme de code mort et, par conséquent, peut être ignoré du programme résultant de la spécialisation.

2.4 Bilan

Dans ce chapitre, nous avons décrit des techniques de spécialisation de programmes utilisées pour rendre plus performants des programmes conçus pour des contextes d'utilisation génériques. Les opportunités de spécialisation sont identifiées en utilisant différentes approches d'analyse de programmes. Ici, nous nous sommes intéressés, en particulier, à l'évaluation partielle et le découpage. Ces deux techniques de spécialisation fonctionnent de manière nettement invasive. Du point de vue de l'accessibilité aux détails des programmes spécialisés, ces techniques considèrent les programmes comme étant des *boîte blanches* car elles analysent et transforment leur code source. Néanmoins, elles permettent la spécialisation des programmes en fonction de leurs paramètres d'entrée et de sortie, respectivement. Autrement dit, lors de la spécification du contexte de spécialisation, le programme peut être vu comme une *boîte noire* où ne sont identifiées que les entrées et les sorties. En conséquence, dans le chapitre 8, nous essayons de combiner les deux visions pour la spécialisation d'applications à base de composant.

Chapitre 3

Développement d'applications à base de composants

Sommaire

3.1 Des objets aux composants **44**
3.2 Notion de composant **45**
3.2.1 Réutilisation : boîte noire vs boîte blanche 45
3.2.2 Interfaces 46
3.3 Caractérisation d'un modèle de composants **46**
3.3.1 Composants 46
3.3.2 Composition 47
3.3.3 Processus de Développement 48
3.4 Modèles industriels **49**
3.4.1 Enterprise Java Beans (EJB) 49
3.4.2 Component Object Model (COM) 53
3.4.3 CORBA Component Model (CCM) 58
3.4.4 Autres modèles industriels 63
3.5 Modèles académiques **63**
3.5.1 ComponentJ 64
3.5.2 Java Layers 66
3.5.3 Fractal 70
3.5.4 ArchJava 73
3.5.5 Autres modèles académiques 76
3.6 Composants vs Modules **76**
3.7 Bilan **79**

Le coût élevé du développement des applications, caractérisant la période connue comme la *crise du logiciel* [McI68], était provoqué, principalement, par la production de logiciel conçu à la demande d'un client final et sans la réutilisation de solutions implémentées auparavant. Plus encore, la technologie de programmation disponible jusque dans les

années soixante n'était pas apte à affronter affronter l'augmentation de la production de logiciels.

La programmation de logiciels à base de composants [HC01, Szy02] est un paradigme dont le principal objectif est de faciliter la construction d'applications à grande échelle. La programmation à base de composants facilite l'encapsulation et, par conséquent, la réutilisation du logiciel, ce qui implique aussi une simplification de la maintenance. Cette approche cherche à développer des logiciels de qualité de manière rapide en assemblant des composants préfabriqués. Pour ce faire, la programmation par composant a été développée comme une forme de livraison de pièces de logiciels génériques et exécutables pouvant donc être réutilisées dans différents environnements et applications.

3.1 Des objets aux composants

Depuis quelques années le but essentiel des nouvelles techniques de développement de logiciels a été la génération de logiciels réutilisables à grande échelle (*in the large*). La programmation par objets a permis, au départ, le développement de logiciels complexes en utilisant, par exemple, les schémas de conception (*design patterns*) [GHJV94] pour améliorer les conditions d'extensibilité du logiciel. L'approche objets, démontrant une bonne couverture des besoins dans le cadre de la programmation à petite échelle (*in the small*), a démontré parallèlement des limitations importantes. En premier lieu, la réutilisation basée sur l'héritage met en cause l'évolution du logiciel car l'implémentation à réutiliser doit être largement accessible. Dans ce cas, la réutilisation du logiciel est mise en œuvre à l'aide du mécanisme d'héritage : le logiciel est traité comme une *boîte blanche*. Un problème bien connu lié à la réutilisation par héritage est la dépendance forte vis-à-vis de l'implémentation initiale, mentionné par Mikhajlov et al. [MS98] comme le *problème de la classe de base fragile*. En second lieu, les objets se sont montrés insuffisants pour résoudre, de façon naturelle pour les développeurs, des problèmes associés à l'accroissement en complexité des environnements des applications. C'est le cas de la solution fournie par les *objets distribués* [OHE96] aux besoins du réseau des réseaux. Il a donc été nécessaire d'envisager des techniques permettant la réutilisation de façon modulaire de pièces génériques de logiciel encapsulant des propriétés fonctionnelles. Dans ce contexte, les objets ont évolué en développant des notions nouvelles comme les *composants* et les *aspects* [KLM+97, EFB01]. Cette évolution, considerée comme le pasage à une *période post-objets* par Cointe et al. [CND+04], a été motivée principalement par la nécessité de fournir des architectures logicielles facilement maintenables et adaptables.

Dans ce contexte, l'architecture est la description de l'organisation du logiciel comme un ensemble de composants, de connexions ainsi que les contraintes d'interactions entre les composants [BCK03, GS93, PW92]. Cette description, exprimée généralement à l'aide d'un *langage de description d'architecture* (*Architecture Description Language* ou *ADL*) [MT00], facilite l'implémentation et l'évolution du logiciel en exposant, par exemple, les composants intervenant dans l'implémentation d'une fonctionnalité déterminée du logiciel.

3.2 Notion de composant

"Components are for composition" [Szy02]. C'est la phrase sur laquelle la plupart des gents sont d'accord. Cependant, arriver à une définition de composant plus précise et compréhensible mais aussi acceptée par tout le monde devient une tache difficile comme le démontre le travail de Wallnau [BW98]. D'autres définitions d'un composant ont été développées par Szyperski [Szy02], Councill et al. [CH01], Marvie et al. [MP02], Larsson [Lar00], Sametinger [Sam97], entre autres. En les comparant, on observe de nombreuses différences concernant la granularité des composants, la nécessité de spécifier les dépendances contextuelles ainsi que l'autonomie des composants. Le problème semble être que toutes ces définitions sont valides dans leur contexte. Pour éviter de rendre le lecteur confus avec des comparaisons stériles nous adoptons, dans cette thèse, la définition proposée par Szyperski et Pfister à la conférence ECOOP'96 :

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

Cette définition couvre les caractéristiques des composants que nous considérons dans notre travail. Le développement à base de composants implique, donc, la construction d'applications par connexion de composants. Durant le développement à base de composants nous distinguons clairement deux rôles, d'une part le *producteur de composants* qui produit des composants réutilisables et d'autre part le *consommateur de composants* qui développe des applications en réutilisant les composants.

3.2.1 Réutilisation : boîte noire vs boîte blanche

Cette séparation entre le producteur et le consommateur s'appuie sur la vision d'un composant comme une boîte noire (*black box*) fournie par le producteur dont l'*implémentation* reste hors de portée du consommateur. Le producteur met à disposition du consommateur une *interface* qui n'est plus qu'une abstraction des fonctionnalités du composant. Toutefois, il existe des approches dans lesquelles les composants sont traités comme des *boîtes blanches* où l'interface contribue simplement à spécifier les limites du composant. Dans ce contexte, un composant est typiquement un morceau de code source susceptible d'être accédé et modifié par le consommateur.

Une abstraction intermédiaire considère les composants comme des *boîtes grises*. Contrairement à l'approche boîte blanche, une boîte grise ne permet que la visualisation des détails de l'implémentation du composant, laissant cette dernière *intouchable* lors de la réutilisation. Cette notion inclut aussi la possibilité de révéler une partie de l'implémentation comme étant une part de la spécification des interfaces [BW97].

En outre, un problème bien connu de la réutilisation de boîtes blanches est lié à l'utilisation de l'héritage. Cela implique une dépendance forte sur l'implémentation initiale, problème connu comme le *problème de la classe de base fragile* (voir, par exemple, [MS98]). Le remplacement de l'implémentation initiale du composant par une nouvelle version implique, généralement, la modification des applications construites préalablement.

3.2.2 Interfaces

Les composants sont censés communiquer à travers des interfaces qui contiennent toute l'information nécessaire afin que le consommateur arrive à une bonne composition. Dans la plupart des modèles de composants, l'interface exprime les *conditions requises* pour l'exécution du composant ainsi que les *fonctionnalités fournies* par le composant. On peut classifier cette information en *services requis* et *services fournis* par la boîte noire mentionnée auparavant. Cette interface a pour but de guider le processus de *composition* (assemblage) de composants en rejetant, par exemple, les compositions incorrectes. Du point de vue syntaxique, l'interface d'un composant est censée conduire la composition de composants, d'une part en exploitant l'information structurelle disponible pour la possibilité d'*interconnection* des composants, et d'autre part en faisant intervenir l'information sur le comportement du composant pour vérifier la compatibilité des *interactions* entre les composants composés [CH01, Szy02]. En revanche, en plus de ce rôle syntaxique, l'interface est assujettie aussi à un rôle sémantique où elle est transformée, au moment de la composition (au moins de façon conceptuelle), en une enveloppe (*wrapper*). Cette enveloppe peut être modifiée pour rendre possible la réutilisation de l'implémentation encapsulée dans un contexte particulier. Contrairement à l'implémentation, l'interface est adaptable par le consommateur du composant. Le composant peut alors être vue comme une boîte blanche.

3.3 Caractérisation d'un modèle de composants

Les modèles de composants peuvent être classés comme des *modèles industriels*, pour ceux qui sont utilisés dans l'industrialisation de composants, ou comme des *modèles académiques*, pour ceux qui ont été conçus pour explorer certaines caractéristiques des composants. Il existe différents modèles de composants qui se distinguent entre eux par les types de composants possibles, la façon dont ces composants sont assemblés, les étapes qui déterminent le cycle de vie des composants, la capacité d'adaptation des composants, le degré de fusion des composants lors de la composition, entre autres. Dans cette section, nous allons détailler quelques caractéristiques présentes dans la plupart des modèles de composants et qui seront utilisées pour la description de plusieurs de ces modèles.

3.3.1 Composants

Selon le modèle, un composant est défini de différentes manières et en utilisant différents moyens, tels que les langages textuels, comme les ADL mentionnés auparavant, et les langages graphiques, comme par exemple le langage UML [SW99, RJB99].

La définition d'un composant est formée principalement par l'interface. Cette interface décrit les éléments qui interviennent au moment de la composition, comme par exemple, des ports, des canaux de communication, des signatures de fonctions, etc. Il existe, par contre, des modèles qui n'incluent pas l'implémentation comme faisant partie de la définition du composant. Dans ce cas, le modèle permet généralement d'exprimer des propriétés abstraites sur les composants et les compositions modélisées (par exemple, Wright [AG97],

SADL [MQR95]). En plus, le modèle doit spécifier l'existence des différents types de composants. En général, nous trouvons deux types de composants : le composant *primitif*, par opposition à un composant *composé* résultant de la composition d'autres composants.

Pour la description des composants on utilise des langages de description de composants, comme les ADL mentionnés ci-dessus, et des langages de programmation pour l'implémentation des composants. Cette séparation permet la définition d'une architecture créée à partir de composants où seulement les interfaces sont utilisées au moment de la composition. Dans ce cas, nous pouvons envisager, parmi les avantages principaux, la réutilisation de l'architecture pour différentes implémentations ainsi que la réutilisation pour différents langages d'implémentation (par exemple, CCM-IDL [Gro02], Rapide [LV95], UniCon [SDK+95]). Cependant, le découpage dans la description demande un grand effort pour garantir que l'implémentation obéisse aux contraintes établies par l'architecture donnée. Une alternative à cette approche est l'extension d'un langage de programmation avec des constructions facilitant la représentation des composants intervenant dans l'architecture (par exemple, ArchJava [ACN02b], ComponentJ [CSC02], Cell [RS02]). Cela nous empêche, par exemple, de travailler avec une description à deux niveaux. En plus, même si l'approche à deux niveaux présente un avantage en ce qui concerne la flexibilité du choix entre différents langages de programmation pour une même architecture, les choses ne s'avèrent pas très claires au moment d'établir la relation entre les types de données. Par exemple, le type de donnée `Integer` utilisé dans la description pourrait correspondre au type `int` dans le langage Java ou `Integer` dans le langage Pascal.

3.3.2 Composition

La construction d'une application s'appuie sur l'assemblage des composants, appelé aussi *composition* des composants. Dans un contexte général, le consommateur de composants assemble des composants en utilisant leur interface. La composition des composants utilise l'information fournie à travers leur interface où l'idée générale est de satisfaire les besoins d'un composant grâce aux services fournis par un autre composant. Il arrive souvent que les services requis par un composant ne soient pas tout à fait les services fournis par d'autres composants. Donc, pour résoudre cette incompatibilité on utilise un *adaptateur* qui permet de remédier aux incompatibilités. Cet adaptateur, appelé aussi *code de colle* (*glue code*) [CORS98], est parfois considéré aussi comme un type de composant et non pas comme un simple morceau de code qui facilite la composition de deux composants dont les interfaces sont incompatibles. Dans certains ADL, c'est aussi consideré comme une entité spécifique, à savoir un *connecteur*.

Dans le cadre de cette thèse, nous décrivons la composition dans les modèles étudiés en identifiant les caractéristiques suivantes.

Hiérarchique. En observant le résultat de la composition nous constatons la faiblesse de la ligne qui divise les deux rôles. Selon le cas, le consommateur peut devenir aussi un producteur de composants en définissant un nouveau composant prêt à être réutilisé ultérieurement. Dans ce cas, on dit que le modèle considère la composition *hiérarchique* où *incrémentale* en permettant la création de composants composés.

Fusionnelle. Considérons, pour l'instant, les phases de construction et d'exécution dans la vie du composant (voir section 3.3.3 pour plus de détails). Quand il est possible d'identifier à l'exécution les composants qui ont été assemblés au moment de la composition, on dit qu'il s'agit d'une composition *non-fusionnelle*. Par contre, dans un cadre de composition fusionnelle, l'information fournie lors de la spécification est diluée après la composition et, par conséquent, il n'est plus possible d'accéder à l'information concernant l'architecture d'origine. L'approche la plus flexible consiste à disposer d'une opération qui permet de sceller un composant structuré. On peut ainsi, soit le laisser *"ouvert"* (la structure interne est accessible si nécessaire), soit le sceller, le transformant alors en composant primitif.

Dynamique. Le remplacement, connu aussi sous le nom de *reconfiguration*, est principalement utile quand il a lieu durant l'exécution du composant. Si c'est le cas, on parle d'une composition *dynamique*, sinon on parle d'une composition *statique* où le résultat de la composition reste inchangé durant tout le cycle de vie. Il existe aussi des modèles où une forme intermédiaire de reconfiguration est possible. Dans ce cas, la composition, qui reste inchangée, est décrite en terme d'une structure ou patron telle que durant l'exécution cette structure est instanciée par des composants concrets. Autrement dit, au moment de la description on connaît la forme de la composition mais on ne connaît pas les éléments. La composition non-fusionnelle facilite le remplacement de composants.

3.3.3 Processus de Développement

Les modèles de développement traditionnels, en cascade (*waterfall*), itératif, en spirale et par prototypage, voient se modifier le processus de construction d'applications avec l'introduction des composants. Par exemple, Brown et al. [BW96] et Morisio et al. [MSP+00] présentent deux modèles de développement à base de composants. Dans le premier cas, il se différencie principalement du modèle en cascade, tandis que, dans le deuxième cas, il s'agit d'un modèle purement dédié à la conception, la codification et l'intégration de composants.

Afin de donner une description simple du processus de développement nous n'identifions que deux grandes phases : la définition de composants susceptibles d'être réutilisés et la construction d'applications par la composition de composants développés préalablement. Ces phases correspondent respectivement aux rôles mentionnés auparavant. Tandis que le producteur de composants intervient pendant la première phase, connue aussi comme *développement pour réutiliser* (*for reuse*), c'est le consommateur de composants qui intervient lors de la deuxième phase, connue aussi comme *développement par réutilisation* (*by reuse*) [Mey94]. La figure 3.1 montre le processus de développement à base de composants.

Lors de la production de composants se distinguent deux étapes : l'étape de *construction* de composants et l'étape de *livraison* de composants.

Construction. Cette étape consiste à créer les composants, c'est-à-dire à fournir la définition des interfaces représentée par la couche blanche autour de l'implémentation qui est elle-même représentée par la boîte noire dans la figure.

Livraison. Une fois le composant créé, il doit être empaqueté et stocké dans des dépôts (*repository*) de composants pour être livré. Parfois, cette étape est associée au proces-

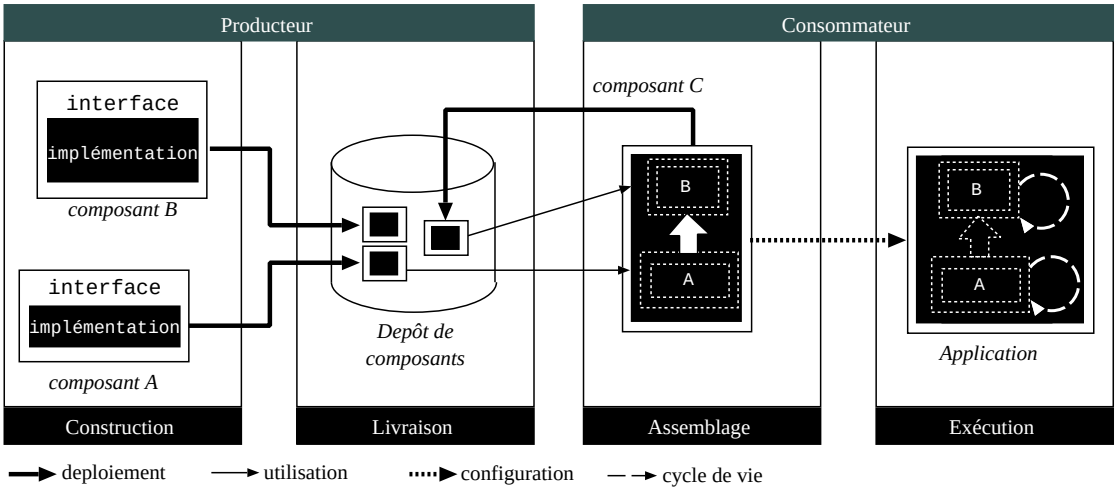


FIG. 3.1 – Processus de développement à base de composants.

sus de *déploiement* de composants. Le déploiement (ou l'installation) d'un composant implique la mise à disposition du composant pour des réutilisations ultérieures, dans l'environnement de développement correspondant.

La consommation de composants se divise en deux étapes principales : l'assemblage (ou composition) et l'exécution.

Assemblage. Le consommateur identifie et sélectionne, depuis les depôts, les composants compatibles selon les besoins spécifiés. Il crée de nouvelles applications en assemblant des composants disponibles. Dans certains modèles, cette étape implique aussi la configuration des composants, laquelle consiste à donner des valeurs concrètes qui seront prises en compte au moment de l'instanciation des composants.

Exécution. Si la composition a toutes ses dépendances satisfaites, alors elle devient une application exécutable. À l'exécution, les composants sont instanciés en fonction de l'architecture spécifiée. Dans certains cas, les composants respectent un cycle de vie géré de manière explicite par l'environnement d'exécution, comme par exemple, instanciation, activation, désactivation, destruction.

3.4 Modèles industriels

Dans cette section nous nous concentrons sur la description de trois infrastructures de composants qui supportent diverses variantes de composants.

3.4.1 Enterprise Java Beans (EJB)

Enterprise Java Beans [DeM03] de Sun Microsystems est une infrastructure de composants pour des composants serveurs (*server-side components*) pour la construction d'applications distribuées. EJB est une partie de la technologie *Java 2 Platform Enterprise*

Edition (J2EE) laquelle s'appuie sur d'autres technologies comme l'invocation distante de méthodes (*RMI - Remote Method Invocation*), le système de nommage et de répertoire de services (*JNDI*), le système de connectivité aux bases de données (*JDBC*), entre autres. La figure 3.2 montre l'architecture à trois niveaux (*3-tiers*) d'une application basée sur des composants EJB. Dans la figure nous distinguons la couche de présentation pour interagir avec les clients, la couche de traitement qui contient toute la fonctionnalité métier, et, finalement, la couche de persistance qui permet le stockage d'informations en utilisant un serveur de base de données. Différents serveurs d'applications sont disponibles, tels que JOnAS [JOn], JBoss [JBo], WebLogic [Web], ..., en plus de l'implémentation de référence fournie par Sun.

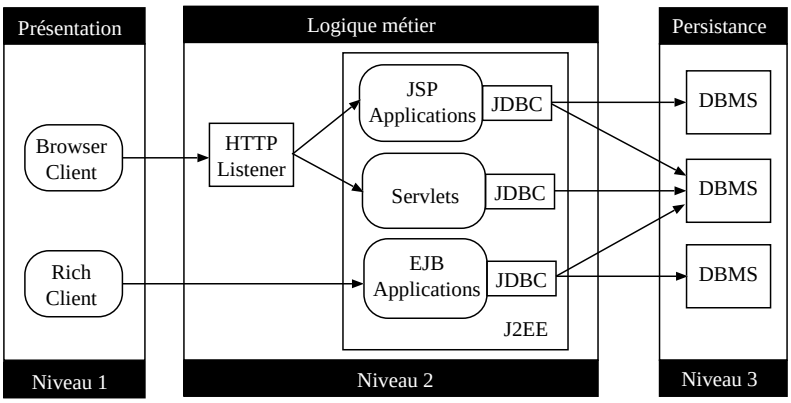


FIG. 3.2 – EJB : utilisation de composants EJB dans une architecture à trois niveaux.

3.4.1.1 Composants

Un JavaBean [Ham97] (voir section 3.4.4) est un cas particulier d'une classe Java. Un composant EJB est un JavaBean implantant un certain nombre de méthodes spécifiques. Ces méthodes permettent au conteneur de gérer les composants uniformément lors de la création et l'interaction (transactions, persistance, etc.). Un composant EJB, appelé aussi simplement *bean*, se compose de trois éléments : *l'interface distante* qui contient les méthodes qu'un client de l'EJB peut appeler, *l'interface locale* qui définit les méthodes qu'un client peut invoquer pour créer, trouver ou supprimer l'EJB et, finalement, la classe qu'implémente le composant proprement dit. En plus des interfaces et de l'implémentation, un composant EJB inclut un *descripteur de déploiement* qui permet la spécification des services offerts par l'environnement d'exécution du composant, appelé le conteneur, tel que la sécurité, la persistance et les transactions.

Il existe trois catégories de beans : les beans *session* (*session beans*), les beans *entités* (*entity beans*) et les beans *message* (*message-driven beans*).

En premier lieu, un bean *session* est chargé d'effectuer une tâche pour un client. Dans cette catégorie d'Entreprise JavaBeans, il y a deux groupes. D'une part, les beans sessions *sans état* (*stateless*), qui ne maintiennent pas d'état conversationnel. Toutes les invocations de méthodes faites sur un EJB sans état ne seront pas forcément traitées par le même

EJB et un même EJB pourra traiter les requêtes de plusieurs clients. D'autre part, les beans session *avec état* (*statefull*), dédiés à un certain client pendant toute la durée de son instanciation. Concrètement, si on modifie une variable d'instance de la classe qui implémente le composant, on retrouve cette valeur lors des prochains appels. Pour résumer, toutes les invocations d'une méthode par le client seront traitées par le même bean. La figure 3.3 montre la définition d'un composant de type session.

(a) Interface Distante : `HelloWorld.java`.

```
package sb;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface HelloWorld
                extends EJBObject {

   public String sayHelloWorld()
                throws RemoteException;
}
```

(b) Interface Locale : `HelloWorldHome.java`.

```
package sb;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface HelloWorldHome
                extends EJBHome {

  HelloWorld create()
            throws RemoteException,
                   CreateException;

}
```

(c) Classe du bean : `HelloWorldBean.java`.

```
package sb;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean
             implements SessionBean {

  public String sayHello() {
   return "Bonjour, le monde";
  }

  public void ejbCreate() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(
              SessionContext sc) {}

}
```

FIG. 3.3 – EJB : exemple "Bonjour, le monde".

En second lieu, à la différence des beans session qui sont détruits lorsque la session du client se termine, l'état d'un bean entité est sauvegardé sur un support de stockage externe (comme une base de données). Autrement dit, la durée de vie d'un bean entité n'est pas liée aux clients. Si nous voulons stocker les informations de l'application comme les factures ou les articles, ils peuvent répondre à ce besoin car ils sont persistants. La persistance peut être gérée soit au niveau du composant (*BMP - Bean Managed Persistence*) soit au niveau du conteneur (*CMP : Container Managed Persistence*). Dans le premier cas, tous

les appels provoquant un accès à la base de données sont dans le code de l'EJB, tandis que dans le deuxième cas, au lieu d'inclure le code correspondant dans l'implémentation du composant, c'est le conteneur qui se charge des appels à la base de données.

En dernier lieu, les beans message permettent le traitement des messages asynchrones [Haa02]. Ce type de composant n'a pas d'état interne (par conséquent, il n'est pas persistant) et peut établir des communications avec plusieurs clients.

3.4.1.2 Composition

Dans ce modèle, il est possible de composer différents composants pour créer un nouveau composant. L'information sur la composition est exprimée explicitement dans le descripteur du composant englobant (balise `ejb-ref`). Par contre, cette information n'est utilisée qu'au moment de la compilation du composant. Le résultat de la compilation, un fichier `.jar`, est le même pour tous les types de composants, même pour le composant composé (voir le cycle de vie détaillé ci-dessous). La reconfiguration n'est possible que de manière statique, c'est-à-dire que tout remplacement d'un composant requiert la recompilation du composant dans sa totalité. Même si dans la spécification du modèle il n'existe aucune référence aux formes de compositions entre composants EJB, on peut classifier la relation entre les composants EJB comme une composition hiérarchique, statique et non-fusionnelle.

3.4.1.3 Processus de développement

Le modèle EJB s'intéresse principalement aux activités liées au développement, au déploiement, et à l'exécution d'une application. La spécification définit six rôles qui interviennent lors du processus de développement associé aux composants. Pour cela, le modèle spécifie une sorte de contrat qui assure que le produit généré par chaque rôle est compatible pour les autres rôles.

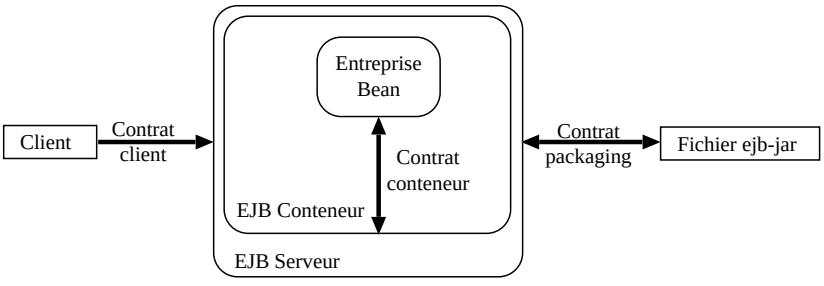


Fig. 3.4 – EJB : Contrats de l'architecture.

– **Fournisseur de composants.** C'est notre producteur de composants. Il est responsable de fournir l'implémentation du composant, les interfaces du composant (i.e. locales et distantes) ainsi que le descripteur correspondant. Ce rôle doit fournir comme résultat un fichier `.jar` (*ejb-jar file*). Ce fichier est conforme à un format particulier, supporté par tous les outils liés aux EJB, défini dans le *contrat packaging*.

En plus des éléments mentionnés, le fichier inclut aussi les classes (*classes d'interposition*) qui permettent au composant de communiquer avec les conteneurs et de respecter le *contrat côté conteneur*. La figure 3.4 montre les différents contrats dans l'architecture EJB.

- **Assembleur d'applications.** L'assembleur d'applications combine des composants en une nouvelle unité. Concrètement, l'assembleur prend un ou plusieurs fichiers *ejb-jar* générés par les fournisseurs de composants et produit un nouveau composant (fichier *ejb-jar*) qui contient les *instructions d'assemblage* dans le descripteur correspondant.
- **Installateur.**
  Connu aussi comme le *déployeur* (*deployer*), l'installateur rend opérationnels les composants, produits autant par le fournisseur que par l'assembleur, à l'aide d'un serveur de composants EJB et d'un conteneur. Il doit généralement assurer que toutes les dépendances externes spécifiées par le composant (par exemple, droit d'accès dans le serveur où le composant est installé) sont disponibles.
- **Fournisseur de serveur.** Actuellement, les spécifications sur l'architecture des composants EJB, considèrent que ce rôle et celui du fournisseur de conteneur sont assurés par la même entité (personne, société ...). C'est un vendeur d'*intergiciel* (*middleware*), un vendeur de base de données, etc.
- **Fournisseur de conteneur.** Ce rôle doit fournir les outils nécessaires pour le déploiement et le support d'exécution des composants. Typiquement, un fournisseur de conteneur fournit le support pour gérer les versions des composants déjà installés. Il peut aussi fournir les outils nécessaires pour que l'administrateur du système puisse contrôler le conteneur.
- **Administrateur du système.** L'administrateur du système est responsable de la configuration et de l'administration de l'environnement opérationnel dans lequel sont inclus les serveurs et les conteneurs.

La figure 3.5 montre les rôles décrits ci-dessus liés aux étapes du cycle de vie des composants.

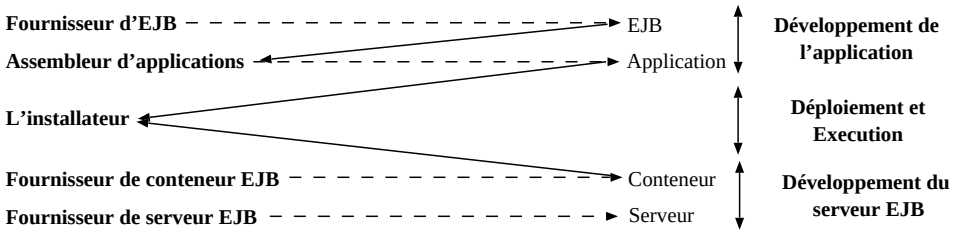


FIG. 3.5 – EJB : processus de développement et ses rôles.

3.4.2 Component Object Model (COM)

COM [Löw01] est le modèle de composant de Microsoft inventé pour simplifier l'intégration de logiciels. Ce modèle a été ensuite étendu par le standard DCOM *Dis-*

tributed Component Object Model [EE98], qui spécifie la création de composants distribués sur des machines différentes aux systèmes d'exploitation différents. COM+ inclut aussi le support pour des transactions, annuaires de services, ... La figure 3.6 montre comment un client peut se connecter à un serveur d'applications, en passant par un serveur de services internet (*Internet Information Server - IIS*) ou par le protocole DCOM dans le cas d'un client normal de composants. À son tour, une application peut aussi se connecter à une base de données. Notez la ressemblance avec l'architecture de EJB (voir figure 3.2).

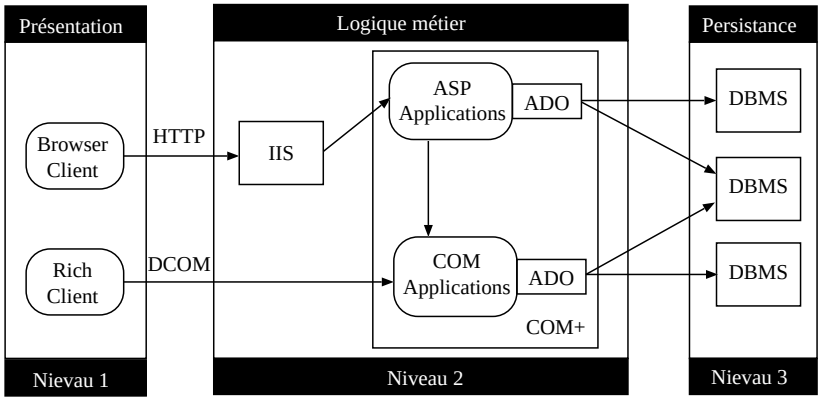


FIG. 3.6 – COM : utilisation de composants COM dans une architecture 3-niveaux.

3.4.2.1 Composants

COM étant une spécification de composants binaires, elle ne spécifie rien sur comment un langage de programmation est associé au modèle. De plus, le modèle ne définit pas concrètement ce qu'est un composant ou un objet. Ce modèle a été conçu comme un standard ouvert à plusieurs plates-formes. Cependant, il a été principalement utilisé dans Microsoft Windows pour lequel il a été initialement créé.

Tirant partie d'une orientation objet, COM permet l'encapsulation, le polymorphisme et la réutilisation des composants dont l'intégration binaire, non fondée sur le code source, permet de ne pas remettre en cause un logiciel tout en augmentant les services et fonctionnalités offertes aux applications. En fait, COM est, à la base, une spécification permettant l'appel de méthodes et l'accès à des propriétés d'un objet, à partir de n'importe quel langage de programmation.

L'entité principale dans ce modèle est l'*interface*. Une interface est représentée, au niveau binaire, comme un pointeur. Concrètement, l'interface COM est un point d'encrage d'un composant qui est entièrement spécifié et qui est toujours disponible lorsque le composant respecte le standard COM. Elle permet l'accès à un ensemble de méthodes conformes au standard binaire de COM, indépendamment des langages d'appel du composant logiciel et de la machine sur laquelle il est localisé. La figure 3.7 montre l'utilisation d'une interface à l'exécution.

Un composant COM, connu aussi comme un objet COM, peut contenir plusieurs interfaces implémentées par une ou plusieurs classes. La figure 3.8 montre un composant

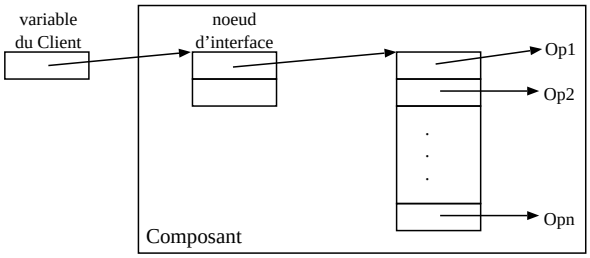


FIG. 3.7 – COM : interface du point de vue binaire.

fournissant les interfaces A et B dont l'implémentation est assurée par l'objet 1, tandis que l'interface C est implémentée par l'objet 2.

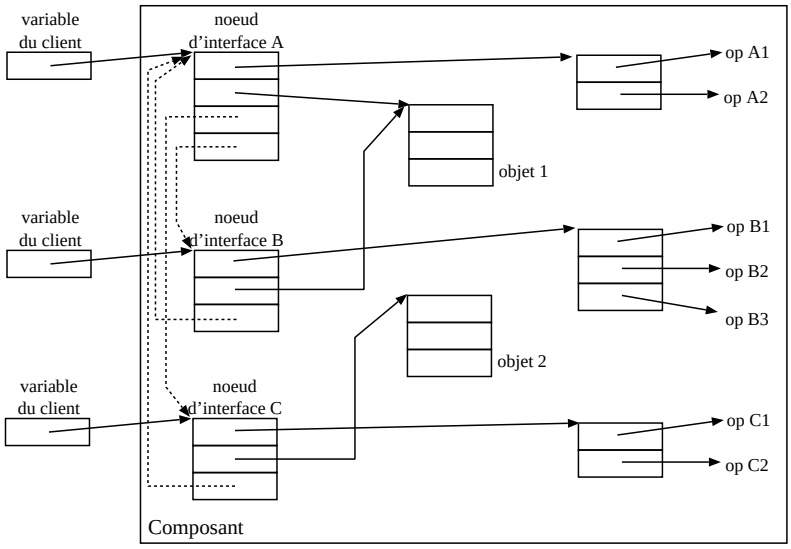


FIG. 3.8 – COM : un composant fournissant plusieurs interfaces.

Le modèle spécifie que toutes les interfaces associées à un composant COM doivent contenir la méthode `QueryInterface`. Cette méthode prend le nom d'une interface, vérifie si cette interface est fournie par le composant et, si c'est le cas, retourne une référence à l'interface. Afin d'identifier une interface, en plus d'un nom qui peut être partagé par plusieurs interfaces, un identificateur d'interface (*Interface Identifier - IID*) est associé à chaque interface.

Tout composant COM gère au moins l'interface `IUnknown`. Cette interface comporte les membres `AddRef` et `Release` pour la gestion des utilisations actives du composant, et le membre `QueryInterface` pour la raison mentionnée ci-dessus. Chaque composant COM a une seule interface `IUnknown`, par conséquent, l'identificateur de l'interface `IUnknown` peut être utilisé pour identifier le composant correspondant. La figure 3.9 montre la description d'une interface `IUnknown` qui contient les trois méthodes obligatoires pour toutes les interfaces COM.

```
[ uuid (00000000-0000-0000-C000-000000000046)]
interface IUnknown {
 HRESULT QueryInterface([in] const IID iid, [out, iid_is(iid)] IUnknown iid);
 unsigned long AddRef();
 unsigned long Release();
}
```

FIG. 3.9 – COM : Description d'une interface `IUnknown`.

3.4.2.2 Composition

Le modèle ne définit pas un mécanisme particulier de composition pour les composants. La composition n'est possible qu'au niveau des objets COM apparaissant dans l'implémentation. Concrètement, les deux formes de composition sont la *contention* (*containment*) et l'*agrégation* (*aggregation*).

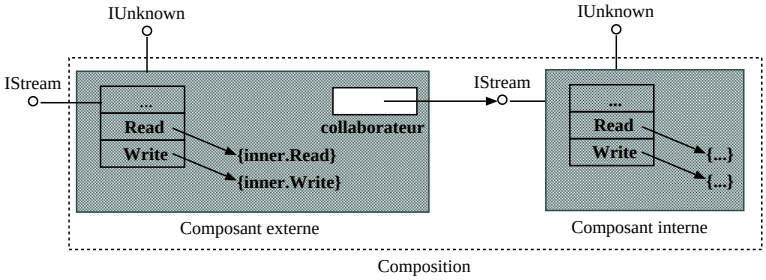
La contention est simplement implémentée à travers une référence d'un composant vers un autre. Le composant référencé est appelé *interne* (*inner*) tandis que le composant référençant est appelé *externe* (*outer*). Si un service fourni par le composant (englobant) externe est en réalité géré par le composant interne, cela implique simplement un transfert (*forwarding*) de l'appel sur le service fourni par le composant interne. Ce type de délégation est, bien sûr, transparent pour le client du composant externe. La figure 3.10(a) montre la composition de composants à travers la contention.

Parfois, le transfert des appels inhérent à la composition par contention provoque la dégradation de la performance de la composition résultante. Pour remédier à ce problème, il est possible de composer des composants en utilisant l'agrégation. L'idée est simplement d'extérioriser la référence de l'objet interne directement à travers l'interface de l'objet externe, pour éviter le coût des transferts des appels.

Ce type de composition est aussi transparent pour le client du composant externe mais, contrairement à la contention, l'agrégation ne s'avère pas transparente pour les composants internes. Concrètement, l'interface du composant externe est, maintenant, celle fournie par le composant interne. Mais, cette interface devra connaître les services fournis aussi par le composant externe. La solution est simple, les appels à la méthode `QueryInterface` sont transférés à l'interface du composant externe. La figure 3.10(b) montre la composition par agrégation. On observe que l'agrégation implique la référence mutuelle entre les deux composants. Cela permet au composant externe d'utiliser les services des composants internes sans être renvoyé à son interface, c'est-à-dire d'éviter un appel cyclique. L'agrégation est généralement utilisée pour l'implémentation des enveloppes de composants.

Notez que la composition obtenue par contention conserve l'identité des deux composants composés, c'est-à-dire qu'il s'agit d'une composition non-fusionnelle. Cependant, la composition par agrégation est intrusive par rapport aux composants composés, il s'agit d'une composition fusionnelle.

(a) Composition par *contention*.



(b) Composition par *agrégation*.

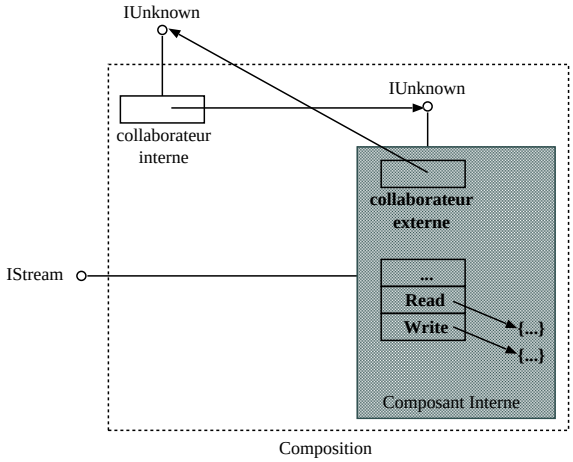


FIG. 3.10 – COM : Composition des composants.

3.4.2.3 Processus de développement

Étant une spécification au niveau binaire, COM ne spécifie pas le cycle de vie des composants. Cependant, à partir de la documentation existante, nous pouvons identifier les phases de création, composition et déploiement de composants, ainsi que le développement d'applications.

- **Création d'un composant.**
  Le producteur de composants spécifie l'interface métier du composant en plus de l'interface `IUnknown`. Lors de la compilation, le producteur affecte aux interfaces l'implémentation correspondante. Comme résultat, le producteur peut produire deux types de fichier selon le type de composant souhaité. Si le composant doit être réutilisé de manière locale ou distante alors il doit fournir un fichier `.EXE`, tandis que si la réutilisation est uniquement locale alors un fichier `.DLL` est suffisant.
- **Composition de Composants.** Le producteur de composants peut générer de nouveaux composants en appliquant les techniques de contention et d'agrégation expliquées auparavant.
- **Déploiement de Composants.** Dans cette phase les composants COM sont rendus disponibles pour leur utilisation. L'interface d'un composant COM est localisable

par la librairie COM de la machine d'exécution de l'application, à l'aide d'un identificateur de classe (*CLSID*). Le *CLSID* est généré lors de la création du composant logiciel. Il est unique, codé sur 128 bits et répond à la norme DCE (*Distributed Computing Environment*) de l'OSF (*Open Software Foundation*). Il est enregistré dans une base de données d'inscription (*Registry* sur Windows).

– **Développement d'Applications.** Un client COM est un morceau de code dans un langage de programmation donné, qui met en œuvre un pointeur d'interface vers un serveur COM. Un client COM fait appel à un serveur COM en passant le CLSID à la librairie COM. Cette dernière dispose d'un service SCM (*Service Control Manager*) qui est chargé de trouver, localement ou sur le réseau, le composant associé au CLSID. Le client n'est pas concerné par la manière dont le composant a été empaqueté (i.e. `.EXE` ou `DLL`).

3.4.3 CORBA Component Model (CCM)

Le modèle CCM (*CORBA Component Model*) [Gro02, Sie00] est la réponse aux besoins de développement à base de composants dans une architecture distribuée CORBA (*Common Object Request Broker Architecture*) [Cor]. Le CCM a été intégré dans la version de CORBA 3. Tout comme les EJB de Sun, la technologie CCM repose sur l'utilisation de conteneurs pour héberger les instances de composants et faciliter leur déploiement. Un serveur de composants est un processus qui contient un nombre arbitraire de conteneurs de composants. Chaque conteneur, associé à un type de composant (service, session, processus et entité), est géré par un POA (*Portable Object Adapter*). Le POA est responsable de la création et destruction du conteneur. Différents services sont rendus disponibles pour les composants à travers le bus de service ORB. La figure 3.11 montre l'architecture proposée par le modèle CCM.

3.4.3.1 Composants

Un composant dans le cadre de CCM, connu aussi comme un *type de composant*, regroupe la définition d'*attributs* et de *ports*. Une particularité des attributs est qu'il est possible de lever des exceptions lorsqu'ils sont mal utilisés. Un port représente une interface soit fournie, soit utilisée (requise) par le composant. CCM spécifie quatre types de ports :

– Une *facette* est une interface fournie par un composant et qui est utilisée par des clients en mode synchrone.
– Un *réceptacle* est une interface utilisée par un composant en mode synchrone.
– Un *puit d'événements* est une interface fournie par un composant et utilisée par des clients en mode asynchrone.
– Une *source d'événements* est une interface utilisée par un composant en mode asynchrone.

La figure 3.12 montre la modélisation d'un distributeur de boissons en utilisant CCM.

Pour définir les composants, ce modèle introduit une version enrichie du langage OMG IDL (*Interface Description Language*), connu comme IDL3 pour le distinguer de la version originale, IDL2, spécifiée par CORBA 2. Par contre, le producteur définit des composants

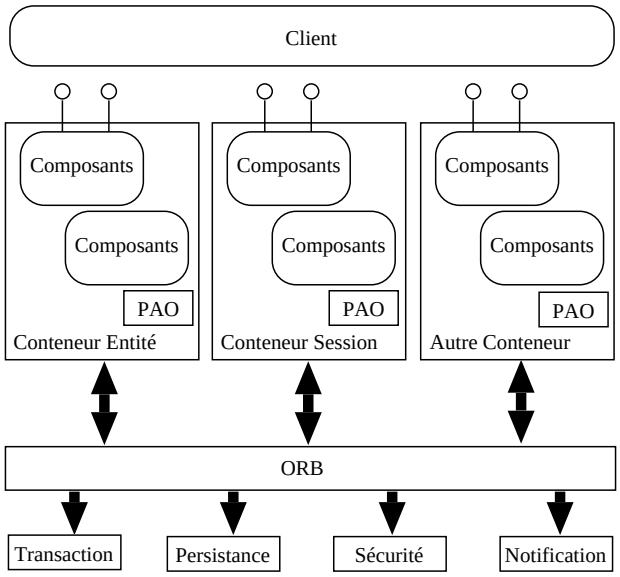


FIG. 3.11 – CCM : modèle de Conteneur.

à l'aide du langage IDL3, mais cette construction ne sert que pour la définition et non pas pour le déploiement. Pour le déploiement la définition IDL3 doit être projetée en IDL2 en utilisant un ensemble de règles qui associent chaque construction en IDL3 à une construction IDL2. Ici, nous nous concentrons sur la définition IDL3 des composants. La figure 3.13 montre la définition IDL3.

La déclaration d'un component se fait à l'aide du mot-clé `component`. Cette déclaration reste au même niveau que celle d'une `interface`. Les interfaces sont définies au sens de CORBA2, c'est-à-dire sous la forme d'un ensemble de fonctions (méthodes).

Dans la définition, l'attribut `on` est défini en utilisant le mot-clé `attribute` plus le type, ici, `boolean`.

Une facette représente un point de vue sur un composant. Ce type de port est utilisé essentiellement pour regrouper logiquement des opérations associées au composant. Il est possible de définir une facette par rôle joué par un composant. Si on regarde le distributeur de boissons, plusieurs interfaces sont disponibles : une interface pour le consommateur (payer, sélectionner une boisson, prendre la boisson), une interface pour le fournisseur (ouvrir distributeur, mettre des boissons, mettre de la monnaie) et une interface pour le dépanneur (ouvrir le capot arrière, purger la pompe). En plus de la *référence de base* d'une instance de composant, chaque référence dispose de sa propre référence, appelée *référence de facette*. La déclaration d'une facette se fait à l'aide du mot-clé `provides`. Pour chaque facette il est nécessaire de déclarer une clause `provide` associée à un nom unique.

Un réceptacle permet à un composant d'accepter une référence d'objet, à savoir une référence de facette, de composant ou un objet au sens CORBA 2. Cette relation, appelée *connexion*, est la manière de définir des compositions de composants. Un réceptacle permet donc d'assembler des instances de composants et potentiellement des objets. Les réceptacles peuvent être de deux types : des réceptacles simples (une seule référence), ou

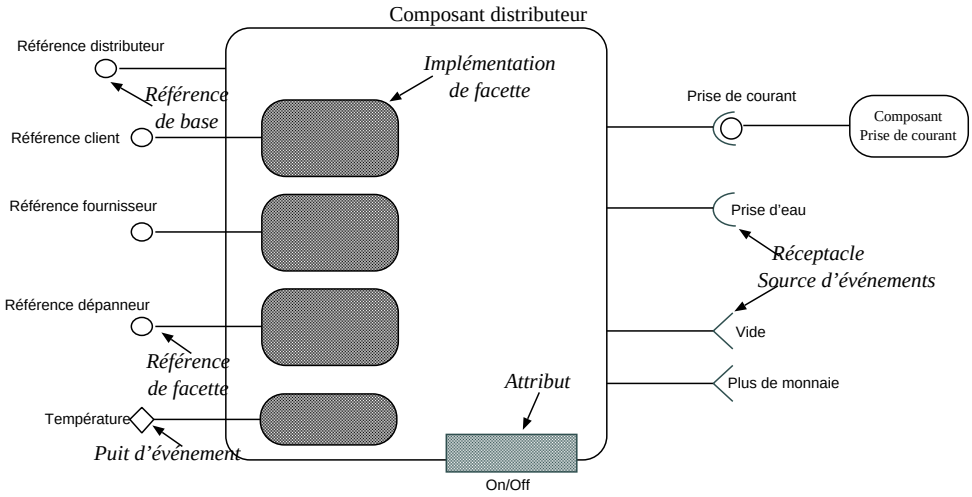


FIG. 3.12 – CCM : modélisation d'un distributeur de boissons.

des réceptacles multiples (plusieurs références). Dans l'exemple, le réceptacle `PriseEau` est simple, alors que le réceptacle `PriseCourant` est multiple. Dans le cas des réceptacles multiples, une limite pour le nombre maximum de connexions simultanées peut être fixée lors de l'implémentation du composant. Pour les deux types de réceptacles, la déclaration se fait au moyen du mot-clé `uses` tel que présenté dans la figure 3.13. Dans le cas de réceptacles multiples, le mot-clé `multiple` s'ajoute à l'utilisation du mot-clé `uses`.

Le modèle d'événements est du type producteur/consommateur (*publisher/subscriber*). Les mécanismes d'événements sont définis par les interfaces des consommateurs. Il y a deux types de ports qui traitent des événements : les *sources* et les *puits* d'événements. Dans le cas des sources d'événements, ils sont de deux ordres. *Emitter* est la catégorie de source qui n'accepte qu'un seul consommateur (un vers un). Dans ce cas, il n'y a pas de médiation par un canal d'événements. La connexion producteur/consommateur est directe. *Publisher* est la catégorie de source d'événements qui accepte plusieurs consommateurs (un vers *plusieurs*). Dans ce cas, l'abonnement d'un consommateur à un type d'événements est délégué à un canal d'événements fourni par l'infrastructure. Dans le cas des puits d'événement, ils permettent à un composant de recevoir des événements d'un certain type. Il n'y a pas, ici, de différentiation entre connexion et abonnement. Le composant ne contrôle pas ce récepteur, il le rend public pour recevoir des événements. Le puit peut donc recevoir des événements en provenance de plusieurs producteurs. La déclaration de sources d'événements se fait selon le cas avec le mot-clé `emits` (un vers un) ou `publisher` (un vers *plusieurs*). Pour les puits d'événements, la déclaration se fait à l'aide du mot-clé `consumes`.

En ce qui concerne l'implémentation des composants, le modèle CCM inclut le langage CIDL (*Component Implementation Definition Language*). Ce langage permet de décrire la structure de l'implémentation d'un composant. Pour cela, un composant est considéré ici comme un ensemble d'éléments comportementaux fournis par des *exécuteurs*. Deux types d'exécuteurs sont définis : les exécuteurs des composants et les exécuteurs de *maisons*

(a) Définition du composant `Distributeur`.

```
component Distributeur {
   attribute boolean on ;
   provides FacadeClient client;
   provides FacadeFournisseur fournisseur;
   provides FacadeDepanneur depanneur;
   uses PriseEau prise_eau;
   uses multiple PriseCourant prise_courant;
   emits PlusDeMonnaieEvt pdm;
   publishes DistributeurVideEvt dvide;
   consumes TemperatureEvt temp;
};
```

(b) Interfaces et types d'événements.

```
interface FacadeClient {...};
interface FacadeFournisseur {...};
interface FacadeDepanneur {...};
interface PriseCourant {...};
interface PriseEau {
  void ouvrir ();
  void fermer ();
  void fournirEau ();
};
valueType PlusDeMonnaieEvt :
          Component::EventBase {...};
valueType DistributeurVideEvt :
          Component::EventBase {...};
valueType TemperatureEvt :
          Component::EventBase {...};
```

FIG. 3.13 – CCM : Définition IDL3 du composant `Distributeur`.

(*home*) de composants. Une maison de composants est un gestionnaire pour des instances d'un même type de composant. Elle gère le cycle de vie des instances de composants. Pour cela, elle offre une fabrique d'instances de composants et des opérations de recherche. Un composant est défini de manière indépendante des types de maison. Par contre, une maison doit spécifier le type de composant qu'elle va gérer. En CIDL, la relation entre la description des composants et leur implémentation est appelée *composition*. Une composition, telle que présentée dans la figure 3.14, spécifie une maison, et donc implicitement un composant. Ensuite, une définition d'exécuteur est associée à la maison.

```
composition DistributeurImpl {
 home executor MaisonDeDistributeursImpl;
      implements corba::MaisonDeDistributeurs;
      manages DistributeursImpl;
```

FIG. 3.14 – CCM : définition CIDL de la maison du composant `Distributeur`.

Celle-ci spécifie la relation entre l'exécuteur de maison et les autres éléments de la composition. Enfin, une composition spécifie une définition d'exécuteur de composant. Cet exécuteur peut potentiellement être segmenté, c'est-à-dire que l'implémentation de l'exécuteur sera décomposée en plusieurs classes. Les relations entre le type de maison, le composant, et les exécuteurs sont décrites dans la figure 3.15

Catégories de Composants. Quatre catégories de composants sont définies par CCM : *service*, *session*, *processus* et *entité*. Les composants *service* sont des composants sans

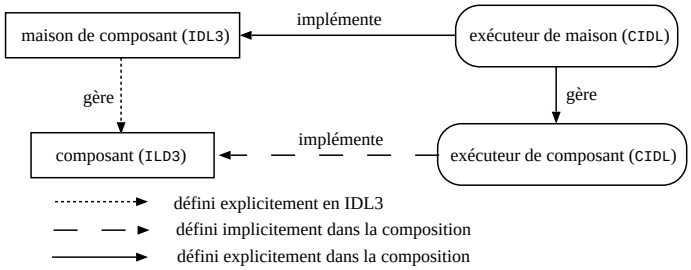


FIG. 3.15 – CCM : Relations entre la maison, le composant et l'exécuteur.

état et sans identité. Leur durée de vie correspond à la durée de traitement requis par l'invocation d'une opération. Les composants *session* sont des composants avec un état volatil, et une identité qui n'est pas persistante. La durée de vie d'un tel composant est une interaction de la part du client. Les composants *processus* sont des composants avec un état persistant géré par le composant ou par le conteneur. Le fait que cet état soit persistant n'est pas visible pour le client. Ce type de composant est utilisé pour modéliser les objets qui représentent des processus métiers, plutôt que des entités. Les composants *entité* sont des composants similaires aux composants session mais dans ce cas la persistance est visible par le client.

3.4.3.2 Composition

La description et l'implémentation d'un composant est déployée de manière individuelle dans un package. CCM prévoit la composition de composant par l'assemblage de composants déployés. Autrement dit, la composition de composants n'intervient pas au moment de la définition des composants intervenants. L'assemblage, en CCM, offre un patron pour instancier un ensemble de composants et les connecter les uns aux autres. Une telle composition, appelée *package d'assemblage*, regroupe un descripteur et les packages de composants (voir section 3.4.3.3).

3.4.3.3 Processus de développement

Les phases de développement de composants dans le modèle CCM sont :

- **Définition de composants.** Le producteur de composant définit de manière abstraite les composant (type de composants) à l'aide du langage IDL3. IDL3 lui permet aussi de définir les gestionnaires des instances de composants (instances des types définis auparavant).
- **Implémentation de composants.** Le producteur associe l'implémentation du composant en utilisant le langage CIDL. CIDL permet de définir la structure de l'implémentation d'un composant ainsi que certains des aspects non-fonctionnels (persistance, transactions, sécurité, ...). L'utilisation de ce langage est associée à un *framework*, le CIF (*Component Implementation Framework*), qui définit comment les parties fonctionnelles (celles programmées par le producteur) et non-fonctionnelles (décrites et générées à parti de IDL/CIDL) doivent coopérer.

- **Packaging et déploiement.** L'installation d'un composant CCM s'appuie sur l'utilisation de packages de composants, ainsi que sur l'utilisation de descripteurs. Le package d'un composant contient le descripteur du composant et un ensemble de fichiers regroupant l'implémentation du composant. Ces différents éléments sont regroupés physiquement dans un fichier `.ZIP`.
- **Exécution.** Tous les composants sont créés et générés par un conteneur. Un composant ne peut pas exister sans être supporté par un conteneur. Le rôle principal des conteneurs est de masquer et prendre en charge les aspects non-fonctionnels des composants qu'il n'est alors plus nécessaire de programmer au sein des composants (parce que, d'une maniere ou d'autre, il faut bien les programmer). Une fois déployé, un composant est associé à un cycle de vie décomposé en deux phases : la phase de configuration et la phase fonctionnelle. Ces deux phases doivent être réalisées pour toutes les interfaces déclarées dans le composant. La configuration d'un composant repose principalement sur l'affectation de valeurs aux attributs. Lors de la phase fonctionnelle le composant est rendu disponible pour être référencé par des clients.

3.4.4 Autres modèles industriels

Le modèle des JavaBeans [DeM03], développé en parallèle avec la version 1.0 des EJB, est une extension du langage Java définissant un modèle de composant utilisé principalement pour les applications graphiques. Un composant est défini comme un ensemble de propriétés, de méthodes ainsi que d'événements, c'est pourquoi il est considéré aussi comme un objet. Les composants communiquent entre eux à travers des événements dont le mécanisme adhère au paradigme de publication/suscription. Lorsqu'une propriété change de valeur, tous les composants enregistrés sont notifiés du changement.

Openwings [Gen03] est une infrastructure pour le développement de services (*SOP - Service-Oriented Programming*) [KS04]. Dans le cas d'Openwings, le développeur se focalise sur la programmation de composants. Un composant peut être vu comme un programme Java qui fournit un ensemble de services. Un service de composant est le point d'accès pour le développeur aux services de l'infrastructure. Concrètement, un composant fournit et requiert des services à travers des interfaces. Ces services sont mis à disposition par différents systèmes de publication et de recherche tels que la technologie Jini dans le cadre de groupes de travail, les services web (*web services*) dans le contexte d'Internet, etc. Cette infrastructure a été utilisée principalement pour le développement d'applications où la sécurité est importante comme, par exemple, dans les applications militaires.

3.5 Modèles académiques

De nombreux modèles ont été proposés dans le contexte académique. Ces modèles sont définis, en général, pour modéliser différentes propriétés, qui se manifestent lors du développement d'applications à base de composants, telles que le typage de composants, l'analyse du flot de données et de contrôle entre composants ... Afin d'illustrer leur caractéristiques et leurs applications, quatre modèles académiques sont présentés dans cette section : ComponentJ, Java Layers, ArchJava et Fractal.

3.5.1 ComponentJ

ComponentJ [CSC02], défini originalement comme un langage supportant la notion de composant au niveau du langage d'implémentation, se base sur un modèle théorique qui modélise et calcule la composition correcte entre composants [CSC00]. Cette vérification est effectuée en considérant les composants comme des entités typées, dont les types sont exprimés par les interfaces des composants.

3.5.1.1 Composants

Dans ce modèle, les composants sont décrits par les interfaces fournies et requises. Dans le dernier cas, il s'agit de *composants avec dépendances*. Les interfaces sont décrites comme un ensemble de signatures de méthodes en suivant le format de déclaration d'interfaces Java. La figure 3.16(a) montre la définition d'un composant à l'aide de la construction `component`. Le composant `ToDoList` fournit l'interface `IQueue` à travers le port `q`, tandis qu'il requiert les méthodes énumérées dans l'interface `IList` associée au port `list`. Ici, un port peut se voir comme une instance d'un interface. La définition des deux interfaces est montrée dans la figure 3.16(b). La définition d'un composant inclut aussi l'implémentation des services fournis. Un bloc d'implémentation, spécifié à l'aide du mot-clé `declare`, est lié par les ports des interfaces fournies en utilisant la clause `plug-into`. Dans la figure, le bloc `mq` est associé au port `q` par la clause `plug mq into q`. Notez que, dans le cas d'un composant avec dépendances, l'implémentation utilise les méthodes passées par les ports requis.

(a) Composant primitif `ToDoList` avec dépendances.

```
component ToDoList {
  provides IQueue q;
  requires IList list;
  declare mq {
    void enQueue(ITask t) {
      ...
      list.insert(...);
      ...
    }
    ITask deQueue(){...}
  }
  plug mq into q;
}
```

(b) Interfaces de ports.

```
port interface IQueue {
  void enQueue(ITask t);
  ITask deQueue();
}

port interface IList {
  void insert(int p, Object o);
  void remove(int p);
  Object get(int p);
  int size();
}
```

Fig. 3.16 – ComponentJ : un composant et ses interfaces.

3.5.1.2 Composition

ComponentJ permet la définition d'un nouveau composant par la composition de plusieurs composants. La structure d'un composant, soit primitif, soit composé, reste transparente pour les clients du composant. La figure 3.17(a) montre le composant `ToDoModular`

construit à partir de la composition du composant `ToDoList` (voir figure 3.16) et du composant `CList`, en fournissant à ce dernier une interface de type `IList` comme le montre la figure 3.17(b). La description du composant `ToDoModular` utilise la clause `intro` pour déterminer l'ensemble des composants à composer. Finalement, la connexion entre les interfaces fournies et requises est exprimée également par la clause `plug-into`. Notez que, c'est un cas particulier, le composant résultant n'inclut pas l'implémentation des interfaces fournies. Cependant, ComponentJ permet de connecter les ports fournis par un sous-composant (*inner component* dans le vocabulaire de ComponentJ) aux ports fournis par le composant englobant associés à une même interface. Par exemple, la clause `plug todo.q into q` représente la délégation du port `q` vers le port `q` du sous-composant `todo`. Il est possible de définir de cette manière un composant composé avec dépendances, ce qui implique que ComponentJ permet la composition incrémentale.

(a) Composant composé `ToDoModular`.

```
component ToDoModular {
  provides IQueue q;
  intro CList list;
  intro ToDoList todo;
  plug list.list into todo.list;
  plug todo.q into q;
}
```

(b) Composant primitif `CList` sans dépendances.

```
component CList {
  provides IList list;
  declares l {
    void insert(int p, Object o) {...};
    void remove(int p) {...};
    Object get(int p) {...};
    int size() {...};
  }
  plug l into list;
}
```

Fig. 3.17 – ComponentJ : Composant composé.

Comme les composants sont considérés comme des entités de premier ordre, les méthodes peuvent retourner des valeurs de type composant. Pour cela, l'ensemble des ports, aussi bien fournis que requis, sont regroupés sous une *interface de composant*. La figure 3.18(a) montre, par exemple, la définition des interfaces de composant `TToDoList` et `TList` qui regroupe respectivement les mêmes interfaces de port fournies et requises du composant `ToDoList` et `CList`. La figure 3.18(b) montre le composant `Factory` qui fournit le port `f` de type `IFactory`. Notez que la méthode `make` retourne une valeur de type `TToDoList`, c'est-à-dire un composant `ToDoList`, qui prend une valeur de type `TList`, c'est-à-dire un composant `CList`. Finalement, l'implémentation de la méthode `make` crée un composant à l'aide de la clause `compose` en définissant une nouvelle composition similaire à celle effectuée par le composant `ToDoModular` de la figure 3.17(a). Cette caractéristique permet la composition dynamique.

3.5.1.3 Processus de développement

Le développement d'applications dans le contexte de ComponetJ se réduit à trois étapes : la construction et la compilation de la part du producteur des composants et l'instanciation des composants de la part du consommateur de composants. La construction de composant est limitée à la description des composants par le langage présenté ci-dessus

(a) Interface de composant `TToDoList`.

```
component interface TToDoList {
  provides IQueue q;
}

component interface TList {
  provides IList list;
}

port interface IFactory {
  TToDoList make(TList l);
}
```

(b) Composant primitif `Factory`.

```
component Factory {
  provides IFactory f;
  declare m {
    TToDoList make(TList l) {
      return compose {
        provides IQueue q;
        intro Clist ll = l;
        declare mm {
          void enQueue(TTask t) {
            ...
            ll.list.insert(...);
            ...
          }
        }
      }
    }
  }
}
```

FIG. 3.18 – ComponentJ : Les composant comme des entités de premier ordre.

où il suffit de décrire les interfaces de ports, les interfaces de composants et le composant avec son implémentation. Ensuite, pour rendre les composants exécutables, dans l'environnement Java, ComponentJ met à disposition un compilateur [Com] qui crée les classes correspondantes. Finalement, un composant peut être instancié soit à partir du code Java, soit en utilisant l'expression `new` dans le langage de description. La figure 3.19(a) montre l'instanciation du composant depuis la classe `Test` à l'aide de la méthode `createIntance` qui prend comme premier paramètre la liste de noms de port requis et comme deuxième paramètre la liste des objets qui implémentent les interfaces des port requis. Afin d'illustrer l'instanciation de composants dans la description des composants nous définissons le composant `FactoryTest` comme le montre la figure 3.19(b).

3.5.2 JAVA LAYERS

Java Layers [Car02] est basé sur GenVoca [BST+94]. GenVoca associe composant à la notion de couche (*layer*) et propose la construction des applications par la composition de couches. Une couche encapsule l'implémentation complète d'une caractéristique (*feature*) où une caractéristique est une spécification de haut niveau qui définit un attribut ou fonctionnalité de l'application (sûre, portable, capable de comprendre plusieurs protocoles de communication ...). Java Layers applique ces idées à Java en s'inspirant de la notion de mixin [BC90].

3.5.2.1 Composants

Java Layers propose une extension du langage Java en incluant des constructions pour exprimer des couches (composants). Concrètement, une couche fournit une ou plusieurs

(a) Instanciation en Java.

```
class List implements IList {
  ...
  void insert(int p, Object o) {...}
  void remove(int p) {...}
  Object get(int p) {...}
  int size() {...}
}

class Test {
  public static void main(Args String[]) {
    Object aCompFactory =
           Factory.createInstance(
                   new String[]{...},
                   new Object[]{...});
    Object newCompToDo =
       ((_provides_f_IFactory)aCompFactory)
                      get_f().make(new List());
    ...
  }
}
```

(b) Instanciation depuis le langage de description.

```
component FactoryTest {
  provides IMain m;
  declares mm {
    void Main(Args String[]) {
      ((new Factory).f.make(new CList))
             .enQueue(...);
    }
  }
  plug mm into main;
}

(new FactoryTest).m.main(...);
```

FIG. 3.19 – ComponentJ : Instanciation de composants.

interfaces Java et requiert une ou plusieurs couches. La figure 3.20(a) montre la couche `TCP` qui fournit les méthodes de l'interface `TransportI` (voir figure 3.20(b)). De la description de la couche `Secure` (voir figure 3.20(c)), on s'aperçoit que la couche `Secure` fournit de même les méthodes de l'interface `TransportI` mais requiert aussi une couche qui implémente l'interface `TransportI`. Une couche qui ne requiert aucune autre couche est appelée couche terminale (*terminal layer*), c'est le cas de la couche `TCP`, tandis qu'une couche qui fournit et requiert la même interface est appelée *couche symétrique* (*symmetric layer*), c'est le cas de la couche `Secure`. Le mot-clé `mixin` indique que le type (couche ou classe) requis sera un supertype du code généré. Cela vient de la notion centrale *mixin* [BC90, SB99] (voir section 3.5.2.2).

3.5.2.2 Composition

Deux couches peuvent être composées si elles sont compatibles par rapport aux interfaces fournies et aux couches requises. Les couches sont composées en définissant des expressions de composition sous la forme d'équations (*type equations*). Une équation est formée par un identificateur dans la partie gauche et une équation qui exprime la composition dans la partie droite. Par exemple, la classe `Comp1` de la figure 3.21(a) est affectée à la composition de la couche `KeepAlive` (implémentant la vérification de l'existence de connectivité) qui est paramétrée par la couche implémentant la sécurité dans les communications `Secure` composée à son tour par la couche associée au protocole de communication `TCP`. Notons qu'une couche accepte comme paramètre soit une couche, soit une classe, comme par exemple la classe `Comp3`. Cela implique que Java Layers permet la composition

(a) Couche terminale `TCP`.

```
layer TCP exports TransportI {
  public int send(byte[] outbuf) {...}
  public int recv(byte[] inbuf) {...}
  public void disconnect() {...}
}
```

(b) Interface `TransportI`.

```
public interface TransportI {
  public int send(byte[] outbuf);
  public int recv(byte[] inbuf);
  public void disconnect();
}
```

(c) Couche symétrique `Secure`.

```
layer Secure<mixin L implements TransportI>
            exports TransportI
{
  public int send(byte[] outbuf) {...}
  public int recv(byte[] inbuf) {...}
  public void disconnect() {...}
}
```

FIG. 3.20 – Java Layer : Définition d'une couche.

incrémentale comme le montre la composition `Comp3`, résultante de la composition de la couche `KeepAlive` et de la composition `Comp2`.

Le sens de la composition résultante, résumée par la figure 3.21(b), correspond à l'ordre exprimé par l'équation.

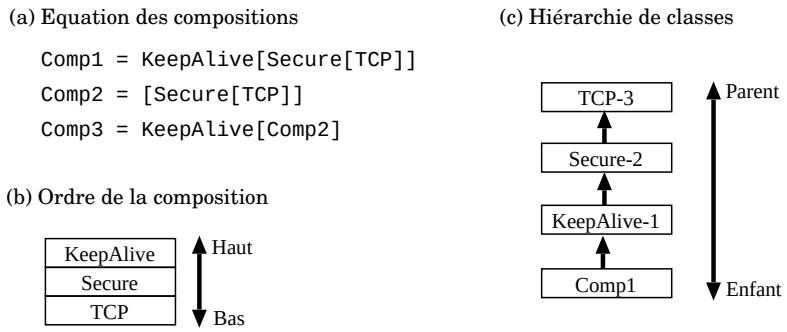


FIG. 3.21 – Java Layer : Composition de couches.

Par contre, la hiérarchie de classes Java générée pour implémenter le résultat de la composition résultante (voir figure 3.21(c)) est structurée dans le sens inverse. De la compilation des couches, une classe et une interface sont créées pour chaque couche. Les classes générées, pour l'exemple, implémentent l'interface `TransportI`. Leur corps est celui spécifié dans la couche correspondante.

Il est fréquent de redéfinir une méthode dans une sous classe en faisant précéder un appel à la définition de la superclasse d'une nouvelle séquence d'instructions. Par exemple, la méthode `send` de la couche `Secure` appelle, après l'encryptage des données, la méthode `send` de la couche `TCP` pour la transmission de données. Java Layers utilise la technique de *mixins* [BC90, SB99] pour l'implémentation des couches. Les mixins, connus aussi par le nom de *sous-types abstraits*, sont des types dont les supertypes sont des paramètres.

La composition d'une couche base (par exemple, `TCP`) et d'une couche plus haute (par exemple, `Secure` se traduit en une composition de type *mixin* où la classe qui implémente la couche base joue le rôle de la superclasse et la classe qui implémente la couche haute celui du mixin. Concrètement, dans l'exemple, la classe associée à la couche `TCP` devient la racine de la hiérarchie obtenue. En plus, si une couche apparaît plusieurs fois dans une équation de composition donnée, chaque référence est exprimée par une implémentation différente, c'est-à-dire qu'une telle composition ne doit pas être interprétée comme étant une composition circulaire.

Pour des raisons d'optimisation, les quatre classes résultantes de la compilation (une pour chaque couche et une pour l'interface) sont *aplaties* en une seule classe (*class flattening*). Cela implique donc que la composition proposée en Java Layers devient fusionnelle au niveau de l'implémentation.

Observons que l'ordre de composition entre les couches `Secure` et `KeepAlive` peut s'inverser. On obtient une application qui encrypte les données d'abord et qui vérifie la connectivité ensuite. Java Layer permet de déterminer des contraintes par rapport aux scénarios de composition dans lesquels une couche peut intervenir. Pour cela, une couche inclut des règles pour contrôler l'existence de couches englobées ou englobantes dans une composition potentielle. Par exemple, la clause `upflow { transport = true }` dans la définition de la couche `TCP` (voir figure 3.22(a)) implique que l'utilisation de cette couche dans une composition quelconque indiquera qu'une couche qui assure les services de transport de données est présente. Par exemple, la définition de la couche `Secure` de la figure 3.22(b) inclut la règle `uptest (exists (transport) && !exists(secure))` qui vérifie l'existence d'une couche de transport et l'absence d'une couche de sécurité des transmissions.

(a) Couche terminale `TCP`.

```
layer TCP exports TransportI
 upflow { transport = true }
{
  public int send(byte[] outbuf) {...}
  public int recv(byte[] inbuf) {...}
  public void disconnect() {...}
}
```

(c) Couche symétrique `Secure`.

```
layer Secure<mixin L implements TransportI>
            export TransportI
 untest (exists(transport) && !exists(secure))
 upflow { secure = true; }
 {
   public int send(byte[] outbuf) {...}
   public int recv(byte[] inbuf) {...}
   public void disconnect() {...}
 }
```

FIG. 3.22 – Java Layer : Contraintes de Composition.

3.5.2.3 Processus de développement

Dans le processus de développement de Java Layers, nous trouvons les phases de définition, de composition et de compilation de couches. Comme résumé par la figure 3.23, lors de la définition des couches, il peut y avoir besoin de définir des classes, en plus des interfaces.

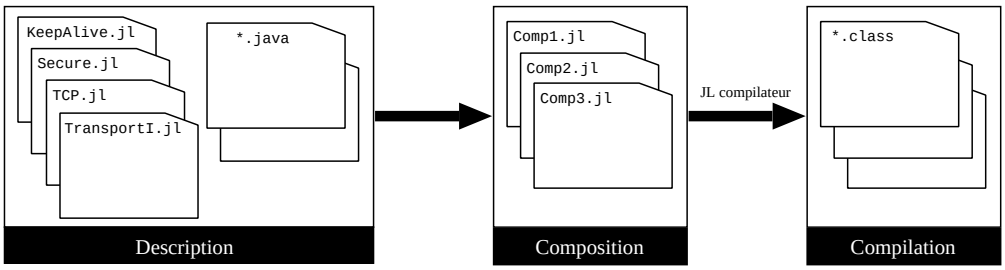


FIG. 3.23 – Java Layer : Processus de développement.

Une fois fournies les équations qui expriment les compositions souhaitées, lors de la phase de composition, le producteur génère, à l'aide d'un compilateur [Jav], les classes et les interfaces Java qui représentent l'application exécutable.

3.5.3 FRACTAL

Fractal [Cou02] est un modèle de composants conçu principalement pour permettre la reconfiguration d'une architecture, c'est-à-dire qu'un composant peut être connecté (ou déconnecté) lorsque l'application est en train de s'exécuter. Les principales caractéristiques de Fractal sont la dynamicité, la réflexivité et l'extensibilité.

3.5.3.1 Composants

Un composant est formé par deux parties : un *contrôleur* et un *contenu*. Le contenu d'un composant est composé d'un ensemble de composants, appelés *sous-composants*, qui sont sous le contrôle du contrôleur associé. Les sous-composants peuvent être aussi des composants composés. Ce processus récursif se termine en retrouvant des *composants primitifs* qui sont des composants encapsulant des classes Java. Un composant primitif se caractérise par un contenu vide. Un composant interagit à travers les interfaces exposées par le contrôleur. Il existe deux catégories d'interfaces : les interfaces fonctionnelles et les interfaces du contrôleur. Il existe deux types d'interfaces fonctionnelles : les *interfaces client* et les *interfaces serveur* qui représentent respectivement les services requis et fournis du composant. Les interfaces du contrôleur sont utilisées par le contrôleur pour gérer l'interaction du composant avec les autres composants : l'interface `LifeCycleController` permet au contrôleur de notifier au composant quand il est activé ou désactivé, l'interface `UserBindingController` doit être défini dans le cas où le composant inclut une interface client et l'interface `AttributeController` implémente l'accès aux attributs du composant.

La figure 3.24(c) montre le composant primitif `Server` qui fournit l'interface `s` de type `Service` incluant une méthode pour imprimer un message dans la console. Le composant peut être paramétré avec l'en-tête du message (`header`) et le nombre d'impressions du message (`count`). L'implémentation du composant est formée par les interfaces fonctionnelles et les classes qui implémentent les interfaces. La figure 3.24(a) montre la définition de l'interface `Service` exprimée par une interface Java. En raison des attributs associés au composant, il doit fournir une interface qui étend l'interface `AttributeController`.

La figure 3.24(b) montre la description de l'interface `ServiceAttributes`. Finalement, l'implémentation du composant (voir figure 3.24(d)) est une classe qui doit implémenter toutes les interfaces exposées par le composant, dans ce cas, les interfaces `Service` et `ServiceAttributes`.

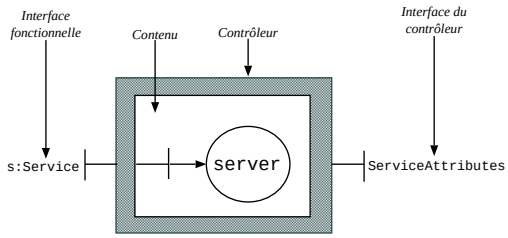
(a) Interface `Service`.

```
public interface Service {
  void print (String msg);
}
```

(b) Interface `ServiceAttributes`

```
public interface ServiceAttributes
    extends AttributeController {
  String getHeader ();
  void setHeader (String header);
  int getCount ();
  void setCount (int count);
}
```

(c) Architecture du composant `server`.



(d) Implémentation du composant.

```
public class ServerImpl
  implements Service, ServiceAttributes {
  private String header = "";
  private int count = 0;
  public void print (final String msg) {
    for (int i=0; i <  count; ++i) {
      System.out.println(header + msg);
    }
  }
  public String getHeader() {
    return header;
  }
  public void setHeader(final String header) {
   this.header = header;
  }
  public int getCount() {
    return count;
  }
  public void setCount(final int count) {
    this.count = count;
  }
}
```

FIG. 3.24 – Fractal : Définition du composant `Server`.

Afin de faciliter le développement d'applications en Fractal, le langage de description d'architecture Fractal ADL a été proposé. Ce langage, basé sur la technologie XML, permet la description des interfaces des composants et supporte la liaison de cette description avec l'implémentation visée. La figure 3.25(a) montre la description du composant `Server` qui utilise la balise `component-type`. Les interfaces du contrôleur ne sont pas incluses dans la définition du type de composant. Afin de spécifier l'implémentation associée au composant on utilise la balise `primitive-template` comme illustré par la figure 3.25(b). Dans ce cas, l'implémentation choisie pour le composant `ServerType`, définie précédemment, est assurée par la classe `ServerImpl` comme indiqué par la balise `primitive-content`. Cette description doit aussi spécifier les interfaces utilisées du contrôleur (ici, l'interface `ServiceAttributes`).

3.5.3.2 Composition

La composition en Fractal est réalisée grâce aux *liaisons* (*bindings*). Une liaison est une connexion entre deux ou plusieurs composants. Une liaison relie une interface de services requis avec une interface de services fournis. Un composant peut être partagé par plusieurs

(a) Composant `Server`.

```
<component-type name="ServerType">
  <provides>
    <interface-type
        name="s"
        signature="Service"/>
  </provides>
</component-type>
```

(b) Liaison de l'implémentation.

```
<primitive-template
  name="ServerImpl"
  implements="ServerType">
  <primitive-content class="ServerImpl"/>
  <controller>
    <attributes signature="ServiceAttributes">
     <attribute name="Header" value=" "/>
     <attribute name="Count" value="1"/>
    </attributes>
  </controller>
</primitive-template>
```

FIG. 3.25 – Fractal : Langage Fractal ADL.

composants composés, c'est-à-dire que plusieurs contrôleurs prennent en charge le même composant. La figure 3.26 montre l'architecture résultant de la composition des composants `Server` et `Client`, représentée par la définition d'un nouveau composant `ClientServer`.

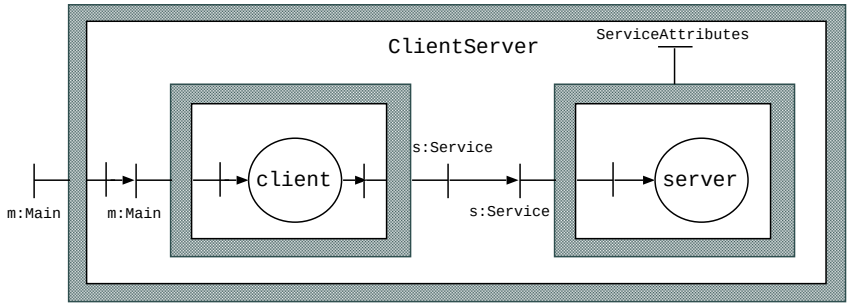


FIG. 3.26 – Fractal : Composant `ClientServer`.

La définition du composant `Client` est similaire au composant `Server`. La figure 3.27(a) et (b) montre la description du composant `Client`. La description du composant composé `ClienServer` (voir figure 3.26) fournit l'interface `ClientServerType` et inclut les sous-composants (i.e. `component-type`) mentionnés par la balise `components` comme le montre la figure 3.27(c). Finalement, la connexion des composants est effectuée par la liaison des interfaces en utilisant la balise `bindings`.

3.5.3.3 Processus de développement

Les phases de développement se réduisent à la création de composants, en fournissant la description des interfaces et des classes d'implémentation unifiées par la description ADL, et la phase de déploiement de composants, où on compile et instancie les composants. Le cycle de vie des composants est en grande partie géré lors de l'exécution par l'API fournie par l'infrastructure de Fractal. Il est possible de désactiver un composant pour le modifier en ajoutant ou enlevant des services, puis de le réactiver. Un composant peut avoir de

(a) Composant `Client`.

```
<component-type name="ClientType">
  <provides>
    <interface-type
        name="m" signature="Main"/>
  </provides>
  <requires>
    <interface-type
        name="s" signature="Service"/>
  </requires>
</component-type>
```

(b) Liaison de l'implémentation.

```
<primitive-template
  name="ClientImpl"
  implements="ClientType">
  <primitive-content class="ClientImpl"/>
</primitive-template>
```

(c) Composant `ClientServer`.

```
<composite-template
 name="ClientServerType">
 <composite-content>
  <components>
   <component name="client" type="ClientType"/>
   <component name="server" type="ServerType"/>
  </components>
  <bindings>
   <binding client="this.m" server="client.m"/>
   <binding client="client.s" server="sever.s"/>
  </bindings>
 </composite-content>
</composite-template>
```

FIG. 3.27 – Fractal : Description du composant `ClientServer`.

nombreuses interfaces différentes au cours de sa vie.

3.5.4 ARCHJAVA

ArchJava [ACN02b, ACN02a] étend le langage de programmation Java pour unifier la description de l'architecture d'une application et son implémentation. ArchJava a comme but principal d'assurer l'*intégrité des communications* (*communication integrity*) [LV95, MQR95] des composants, c'est-à-dire que les communications s'effectuent bien, au niveau de l'implémentation, au travers des connexions définies par l'architecture (absence de connexions *cachées*). Afin de pouvoir décrire des architectures, le développeur utilise de nouvelles constructions telles que des *composants*, des *ports* et des *connexions*.

3.5.4.1 Composants

Un composant est un type spécial d'objet qui communique avec d'autres composants en fonction de l'architecture définie à la composition. Un composant est défini à l'aide la construction `component class` comme illustré par la définition du composant `Parser` de la figure 3.28(a). Dans le contexte d'ArchJava, un port, représenté par le mot-clé `port` plus un identificateur, définit l'extrémité d'un canal de communication entre composants. Un port contient un ensemble de méthodes associées à un des mot-clés suivants : `requires`, `provides` et `broadcasts`. Une méthode annotée comme `provides` est implémentée par le composant et mise à disposition pour les composants connectés à ce port. Au contraire, une méthode annotée comme `requires` doit être implémentée par le composant connecté au port correspondant. Finalement, une méthode `broadcasts` est un type de méthode `requires` sauf qu'elle peut être connectée plusieurs fois et doit retour-

ner `void`. Les méthodes fournies peuvent être implémentées par des appels aux méthodes requises.

(a) Composant `Parser`.

```
public component class Parser {
 public port in {
  provides void setInfo(Token symbol,
                        SymTabEntry e);
  requires Token nextToken()
                  throws ScanException;
 }
 public port out {
  provides SymTabEntry getInfo(Token t);
  requires void compile(AST ast);
 }
 void parse(String file) {
  Token tok = in.nextToken();
  AST ast = parseFile(tok);
  out.compile(ast);
 }
 AST parseFile(Token lookhead) {...}
 void setInfo(Token t, SymTabEntry e) {...}
 SymTabEntry getInfo(Token t) {...}
 ...
}
```

(b) Composant composé `Compiler`.

```
public component class Compiler {
 private final Scanner scanner = ...;
 private final Parser parser = ...;
 private final CodeGen codegen = ...;

 connect scanner.out, parser.in;
 connect parser.out.codegen.in;

 public static void main(String[] Args) {
  new Compiler().compiler(args)
 }
 public void compile(String[] args) {
  ...
  ... parser.parse(file);...
 }
}
```

FIG. 3.28 – ArchJava : Les composants et la composition.

3.5.4.2 Composition

La composition est exprimée par la définition de composants composés (*composite components*). Un composant composé est construit à partir d'instances de composants, appelées des sous-composants. Par exemple, le composant `Compiler` de la figure 3.28(b) définit l'architecture d'un compilateur en incluant les sous-composants `scanner`, `parser` et `codegen` qui sont des intances du composant `Scanner`, `Parser` et `CodeGen` respectivement. L'exemple montre que l'instance de composant `parser` communique avec les instances de composants `scanner` et `codegen` tandis que la communication entre les instances de composants `scanner` et `codegen` n'est pas permise. Cette communication est rendue possible par la connexion des ports des composants à l'aide de la construction `connect` qui lie les méthodes requises et les méthodes fournies de même signature. Le composant `Compiler`, par exemple, connecte le port `out` du sous-composant `scanner` avec le port `in` du sous-composant `parser`.

Un composant peut être instancié en utilisant la même construction, `new`, que celle utilisée pour l'instanciation des objets. Par contre, le composant instancié conserve une référence de l'instance de composant dont il est un sous-composants, appelé *composant parent* (*parent component*). Les méthodes fournies par un sous-composant peuvent être appelées directement par le composant parent. Cela implique que pour assurer l'intégrité

de la communication, aucune référence sur les sous-composants ne devient disponible en dehors du composant englobant. Pour cette raison, ArchJava interdit l'utilisation des types de composants dans la définition de ports ou d'interfaces.

Cependant, un composant peut être créé et connecté dynamiquement en utilisant un *parton de connexion* (*connection pattern*). Un patron de connexion permet de spécifier statiquement un ensemble de connexions entre composants instanciés dynamiquement. Par exemple, la clause `connect pattern Router.workers, Worker.server` décrit les connexions entre le sous-composant `Router` et des sous-composants `Worker` dynamiquement créés (voir figure 3.29(a)). Une expression de connexion satisfait un patron si les ports connectés sont compatibles et si les instances de sous-composant sont du même type que ceux spécifiés par le patron.

(a) Composant `WebServer`.

```
public component class WebServer {
 private final Router r = new Router();
 connect r.request, create;
 connect pattern Router.workers, Worker.serve;

 private port create {
  provides r.wrokers requestWorker() {
   final Worker newWorker = new Worker();
   r.workers connection
     = connect(r.workers, newWorker.serve);
   return connection;
  }
 }
 public void run() { r.listen();}
}
```

(b) Architecture du composant.

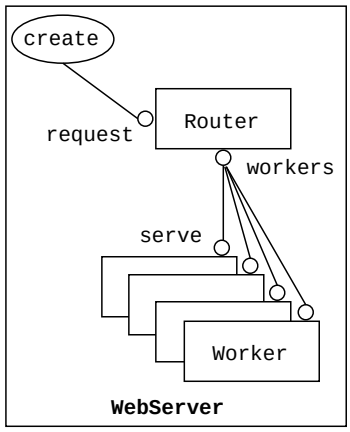


FIG. 3.29 – ArchJava : Patron de connexion.

En ArchJava, un même composant peut intervenir simultanément dans plusieurs connexions à travers un même port. Par exemple, le sous-composant `Router` du composant `WebServer` communique avec des composants de type `Worker` en utilisant différentes connexions (voir figure 3.29(b)). Pour cela, ArchJava inclut la notion d'interface de port (*port interface*) pour décrire un port qui peut être instancié plusieurs fois.

Finalement, ArchJava offre dans sa dernière version la réification des connexions en permettant la définition de connecteurs [ASCN03]. Un connecteur est représenté par une classe qui implémente la vérification lors de la connexion (*connection type-checking*) entre les ports intervenants. Cette approche permet d'effectuer une vérification plus en détail des méthodes des ports, à savoir le type des arguments, le type de retour, ... La classe du connecteur est spécifiée en ajoutant le mot-clé `with` plus le nom de la classe à la fin de la clause `connect`.

3.5.4.3 Processus de développement

Du fait que l'implémentation et l'architecture d'une application sont inséparables, les composants sont traités comme des boîtes blanches. En conséquence, un consommateur a accès à la définition du composant au moment de la réutilisation. En fait, en ArchJava, on distingue deux étapes : la construction et la compilation. Lors de la construction le producteur/consommateur définit l'architecture de l'application en générant des fichiers `.archj` qui sont ensuite fournis au compilateur ArchJava [Arc]. Ce dernier génère, finalement, les classes et les interfaces qui implémentent les composants et connecteurs de l'architecture visée.

3.5.5 Autres modèles académiques

Dans le modèle ACEEL (*Adaptative ComponEnt modEL*) [Che05], un composant est une entité logicielle qui fournit un service donné via une interface séparée de l'implémentation mettant en œuvre le service. Un composant peut être associé à différentes implémentations. L'objectif de ce modèle est de permettre le changement dynamique du comportement d'un composant en fonction des conditions de l'environnement dans lequel le composant est exécuté. Dans ce contexte, un composant est définit en deux niveaux : le *niveau de base*, exprimé par une version évoluée du patron de conception *Strategy* [GHJV94] pour représenter tous les comportements (implémentations) du composant, et le *métaniveau* qui représente la structure d'adaptation du composant surveillant l'environnement.

ACOEL (*A Component-Oriented Extensional Language*) [Sre01, Sre02]Est conçu comme une extension du langage Java (ou C). Il permet de définir des composants est de les composer dynamiquement. Un composant est composé de deux parties : son implémentation définie par un ensemble de classes, de méthodes et de champs, et un ensemble de ports d'entrée et de sortie. ACOEL utilise aussi la technique des *mixins* [BC90] comme un support pour des fonctionnalités applicables à un ensemble de composant. Tout comme les composants, il est possible aussi de composer plusieurs mixins.

CwEP(*Components with Explicit Protocols*) [FS02] est un modèle qui étend la notion d'interface avec des *protocoles d'interaction*. De tels protocoles définissent la disponibilité des services offerts par un composant et spécifie les interactions entre les composants et ses collaborateurs. Dans CwEP, une application est vue comme une hiérarchie plate de composants interagissant entre eux car l'intérêt est focalisé sur le comportement des composants et la composition de ces comportements plutôt que sur la structure des composants et la composition des ces structures.

3.6 Composants vs Modules

Les notions de composant et de module sont souvent utilisées de manière interchangeables car les deux notions permettent de structurer les applications comme un ensemble d'entités connectées entre elles.

Toutefois, une différence significative que l'on peut voir entre les deux notions est la possibilité de distinguer dans le cas des composants les différents éléments de la composition à l'exécution de l'application, c'est-à-dire la possibilité d'identifier un élément de la composition et, éventuellement, de pouvoir la débrancher, le changer, etc. Un composant est une abstraction qui persiste à tout moment du cycle de vie du composant, principalement à l'exécution. Par contre, la composition de modules résulte en une application où l'ensemble des morceaux de code associés aux modules est connecté en éditant les liens correspondants. Finalement, l'application résultante ne se distingue pas d'une autre dont la conception a été réalisée de manière monolithique. Quelques auteurs préfèrent parler de la séparation des étapes de *chargement* (*loading*) et de *liaison* (*linking*) prises en compte dans le contexte des composants. La possibilité de charger (ou décharger) et de lier (ou délier) les composants de manière indépendante n'est pas possible dans le cas de modules. Le code des modules, lié selon la composition souhaitée, est chargé de manière définitive lors de l'exécution de l'application résultante. Concrètement, un composant peut être connecté à d'autres composants, par exemple dans un cadre distribué en utilisant le réseau, tandis que la liaison de modules est confinée à un processus simple et local. Le fait que, en général, les composants contiennent des objets permet d'associer un état persistant à l'exécution, ce qui n'est pas vrai pour un module dont la conception est plutôt basée sur une approche fonctionnelle. Le modèle de modules *Units* [FF98], initialement défini pour le langage MzScheme, a été rendu disponible pour le langage Java et C par les implémentations Jiazzi [MFH01] et Knit [RFS+00] respectivement. Cells [RS02], conçu comme un modèle de conteneur d'objets et de code, est défini comme un modèle hybride disposant de caractéristiques des composants et des modules.

Toutes comme les architectures à base de composants peuvent être décrites à l'aide de langages de description d'architectures, les architectures à base de modules peuvent être décrites à l'aide des *langages d'interconnexion de modules* (*MIL - Module Interconnection Language*) [PDN86]. Dans les deux cas, les entités sont définies à travers d'interfaces explicites, qui exposent les fonctionnalités fournies et/ou requises, servant comme un moyen de communication entre elles. Cependant, ces langages se distinguent par la nature de l'information exprimée. Les ADL permettent de décrire le flot de controle et de données en explicitant les connexions entre les composants, alors que les MIL se limitent à décrire la relation d'utilisation des modules [MT00]. Pour mieux illustrer les différences, considérons une application qui implémente une horloge. Cette application se compose d'un ensemble de compteurs, à savoir un compteur pour gérer les heures, un pour les minutes et un pour les secondes. Ce sont des compteurs similaires qui sont incrémentés modulo 24 ou 60 suivant leur composition dans la composition.

Tout d'abord, la définition du composant `Clock`, telle que présentée dans la figure 3.30, a été développée à l'aide d'ArchJava (voir section 3.5.4). Ce composant est le résultat de la composition de deux types de composant : le composant `DCounter` qui incrémente un autre compteur à chaque fois qu'il atteint 0 et le composant `Counter` qui n'incrémente pas de nouveau compteur. Comme indiqué dans le corps du composant `Clock`, une instance du premier type de composant est utilisée pour gérer les heures et deux instances du deuxième type de composant sont utilisées pour gérer respectivement les minutes et les secondes. En utilisant les clauses `connect`, les sous-composants sont finalement connectés.

(a) Composant primitif `Counter`.

```
component class Counter {
 public port out {
  provides void init();
  provides int getValue();
 }
 public port tick {
  provides void tick();
 }
 private base;
 private value;

 void init(int base){
  this.base = base;
  this.value = 0;
 }
 void tick() {
  value = value + 1;
  if (value > base)
      value = 0;
 }
 int getValue() {
  return this.value;
 }
}
```

(b) Composant primitif `DCounter`.

```
component class DCounter {
 public port out {
  provides void init();
  provides int getValue();
 }
 public port tick {
  provides void tick();
 }
 public port dtick {
  requires void tick();
 }
 ...

 void init(int base){...}
 void tick() {
  value = value + 1;
  if (value > base){
   value = 0;
   dtick.tick();
  }
 }
 int getValue() {...}
}
```

(c) Composant composé `Clock`.

```
component class Clock {
 public port out {
  provides void init();
  provides void showTime();
 }
 private final Counter hh =
         new Counter();
 private final Counter mm =
         new DCounter();
 private final Counter ss =
         new DCounter();

 connect hh.tick to mm.dtick;
 connect mm.tick to ss.dtick;

 void init() {
  this.hh.init(24);
  this.mm.init(60);
  this.ss.init(60);
 }
 void showTime() {
  ss.tick();
  // délai d'un seconde...
  System.out.println(
      hh.getValue()+'h'+
      mm.getValue()+':'+
      ss.getValue());
  this.showTime();
  // ...
 }
}
```

FIG. 3.30 – L'application `Clock` en ArchJava.

Le composant `hours` est connecté par le port `tick` au port `dTick` du sous-composant `minutes`. La même connexion est effectuée entre le compteur des minutes (i.e. `minutes`) et celui des secondes (i.e. `seconds`).

La figure 3.31 montre comment la même application peut être implémentée à l'aide d'un MIL. Ici, la syntaxe du MIL appelé INTERCOL [Tic79] a été utilisée. Dans ce contexte, une description, ou système (`System`), est une séquence de définitions de modules (`module`) ainsi qu'un ensemble de compositions (`COMPOSITION`). Un module spécifie une interface comme un ensemble d'entités, appelées ici des *ressources*, qui sont définies par un module et utilisées par un autre. Les interfaces incluent les ressources requises (`require`) et fournies (`provide`). En particulier dans INTERCOL, la description d'un module représente une famille d'implémentations (`implementation`), pour différentes plates-formes et langages, qui respectent une même interface. Finalement, une composition ne tient compte que des implémentations possibles spécifiées dans la description fournie. De ce fait, on dit qu'une

composition en INTERCOL représente un membre d'une famille d'applications. Observons que le langage de description ne permet pas de spécifier les instances des modules utilisés dans l'implémentation résultante (voir figure 3.32). Il n'y a pas la notion d'instance de module dans INTERCOL. En fait, la définition des modules a dû être modifiée, par rapport à la conception à base de composants, en tenant compte du fait que les modules ne sont pas des entités à état persistant.

```
System Clock
 provide clock_proc
 module counter
  provide package counter_proc
   type base,value:INTEGER
   procedure tick(base,value)
  end counter_proc
  implementation COUNTER01..Pascal
  implementation COUNTER02..ML
 end counter

 module time
  provide package time_proc
   type hh, mm, ss :INTEGER
   function timeToString(hh,mm,ss)
            return STRING
  end time_proc
  implementation TIME..Pascal
 end time

 module main
  provide procedure clock_proc
    procedure showTime
  require counter_proc, time_proc
  implementation CLOCK01..Pascal
 end main
 COMPOSITION
 clock_a = [CLOCK01,COUNTER01,TIME]
end Clock
```

FIG. 3.31 – L'application `Clock` dans le MIL INTERCOL.

L'application à base de composants décrit de manière explicite l'existence de trois instances d'un composant qui sont composées dans un nouveau composant, alors que la version à base de modules ne considère que la relation d'utilisation entre les modules. Les figures 3.33(a), une représentation graphique d'une composition ArchJava, et 3.33(b), un arbre de description des familles de systèmes dans INTERCOL, illustrent les différences mentionnées ci-dessus.

D'un point de vue de plus bas niveau, nous disons que l'existence des modules disparaît une fois l'application finale créée. Autrement dit, il n'est plus possible de distinguer les composants, ni statiquement ni dynamiquement, après l'édition de liens.

3.7 Bilan

Dans cette section nous avons présenté un ensemble représentatif de modèles de composants aussi bien dans le milieu industriel que dans le milieu académique. Malgré la diversité existante, il a été constaté que la plupart des modèles partagent les mêmes caractéristiques telles que la notion de boîte noire, la composition et l'intégration de ces notions au processus de développement.

(a) Implementation du module COUNTER01

```
unit COUNTER01;
interface
 procedure tick(base: integer,
                var value: integer);
implementation
 procedure tick;
  begin
   if (base == value) value:=0
   else value:= value + 1;
  end;
end.
```

(b) Implementation du module TIME

```
unit TIME;
interface
 function timeToString(hh, mm, ss:integer)
          :string;
implementation
 function timeToString;
 begin
  timeToString:= hh +'h'  +
                 mm + ':' +
                 'ss';
 end;
end;
```

(c) Implementation du programme CLOCK01.

```
program CLOCK01;
 use COUNTER01, TIME;
 procedure showTime;
  var hh, mm, ss, valhh: integer;
      valmm, valss : integer;
  begin
   hh:= 24; mm, ss:= 60;
   for h:= 0 to hh-1 do
    begin
     for m:= 0 to mm-1 do
      begin
       for s:= 0 to ss-1 do
        begin
         write(timeToString(valhh,valmm,valss));
         tick(ss,s);
        end;
       tick(mm,m);
      end;
     tick(hh,h);
    end;
  end;
begin
 showTime;
end.
```

FIG. 3.32 – Implémentation en Pascal de l'application Clock.

Les modèles industriels, EJB, CCM et COM, ont présenté un intérêt commun pour les mêmes préoccupations, à savoir les services techniques de persistance, de sécurité et de transactions. Ces solutions, par contre, ne répondent pas de manière claire aux problèmes de la création d'un nouveau composant par composition/assemblage d'autres composants, du raisonnement sur ces composant, vues comme des architectures, ainsi que la gestion dynamique de ces compositions. En ce qui concerne les modèles académiques, l'effort s'est porté sur l'expressivité de la description des composants et de leurs connexions.

Globalement, concentrés sur le besoin du développement à grande échelle, les modèles de composants n'ont pas mis en question l'efficacité des applications résultantes. Même si quelques modèles permettent la reconfiguration dynamique, le processus d'adaptation n'intervient qu'au niveau des interfaces alors que leur implémentation reste inchangée. Les possibilités d'adaptation des composants dans les modèles de composant existants sont superficielles. En conséquence, les applications résultantes conservent le degré de généralité initial des composants constituants.

En effet, il existe de nombreux domaines d'application où la conception à base de composant produit des applications notablement inefficaces. Le développement à base de composants étant applicable à tous les domaines requiert une autre conception de l'implémentation des composants pour ces domaines [Bat98].

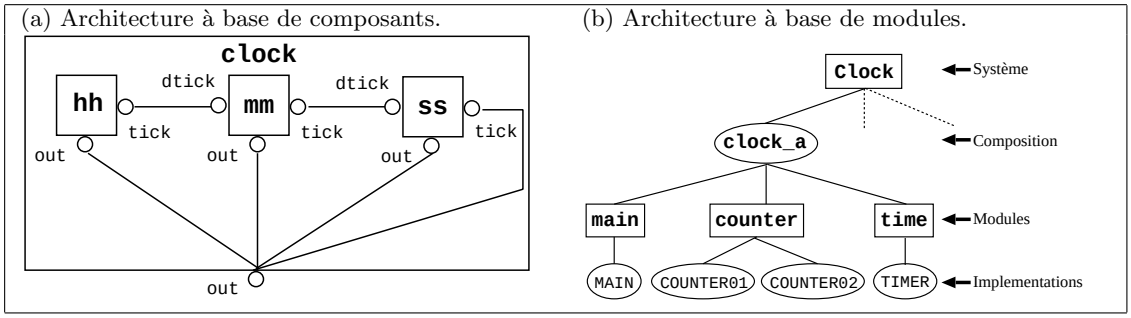


FIG. 3.33 – Composants vs Modules : Architecture résultante.

Chapitre 4

Spécialisation modulaire

Sommaire

4.1 Évaluation partielle sensible aux modules **83**
4.1.1 Analyse symbolique de temps de liaison 84
4.1.2 Extension génératrices polymorphiques 86
4.1.3 Spécialisation de modules 87
4.2 Scénarios de spécialisation **88**
4.2.1 Définition de scénarios 88
4.2.2 Sous-spécialisation et sur-spécialisation 89
4.2.3 Compilation et application des modules 90
4.3 Analyse globale et séparée de programmes **91**
4.4 Bilan **92**

Comme nous verrons dans la partie de la contribution de la présente thèse, nous nous proposons appliquer des techniques de spécialisation de programmes pour la spécialisation de composants. Pour cette raison, nous abordons dans ce chapitre différents approches qui ont permit de factoriser le processus d'analyse en prenant en compte l'architecture de programmes structurés au tour d'abstractions permettant leur *modularisation* (par exemple, modules, classes, foncteurs, bibliothèques). En effet, une approche monolithique de l'analyse de programmes est devenue irréaliste.

4.1 Évaluation partielle sensible aux modules

L'évaluateur partielle *Tempo* [CHN+96] a été développé pour la spécialisation de programmes "*système*" en C. À cause de la structure et la taille des programmes de ce type, l'approche d'analyse de temps de liaison dans Tempo considère l'isolation d'un morceau de code (procédure) à spécialiser. Cette approche, appelée *spécialisation par modules* (*module-oriented specialization*), prend en compte le temps de liaison du contexte du module. Cette information permet à l'évaluateur partiel de raisonner, par exemple, sur les variables globales en termes d'effets de bord. En effet, l'analyse de temps de liaison est basée sur une

analyse d'alias, en plus de l'analyse d'effets de bord, afin d'enrichir l'information externe au module.

Avec la programmation à base de modules, chaque module est d'abord compilé de manière séparée, et ensuite lié à autres modules sous la forme d'un programme exécutable (voir section 3.6). Sur la même idée, il est possible de préparer chaque module pour une spécialisation ultérieure comme proposé par Dussart *et al.* [DHH97], qui convertissent chaque module (implémenté à l'aide d'un sous-ensemble du langage fonctionnel Haskell) en une extension génératrice. Cette forme de spécialisation de programmes sensible à la structure du programme (*module-sensitive program specialisation*) a été appliquée pour faire face aux problèmes suivants :

- Coût de la spécialisation : étant donné que l'entrée est constituée par le code source du programme visé plus les bibliothèques des fonctions utilisées, le spécialiseur peut prendre un temps excessif à générer les programmes spécialisés. Par contre, la spécialisation d'un module qui a été *prétraité* préalablement sous la forme d'une extension génératrice est plus performante que celle du module original. En plus, sous l'hypothèse que les bibliothèques ont été aussi traitées de la même manière, le coût de la spécialisation d'un programme peut être substantiellement réduit.
- Disponibilité du code : comme le code source des librairies devient disponible au moment de la spécialisation du programme qui les utilise, l'approche monolithique s'avère irréaliste dans le cas de bibliothèques conçues comme des produits commerciaux. La spécialisation basée sur l'utilisation d'extensions génératrices, au contraire, n'est pas dépendante de la disponibilité du code source.
- Coût de l'édition de liens : le coût de la compilation du programme résultant de la spécialisation, sous sa forme monolithique, est plus important que celui de l'édition de liens des modules déjà spécialisés (et compilés) séparément.

Rappelons que l'exécution d'une extension génératrice pour la génération de la fonction spécialisée est plus performante que l'utilisation du spécialiseur, qui doit encore inspecter et interpréter le code source correspondant.

4.1.1 Analyse symbolique de temps de liaison

Comme mentionné dans la section 2.1.3, la construction de l'extension génératrice se base sur les annotations de temps de liaison du programme à spécialiser. Du point de vue pratique, une analyse de temps de liaison classique où les constructions sont annotées comme statique ou dynamique en fonction des paramètres n'est pas applicable car l'information sur le contexte d'utilisation de la fonction n'est pas disponible au moment de l'analyse. Pour cette raison, la technique d'*analyse symbolique de temps de liaison* [HM94] a été utilisée. Cette approche permet de calculer le temps de liaison d'un programme en fonction des *variables* de temps de liaison au lieu des valeurs proprement dites. Cela est possible car les calculs faits sur les temps de liaision des constructions sont, en essence, les mêmes. L'analyse symbolique peut être vue comme une manière de factoriser l'analyse en deux parties : une partie symbolique qui ne dépend d'aucun contexte déterminé et, en conséquence, l'annotation résultante peut être réutilisée, et la spécialisation proprement dite qui dépend des valeurs concrètes de temps de liaison. Le fait que le résultat

(a) Definition de la fonction `Power`.

```
power n x =
  if n = 1
     then x
     else x *
          power (n-1) x
```

(b) Annotation de temps de liaison.

```
power {t u} n x =^t
  if n =^t [S → t]1
     then [u → (t ⊔ u)]x
     else [u → (t ⊔ u)]x *^(t⊔u)
          power {t u} (n-^t [S → t]1) x
```

FIG. 4.1 – Analyse symbolique de temps de liaison.

de la partie indépendante de l'analyse soit un programme annoté avec des annotations symboliques, programme qui peut être spécialisé par rapport aux valeurs concrètes, rend possible une analyse polyvariante sans l'identification de toutes les valeurs possibles pour les paramètres d'entrée [Bul93].

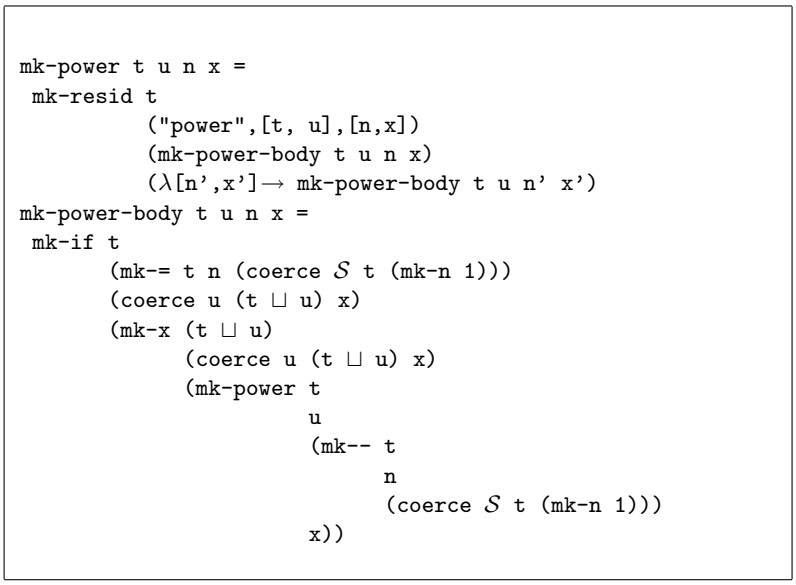
Dans ce contexte, chaque expression est associée à un *type de temps de liaison*. Le type d'une expression simple peut être la valeur statique ($\mathcal{S}$), la valeur dynamique ($\mathcal{D}$), une variable ou le plus petit majorant de deux annotations. Par contre, dans le cas des fonctions anonymes (c'est-à-dire, des λ-expressions), le temps de liaison est exprimé sous la forme $\alpha \rightarrow^b \beta$ [DHH97].

Pour illustrer cette approche prenons la fonction `power` de la figure 4.1(a). Cette fonction peut être annotée, par exemple, comme le montre la figure 4.1(b). Ici, t et u sont les temps de liaison des paramètres `n` et `s`, respectivement. L'annotation sur le symbole égal de la fonction indique quand la fonction doit être complètement dépliée ou spécialisée afin de créer une version résiduelle. L'annotation $[\alpha \rightarrow \beta]_e$ représente une *coercition de temps de liaison*. Elle permet de convertir le temps de liaison de l'expression e d'un type α à un type β, ce qui peut être vu comme une généralisation de l'opérateur *lift* (voir section 2.1.2.1). Par exemple, le temps de liaison de l'expression du test `n = 1` est calculé comme le plus petit majorant entre la valeur statique (i.e. le temps de liaison de la constante `1`) et celui de la variable `n`. L'annotation $[\mathcal{S} \rightarrow t]$`1` permet de rendre dynamique (*lifting*) la constante `1` quand le test doit être résidualisé car t est le temps de liaison associé au test. Par exemple, si t est statique, c'est-à-dire que `n` est statique, alors l'appel de la fonction peut être déplié. Par exemple, pour `n` égal à 3 l'appel est remplacé par *x * (x * x)*. Au contraire, si t est dynamique, la fonction va être residualisée car le test de l'expression conditionnelle n'est pas évalué à la spécialisation.

4.1.1.1 Inférence de type de temps de liaison

Dussart et al. [DHM95] ont défini un système de types afin de vérifier les annotations sur un programme dont les fonctions sont affectées à des types de temps de liaison qui se comportent de manière polymorphique. Par exemple, la fonction `power` peut être typée par $\forall t, u.t \rightarrow u \rightarrow t \sqcup u$. Chaque fonction a un type principal à partir duquel il est possible de dériver d'autres types instanciés. L'algorithme utilisé pour le calcul des types principaux peut être aussi utilisé pour l'inférence des annotations sur le programme.

Dans ce contexte, l'inférence de temps de liaison sur un module requiert le type associé aux fonctions importées mais, par contre, il n'est pas nécessaire de connaître l'information



```
mk-power t u n x =
 mk-resid t
          ("power",[t, u],[n,x])
          (mk-power-body t u n x)
          (λ[n',x']→ mk-power-body t u n' x')
mk-power-body t u n x =
 mk-if t
       (mk-= t n (coerce 𝒮 t (mk-n 1)))
       (coerce u (t ⊔ u) x)
       (mk-x (t ⊔ u)
             (coerce u (t ⊔ u) x)
             (mk-power t
                       u
                       (mk-- t
                             n
                             (coerce 𝒮 t (mk-n 1)))
                       x))
```

FIG. 4.2 – Extension génératrice de la fonction `Power`.

sur le contexte d'utilisation ultérieur du module analysé. Une fois le module analysé, les types de temps de liaison associés aux fonctions exportées sont exposés sous la forme d'un fichier externe à la définition du module (*binding-time interface file*). L'analyse d'un module qui importe un module donné utilise ce fichier afin d'analyser les fonctions importées. Cela implique que l'analyse doit suivre un ordre déterminé par l'arbre de dépendances entre modules et, par conséquent, dans cette approche la dépendance cyclique entre modules n'est pas permise.

4.1.2 Extension génératrices polymorphiques

Dans le scénario classique, l'extension génératrice d'une fonction est créée en exploitant l'annotation de temps de liaison associée. Dans le cas de temps de liaison symbolique, cette annotation de temps de liaison n'est pas totalement connue lors de la construction de l'extension génératrice. C'est d'une certaine manière l'extension génératrice elle-même qui va terminer son calcule lors de la spécialisation. Concrètement, il n'est pas possible de se servir des valeurs de temps de liaison pour simplifier la construction des extensions génératrices. En conséquence, l'extension génératrice doit conserver dans sa définition les paramètres de temps de liaison (voir figure 4.1(b)), lesquels sont utilisés dans le calcul des temps de liaison. La figure 4.2 montre l'extension génératrice de la fonction `power`.

La fonction `mk-power` prend comme paramètres, les paramètres de temps de liaison (i.e. `t` et `u`) ainsi que les paramètres de la fonction `power` (i.e. `n` et `x`). Ici, les opérations `op` dans le code source sont modélisées par la fonction `mk-op` avec un paramètre supplémentaire qui détermine si l'opération est statique ou dynamique. La fonction `mk-power-body` permet de générer la version spécialisée, dont la définition est similaire à celle de la version annotée de

(a) Programme original

```
module A where
    f x =^D ...
module B where
    import A
    g y =^S f (y +^D 1)
module C where
    import B
    h z =^D g(2 *^D z)
```

(b) Programme spécialisé

```
module A where
    f x = ...
module B where
    import A
    g y = f (y + 1)
module C where
    import B
    import A
    h z = g(2 * z)
    h z = f(2 * z + 1)
```

FIG. 4.3 – Spécialisation de modules.

la fonction `power` de la figure 4.1(b). La fonction `mk-power` produit des versions spécialisées d'appels à la fonction `power`, laquelle peut être complètement dépliée ou residualisée. Cette décision est prise par l'application de la fonction `mk-resid` à travers le paramètre `t`, lequel permet, par exemple, de déplier le corps de la fonction si la valeur affectée est statique. Les autres paramètres de la fonction `mk-resid` sont : un triplet utilisé pour identifier la version spécialisée de la fonction au cas où le paramètre `t` est dynamique, le résultat du dépliage de l'appel de la fonction dans le cas où le paramètre `t` est statique et, finalement, une fonction pour créer le corps de la version spécialisée de la fonction si le paramètre `t` est dynamique.

Notez que, à cause de l'annotation symbolique, le programme générateur des extensions génératrices *cogen* (voir section 2.1.4.3), qui doit aussi *résidualiser* les variables de temps de liaison, a l'impossibilité de raisonner sur des valeurs concrètes de temps de liaison.

4.1.3 Spécialisation de modules

Afin de conserver la factorisation aussi à la spécialisation, le programme spécialisé doit conserver la même structure modulaire que le programme original. Une solution simple peut être de réunir les versions spécialisées d'une fonction dans le même module que la fonction originale. Cela modifie par contre les relations de dépendance entre modules, plus précisément la relation d'importation dans le cas des travaux considérés ici. Par exemple, considérons le programme partiellement annoté de la figure 4.3 [DHH97].

Notons que, comme la définition de la fonction `g` est annotée comme statique, l'appel à la fonction `g` peut être dépliée. Ainsi, l'appel à `h` peut être déplié dans le corps de `h` dont `h z = f(2 x z + 1)` est la version spécialisée résultante du dépliage, incluse dans la définition du module `C`. Par contre, comme la définition originale du module `C` n'importe pas le module `A`, la fonction `f` ne peut pas être réutilisée dans la définition de la fonction `h`. Il est possible de garantir l'accès aux définitions des fonctions utilisées en analysant le code résultant de la spécialisation. Du fait de la hiérarchie modulaire existante, la relation acyclique entre les modules est conservée.

4.2 Scénarios de spécialisation

Comme nous l'avons vu dans la section 2.1, un évaluateur partiel prend comme entrée un programme et l'information du temps de liaison des paramètres. Dans ce cas, l'évaluateur est utilisé comme une boîte noire, pour laquelle l'utilisateur n'a aucune possibilité de spécifier des opportunités de spécialisation concernant la structure du programme. Un solution à ce problème d'expressivité a été apportée par Le Meur *et al.* [LMLC04] dont la principale contribution est un langage qui permet à l'utilisateur de l'évaluateur partiel de décrire les opportunités de spécialisation en termes de *scénarios de spécialisation*. Un scénario de spécialisation est une déclaration sur une fonction, une variable globale ou une structure de données susceptible d'une spécialisation sur lesquelles on spécifie le contexte de spécialisation en termes de temps de liaison. Il s'agit d'un langage de modules qui permet à l'utilisateur d'identifier les fragments de code ou les structures de données qui doivent être spécialisés. Les scénarios concernant une même fonctionnalité du programme peuvent être reliés sous la forme d'un *module de spécialisation*. Le langage de spécification des scénarios de spécialisation, étant un langage de haut niveau, il permet d'exprimer des concepts dans le domaine de la spécialisation par évaluation partielle de manière simple et intuitive. Ce langage a été utilisé pour la spécialisation de programmes C à l'aide de l'évaluateur partiel Tempo [CHL+98].

4.2.1 Définition de scénarios

Afin d'illustrer comment il est possible de spécifier des opportunités de spécialisation, nous allons prendre l'exemple de la fonction `dot` de la figure 4.4(a). Cette fonction implémente la multiplication de deux vecteurs de valeurs entières. Elle vérifie, tout d'abord, leur taille et, ensuite, elle réalise le calcul correspondant. Considérons, par exemple, la spécialisation de cette fonction par rapport à un vecteur `u` entièrement statique. Pour cela, l'information minimale nécessaire à l'évaluateur sera le nom de la fonction et le fait que l'entrée `u` est un vecteur statique. Cependant, cette forme de spécialisation s'avère insuffisante pour guider le processus de spécialisation en ce qui concerne la structure du programme. Par exemple, l'utilisateur pourrait empêcher l'évaluateur de spécialiser la fonction `error`. La figure 4.4(b) montre un ensemble de déclarations de scénarios de spécialisation réunies dans le module de spécialisation `vector`.

Les déclarations `VecDS` et `VecSS` sont des scénarios de spécialisation sur la structure de donnée `Vec`. Dans le premier cas, le scénario indique que le champ `data`, représentant les données du vecteur, doit être considéré comme dynamique et que le champ `length`, représentant la taille du vecteur, doit être considéré comme statique.

Les déclarations `Btdot1` et `Btdot2`, par contre, sont des scénarios de spécialisation sur la fonction `dot`. Le scénario `Btdot1` détermine que le code de la fonction est disponible pour la spécialisation, c'est qui est indiqué par le mot clé `intern`. De plus, il permet de spécifier que des opportunités de spécialisation peuvent être exploitées quand les pointeur sur les vecteurs `u` et `v` sont tous les deux statiques, ce qui est indiqué par l'expression `S(*)`, et quand les vecteurs satisfont les conditions imposées par le scénario `VecDS`. Le scénario `Bterr`, déclaré `extern`, indique que la fonction `error` ne doit pas être spécialiseé. Une fois

(a) Définition originale.

```
struct vec {
  int *data;
  int length;
};

int dot(struct vec *u, struct vec *v) {
  int i = 0;
  int sum = 0;
  if (u->length != v->length)
    error();
  for(i = 0; i < u->lenght; i++)
    sum += u->data[i] * v->data[i];
  return sum;
}
```

(b) Scénarios de spécialisation.

```
Module vector {
 ...
 Defines {
  From dotproduct.c {
   VecDS::struct vec
          {D(int *) data; S(int) length;};
   VecSS::struct vec
          {S(int *) data; S(int) length;};
   Btdot1::intern dot
           (VecDS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;} };
   Bterr::extern error();
   Btdot2::intern dot
           (VecSS(struct vec) S(*) u,
            VecDS(struct vec) S(*) v)
           { needs{Bterr;} };
 ...}...}
 Exports {VecDS; VecSS; Btdot1; Btdot2; ...}
}
```

FIG. 4.4 – Modules de spécialisation : Spécialisation de la fonction `dot`.

les scénarios définis, ils sont accessibles depuis l'extérieur du module si ils sont inclus dans la clause `Exports`.

Initialisation du contexte de spécialisation. Il existe parfois, pour l'utilisateur, la nécessité d'initialiser les valeurs des variables statiques globales ainsi que les paramètres statiques d'entrée. C'est le cas du contenu du vecteur `u` et du pointeur sur le vecteur `v` dans le scénario `Btdot2` de la figure 4.4(b). Pour cette raison, le langage de spécification de scénarios permet à l'utilisateur de spécifier ces valeurs à l'aide de la clause `inits` associée à un scénario.

4.2.2 Sous-spécialisation et sur-spécialisation

L'idée que se fait l'utilisateur du résultat de la spécialisation peut varier significativement de la réalité. Par exemple, il se peut que les valeurs statiques ne soient pas propagées de la manière attendue et donc que le programme résultant ne soit pas complètement spécialisé. Au contraire, si les valeurs statiques sont propagées sur des contextes où il n'existe aucune opportunité d'optimisation, à l'exception de quelques constantes, alors la spécialisation risque de générer une explosion du code où, dans le pire des cas, elle entre dans une boucle infinie. Dans le premiercas, nous parlons d'un problème de *sous-spécialisation*, tandis que dans le second cas nous parlons de *sur-spécialisation.*

Normalement, la sous-spécialisation est une conséquence des approximations résultant de l'analyse de temps de liaison, telles que la granularité dans le traitement des structures

de données, le flot de contrôle, etc. Les scénarios de spécialisation permettent d'aider l'utilisateur à identifier quelles sont les constructions qui deviennent dynamiques à cause de ces approximations. Dans ce cas, si le résultat de l'analyse n'est pas conforme au scénario spécifié, l'utilisateur peut savoir quelles sont les variables ou structures concernées par le problème de sous-spécialisation.

La sur-spécialisation, par contre, est produite par la propagation de valeurs statiques qui génèrent la duplication du code, généralement, le corps d'une boucle. Ce problème peut être éliminé par la spécification d'un scénario où la variable qui provoque la duplication est annotée comme dynamique. Par exemple, l'appel d'une fonction dans le corps d'une boucle peut conduire à la spécialisation de cette fonction pour chaque valeur de la boucle même quand le résultat de l'appel ne dépend pas de l'index de la boucle. Dans ce cas, un scénario où la fonction concernée est déclaré comme externe est une solution possible (voir le scénario `Bterr` de la figure 4.4(b)).

4.2.3 Compilation et application des modules

Une fois déclarés, les modules de spécialisation sont intégrés au processus de spécialisation par un compilateur qui permet de vérifier la dépendance entre les scénarios (*inter-module*), le bon usage des scénarios à travers les modules (*intra-module*) et, finalement, que les constructions référencées par les scénarios correspondent aux constructions du code source spécialisé. Ensuite, le compilateur traduit les scénarios de spécialisation sous la forme d'un fichier de configuration utilisé par Tempo pour effectuer la spécialisation proprement dite. Dans le processus de spécialisation, schématisé dans la figure 4.5, deux rôles sont proposés : le développeur et l'utilisateur.

Le développeur identifie les opportunités de spécialisation sur le code source et les exprime sous la forme de scénarios de spécialisation. L'outil (*Developer Interface*) produit, comme résultat de l'analyse, une instance du programme prêt à être spécialisé (*specializable component*) une fois que les valeurs concrètes sont fournies par l'outil de l'utilisateur (*user interface*) afin de générer finalement le programme spécialisé (*specialized component*).

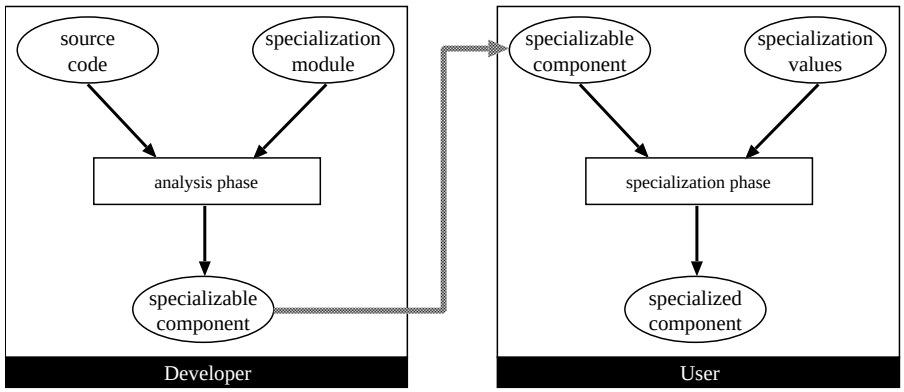


Fig. 4.5 – Scénarios de spécialisation : Processus de spécialisation.

4.3 Analyse globale et séparée de programmes

La connaissance de la totalité du code d'un programme et de son point d'entrée rend possible des analyses fines sur la façon dont chaque unité du programme est utilisée, ce qui permet, finalement, de le compiler plus efficacement. Cependant, à cause de la structuration des programmes permettant leur *modularisation* (par exemple, à l'aide de modules, classes, foncteurs, bibliothèques), une approche de l'analyse de programmes est devenue simplement irréaliste. Si une analyse statique requiert l'intégralité du programme, elle est incapable de tirer profit de la structure du programme pour factoriser l'effort d'analyse. Une approche qui s'adapte à la structure modulaire des programmes a sa place tout au long du processus de développement, pas seulement à la fin quand le développement du programme est complet. Concrètement, une analyse modulaire rend possible le traitement séparé des différents modules sans limiter leur utilisation ultérieure dans la construction d'un programme. Cela implique, parfois, la combinaison d'une approche *séparée*, laquelle est *complétée* par une approche *globale*. La question du caractère global ou séparé de la compilation concerne *a priori* tous les paradigmes de programmation.

Langages impératifs. Les optimisations mises en œuvre par les compilateurs de langages impératifs (voir, par exemple, [ASU86]) sont principalement de deux types. Il y a d'abord les optimisations dites *locales* qui sont effectuées en ne considérant que le code source d'un *bloc* de base à la fois. Dans tous les autres cas, nous parlons d'optimisations *globales*. À partir du graphe de flot de données, il est possible de calculer des informations qui serviront aux différentes optimisations. Le calcul de ces informations s'effectue généralement par approximations successives d'un point fixe. On calcule d'abord les informations locales à chaque bloc de base et ensuite on propage ces informations le long des arcs du graphe, jusqu'à ce qu'on ne puisse plus dériver d'informations nouvelles.

Langages d'ordre supérieur. Dans le cas des langages d'ordre supérieur comme ML, Lisp ou Haskell, les analyses sont plus complexes. Même si les graphes de flot de contrôle sont souvent plus simples, les boucles étant habituellement exprimées à l'aide de fonctions récursives dans le cas des langages fonctionnels et les appels de fonctions étant très fréquents, des analyses globales interprocédurales deviennent nécessaires pour calculer les informations qui permettront d'optimiser les programmes. Dans ce contexte, la détermination du graphe de contrôle devient un problème de flot de données lequel est déterminé à l'aide du graphe de flot de contrôle lui-même.

Langages à objets. Les langages à objets, bien que souvent plus près des langages impératifs quant à leur syntaxe, ne sont pas aisés à optimiser. Bien que la plupart des langages à objets ne permettent pas de créer des fonctions de première classe, ils offrent des défis semblables à ceux des langages d'ordre supérieur en raison de caractéristiques comme l'héritage ou le polymorphisme. Ici, la détermination de la classe exacte du receveur d'un message, permettant de décider quelle méthode exactement doit être appelée, est une propriété dynamique, c'est-à-dire qu'elle dépend de l'exécution du programme. On

comprend donc l'importance d'analyses de haut niveau permettant d'optimiser ces langages. L'analyse la plus importante est certainement l'analyse de la *hiérarchie de classes* (ou de type). Cette analyse permet de déterminer, pour chaque site d'appel, l'ensemble des classes concrètes dont le receveur de l'appel peut être une instance à l'exécution.

Dans ce contexte, Besson et Jensen [BJ03] proposent une analyse de programmes Java modélisés par un ensemble de contraintes dont la plus petite solution est le résultat de l'analyse. Le fait de travailler sur un langage à chargement dynamique, comme Java, entraîne l'indisponibilité d'une partie du code au moment de l'analyse. Par conséquent, l'objet associé à un fragment (classe) de programme est un système de contraintes *partielles* et la composition de fragments est modélisée par un ensemble d'opérateurs. Autre exemple, Privat et Ducournau [PD04], proposent l'intégration de techniques d'optimisation globales, à savoir l'analyse de types et la coloration, dans un cadre de compilation séparée. Cette approche a été appliquée sur des langages à objets sans chargement dynamique, tels que Eiffel et C++. L'analyse de types est décomposée en une phase locale, réalisable au cours d'une compilation séparée, et une phase globale, qui peut s'effectuer sur la totalité des codes compilés. La phase de compilation séparée produit un code analogue à celui d'une compilation traditionnelle, mais ce code est accompagné des informations nécessaires à l'optimisation globale ultérieure. Concrètement, le code contient des balises permettant des substitutions à l'issue de cette optimisation.

4.4 Bilan

Dans ce chapitre, différentes approches qui permettent le traitement des unités de programme de manière séparée ont été décrites. Une analyse statique, par définition, a lieu au moment de la compilation des modules et, donc, le résultat est une approximation qui ne peut pas être affinée en l'absence des informations sur le contexte de liaison. Comme nous l'avons vu dans ce chapitre, la plupart des approches combinent les deux points de vue, séparé (local) et global, en retardant l'analyse le plus possible. Cette sorte d'analyse (ou optimisation) *tardive* est la base de l'approche générative où l'analyse (ou optimisation) est finalement réalisée quand le module est associé à un contexte d'application concret. Nous verrons dans le chapitre 8, comment il est possible d'appliquer cette approche à la spécialisation des applications à base de composants.

Deuxième partie

Contribution

Chapitre 5

Introduction à la spécialisation dans les langages à objets

Sommaire

5.1 Problématique **96**
5.1.1 Annotation structurelle 96
5.1.2 La spécialisation et l'héritage 99
5.1.3 Sensibilité au polymorphisme 106
5.2 Notre approche **108**
5.2.1 L'analyse 108
5.2.2 La spécialisation 111
5.3 Le langage : EFJ **112**
5.3.1 Syntaxe 112
5.3.2 Un exemple : programme *Point2DStepping* 115
5.3.3 Constructions annotables 116
5.3.4 Règles de types déclarés 116
5.4 Bilan **118**

Dans le chapitre 2 nous avons décrit la spécialisation de programmes appliquée principalement à un sous-ensemble impératif de Java. Toutefois, la prise en compte des caractéristiques associées aux langages à objets, à savoir la définition de classes, l'héritage et le polymorphisme, requiert d'affiner les analyses mentionnées dans le chapitre 2. De nouvelles sensibilités doivent être considérées et il est nécessaire de concilier spécialisation de programme et héritage. Dans ce chapitre nous étudions la problématique associée à la spécialisation dans les langages à objet ainsi que l'approche de spécialisation prise en compte dans cette thèse. Afin de simplifier la définition de l'évaluateur partielle hors ligne montrée dans la chapitre 6 nous abordons aussi les caractéristiques du langage à objets, EFJ, une extension de la version de FJ [IPW01] proposée par Schultz [Sch00], qui a été conçu comme un sous-ensemble de Java préservant les caractéristiques intrinsèques aux langages à objets.

5.1 Problématique

Dans cette section, nous décrivons les problèmes liés à la spécialisation de programmes conçus sur des langages à objets à base de classes avec héritage. En effet, un programme est conçu comme un ensemble de classes qui encapsulent des méthodes, des champs et des constructeurs, où les classes peuvent être instanciées en des objets et réutilisées grâce aux mécanismes d'héritage.

5.1.1 Annotation structurelle

La structure apportée par la définition d'une classe est partagée, en général, par des objets différents créés à partir de l'instanciation de la classe mais aussi par des objets créés à partir des sous-classes. Pour simplifier la discussion nous ne considérons, dans cette section, que des programmes où l'héritage n'intervient pas. L'introduction des classes implique deux types d'analyse : la sensibilité à la structure et la sensibilité à l'instanciation.

5.1.1.1 Analyse sensible à la structure

Considérer une construction de programme seulement comme statique ou comme dynamique s'avère insuffisant pour annoter des constructions associées à un type composé comme celui associé à une classe par une classe. Une analyse *sensible à la structure* calcule de manière indépendante le temps de liaison des champs d'une classe. L'annotation de temps de liaison de la classe est composée par les annotations sur les champs. Nous appelons l'annotation de classe *annotation structurelle* par opposition à celle associée à une construction de type primitif (*i.e.*, `int`, `boolean`,...), appelée *annotation primitive*. Afin d'illustrer cette approche, nous prenons la classe `Point2D` de la figure 5.1(a). La structure de la classe est composée de trois champs : le champ `x` qui représente l'abscisse du point, le champ `y` qui représente son ordonnée et le champ `offset` qui représente le pas permettant de calculer les coordonnées du point suivant dans une grille comme le montre l'implémentation de la méthode `next`. La figure 5.1(b) montre l'analyse du temps de liaison de la classe `Point2D` où le champ `x` et le champ `offset` sont affectés à des valeurs statiques tandis que le champ `y` reste dynamique. L'analyse de la méthode `next` est effectuée, en plus, par rapport au temps de liaison du paramètre `dec`. L'annotation structurelle permet d'annoter les constructions dont le type correspond à une classe en fonction du temps de liaison de cette dernière, comme, par exemple, dans le cas de la référence à l'instance (`this`) ou du retour de la méthode `next`, qui peuvent être annotés selon l'annotation de la classe `Point2D`. Cette annotation structurelle (voir figure 5.1(b)) adopte la forme d'un vecteur des annotations des champs de la classe `Point2D` selon l'ordre de définition des champs. Concrètement, il s'agit de résidualiser un champ dynamique et d'évaluer un champ statique appartenant à une même classe afin de mieux profiter des opportunités de spécialisation, principalement, en ce qui concerne l'accès aux membres. Cela permet de manipuler des instances de classes partiellement spécialisées. En revanche, une approche insensible à la structure associe une annotation primitive à toutes les constructions du programme indépendamment du type. Dans ce cas, l'annotation d'une classe est représentée par le plus petit majorant obtenu à partir du temps de liaison de ses champs.

(a) Classe `Point2D`.

```
(1) class Point2D {
(2)  int x;
(3)  int y;
(4)  int offset;

(5)  Point2D(int x,
(6)          int y,
(7)          int offset) {
(8)     this.x = x;
(9)     this.y = y;
(10)    this.offset = offset;
(11) }

(12) Point2D next(int dec) {
(13)    return new Point2D(
(14)       this.x + this.offset + dec,
(15)       this.y + this.offset + dec,
(16)       this.offset);
(17) }
(18)}
```

(b) Instanciation de la classe `Point2D`.

```
// Contexte: - Point2D.[x_S, y_D, offset_S]
             - Point2D.next.dec_S
(1) class Point2D_[S,D,S] {
(2)  int x_S;
(3)  int y_D;
(4)  int offset_S;

(5)  Point2D(int x_S,
(6)          int y_D,
(7)          int offset_S) {
(8)     this_[S,D,S].x_S = x_S;
(9)     this_[S,D,S].y_D = y_D;
(10)    this_[S,D,S].offset_S = offset_S;
(11) }

(12) Point2D_[S,D,S] next(int dec_S) {
(13)    return_[S,D,S] new Point2D(
(14)       this_[S,D,S].x_S + this_[S,D,S].offset_S + dec_S,
(15)       this_[S,D,S].y_D + this_[S,D,S].offset_S + dec_S,
(16)       this_[S,D,S].offset_S);
(17) }
```

Fig. 5.1 – Annotation structurelle sur la classe `Point2D`.

Ce type de sensibilité n'est pas nouveau, il a déjà été par exemple appliqué aux structures en C dans le cadre de Tempo, en étant considéré comme une partie intégrante de la sensibilité à l'utilisation (voir [HNC97] et section 2.1.2.1).

5.1.1.2 Analyse sensible à l'instanciation

La sensibilité à l'instanciation conditionne l'indépendance du temps de liaison d'un champ par rapport à celui du même champ appartenant à une instance différente. Nous allons illustrer cette idée avec un exemple simple (tous les champs sont de type primitif). Dans le programme de la figure 5.2(a) la classe `Point2D` est instanciée trois fois dans des contextes différents. Dans la figure 5.2(b), nous observons l'évolution des annotations du temps de liaison sur les champs de la classe `Point2D` tout au long de l'exécution du programme. Une analyse insensible à l'instanciation associe à un champ une unique

(a) Instanciation de la classe `Point2D`.

```
(19) class Program {
(20)  void main(int dyn, int sta) {
(21)    Point2D point1 = new Point2D(dyn, dyn, sta);
(22)    Point2D point1Next = point1.next(sta);
(23)    Point2D point2 = new Point2D(dyn, sta, dyn);
(24)    Point2D point2Next = point2.next(dyn);
(25)    Point2D point3 = new Point2D(dyn, sta, sta);
(26)  }
(27) }
```

(b) Annotations des instances créées.

```
// Contexte:
   - Program.main.[dyn, sta]
                   D    S
...
(21) point1.[x,y,offset]
             D D    S
(22) point1Next.[x,y,offset]
                 D D    S
(23) point2.[x,y,offset]
             D S    D
(24) point2Next.[x,y,offset]
                 D D    D
(25) point3.[x,y,offset]
             D S    S
...
```

FIG. 5.2 – Instanciation de la classe `Point2D`.

annotation de temps de liaison pour toutes les instances. Autrement dit, tous les objets instances d'une même classe partagent le même temps de liaison. En fait, l'annotation d'un champ prend le plus petit majorant (voir section 2.1.2.1) des valeurs affectées lors de l'analyse. L'analyse monovariante prendra donc en compte un seul objet type receveur dont tous les champs seront annotés comme dynamique car à chacun des champs ont été affectées des valeurs dynamiques en différents points de l'exécution du programme. Cela implique que la classe ne sera pas spécialisée.

En revanche, une analyse polyvariante associe différentes annotations à un même champ en fonction des objets instanciés. À partir des instances créées par le programme de la figure 5.2(a), l'analyse se base sur les trois contextes d'instanciation possibles, un contexte pour chaque point d'instanciation du programme (lignes `(21)`, `(23)` et `(25)`). D'une part, la figure 5.3(a) montre le résultat de l'analyse la classe `Point2D` en prenant en compte le contexte d'instanciation de l'objet `point1`. Le résultat de la spécialisation de la classe en fonction des valeurs concrètes, une nouvelle classe `Point2Dpoint1` est montré dans la figure 5.3(b). Effectivement, nous constatons la propagation de la valeur ainsi que l'élimination de la définition du champ `offset` (cette élimination est une optimisation orthogonale à la sensibilité à l'instanciation). Il est aussi à noter que le retour de la méthode `next` est de type `Point2Dpoint1`. La création d'une instance au sein de la méthode `next` n'a en effet pas crée de nouveau contexte de spécialisation. Cela est possible car ce champ reste statique pour l'instance `point1` tout au long de l'exécution du programme. D'autre part (voir figure 5.4), l'instance `point2` a été créée à partir de valeurs dynamiques pour l'abscisse et le pas tandis que une valeur statique est affectée à l'ordonnée. Notons que la propagation d'une valeur dynamique à travers le champ `offset` affecte les annotations du champ `y` à cause de l'affectation de la ligne `(15)` de la méthode `next`. En effet, une analyse de temps de liaison monovariante par rapport à ce contexte fait monter la valeur abstraite du champ `y` de statique à dynamique. Par contre, à travers une analyse du temps d'évaluation (voir section 6.3) sur le résultat d'une analyse du temps de liaison

(a) Instanciation de la classe `Point2D`.

```
// Contexte: - Point2D.[x_D, y_D, offset_S]
             - Point2D.next.dec_S
(1) class Point2D {
(2)  int x_S;
(3)  int y_D;
(4)  int offset_S;

(5)  Point2D(int x_D,
(6)          int y_D,
(7)          int offset_S) {
(8)      this_[D,D,S] .x_D =_D x_D;
(9)      this_[D,D,S] .y_D =_D y_D;
(10)     this_[D,D,S] .offset_S =_S offset_S;
(11) }

(12) Point2D_[D,D,S] next(int dec_S) {
(13)    return_[D,D,S] new Point2D(
(14)        this_[D,D,S] .x_D +_D this_[D,D,S] .offset_S +_D dec_S,
(15)        this_[D,D,S] .y_D +_D this_[D,D,S] .offset_S +_D dec_S,
(16)        this_[D,D,S] .offset_S);
(17) }
```

(b) Spécialisation de la classe `Point2D`.

```
// Valeurs : - Point2D.offset = 1
             - Point2D.next.dec = 1
(1) class Point2Dpoint1 {
(2)  int x;
(3)  int y;
(4)  [          ]

(5)  Point2Dpoint1(int x,
(6)                int y,
(7)                [          ]) {
(8)     this.x = x;
(9)     this.y = y;
(10)    [                    ]
(11) }

(12) Point2Dpoint1 next_dec1([          ]) {
(13)    return new Point2Dpoint1(
(14)       this.x + 2,
(15)       this.y + 2,
(16)       [          ]);
(17) }
```

FIG. 5.3 – Analyse de la classe `Point2D` par rapport à l'instance `point1`.

il est possible de conserver le contexte initial d'instanciation. La figure 5.4(b) montre le résultat de l'analyse d'évaluation obtenu à partir de l'analyse de temps de liaison de la figure 5.4(a). Concrètement, l'analyse d'évaluation permet d'annoter le champ `y` comme statique pendant l'instanciation mais comme dynamique lors des utilisations ultérieures (i.e. appels de méthodes). Finalement, l'utilisation de l'instance `point3` n'affecte aucun temps de liaison du contexte d'instanciation (voir ligne `(25)` dans la figure 5.2(a)).

5.1.2 La spécialisation et l'héritage

5.1.2.1 Langages à objets sans héritage

Les approches décrites auparavant ne considèrent pas l'héritage entre les classes du programme, en conséquence, le résultat de la spécialisation est une classe résiduelle qui encapsule les méthodes spécialisées. La hiérarchie des classes voit donc sa structure se

(a) Instanciation de la classe `Point2D`.

```
// Contexte: - Point2D.[x, y, offset]
                      D  S     D
             - Point2D.next.dec
                             S
(1) class Point2D {
(2)  int x;
          D
(3)  int y;
          S
(4)  int offset;
             D
(5)  Point2D(int x,
                 D
(6)          int y,
                 S
(7)          int offset) {
                    D
(8)      this .x = x;
        [D,S,D]  D  D
(9)      this .y = y;
        [D,S,D]  S  S
(10)     this .offset = offset;
        [D,S,D]     D     D
(11) }

(12) Point2D next(int dec) {
      [D,D,D]          S
(13)    return new Point2D(
         [D,D,D]
(14)        this .x + this .offset + dec,
           [D,S,D]  D     [D,S,D]     D   D   S
(15)        this .y + this .offset + dec,
           [D,S,D]  S     [D,S,D]     D   D   S
(16)        this .offset);
           [D,S,D]     D
(17) }
```

(b) Spécialisation de la classe `Point2D`.

```
// Valeurs : - Point2D.offset = 1
             - Point2D.next.dec = 1
(1) class Point2Dpoint2 {
(2)  int x;
(3)  [          ]
(4)  int offset;

(5)  Point2Dpoint2(int x,
(6)                [              ]
(7)                int offset) {
(8)     this.x = x;
(9)     [              ]
(10)    this.offset = offset;
(11) }

(12) Point2D next_dec1([          ]) {
(13)    return new Point2D(
(14)       this.x + this.offset + 1,
(15)       1 + this.offset + 1,
(16)       this.offset);
(17) }
```

FIG. 5.4 – Analyse de la classe `Point2D` par rapport à l'instance `point2`.

modifier à cause de l'insertion des nouvelles classes. Chacune des classes résiduelles correspond aux abstractions des objets receveurs (i.e. classification des objets receveurs selon les champs statiques et dynamiques) qui ont été calculées lors de l'analyse. La structure associée aux classes résiduelles ne contient que la définition des champs qui restent dynamiques comme, par exemple, la classe `Point2Dpoint1` dont la structure ne considère pas la définition du champ `offset` (voir figure 5.3(b)).

Il existe deux manières d'introduire des classes résiduelles : soit on considère les objets receveurs comme instances des classes résiduelles, soit on considère l'utilisation d'une classe englobante qui permet, lors de l'exécution, de choisir la méthode adéquate. Toutefois, il est aussi possible de ne pas introduire de nouvelle classe en modifiant la classe cible [BN02]. Dans le premiercas, le choix de la méthode spécialisée est effectué depuis le site invoquant tandis que dans le deuxième cas la spécialisation est effectuée depuis le site invoqué.

Spécialisation depuis le site invoquant. La spécialisation depuis le site invoquant remplace l'instanciation de la classe d'origine par l'instanciation d'une classe résiduelle. Par exemple, en reprenant les classes résiduelles générées à partir des contextes d'instanciation mentionnés, toutes les instances des objets dans lesquelles les valeurs des champs `x` et `y` sont dynamiques alors que la valeur du champ `offset` ne change pas et est égale à `1` sont résidualisées comme instances de la classe `Point2Dpoint1`. Pour les programmes Java, cette approche force une valeur à être identique tout le long du programme. Il existe, par contre, des langages comme NeoClassTalk [Riv87], Fickle [DDDCG00] et Cecil [Cha93] où les objets peuvent changer de classe au cours de l'exécution. Cela permettrait de changer la classe résiduelle de manière dynamique. Cependant, cette approche devient difficile à appliquer lorsqu'il existe plusieurs références sur un même objet. Dans ce cas, le remplacement d'un objet par une instance d'une classe résiduelle impose le changement de la classe de toutes les références sur l'objet, ce qui demande à prendre en compte les alias dans l'analyse. Il est en effet nécessaire de connaître pour chaque référence ses points d'instanciation possibles (il peut y en avoir plusieurs du fait des alias) et de faire en sorte qu'une seule classe résiduelle (du moins en l'absence d'héritage) soit associée à ces points d'instanciation. Il est alors nécessaire de faire appel à une analyse d'alias pour pouvoir spécialiser la méthode `next` par rapport au contexte d'instanciation adéquat qui dans ce cas correspond à celui de l'instance `point1`.

Spécialisation depuis le site invoqué. La spécialisation depuis le site invoqué demande la création d'une nouvelle classe qui branche le flot de contrôle du programme sur la méthode spécialisée selon les valeurs concrètes. Cette classe prend la forme d'une classe *factory* où une instance de la classe résiduelle correspondante est créée en fonction du temps de liaison et des valeurs concrètes des paramètres. En l'absence d'héritage il n'est pas possible de définir une seule méthode qui retourne une instance de n'importe quelle classe résiduelle car il n'existe pas de relation entre elles. De ce fait, une solution envisageable est d'associer à la classe englobante la structure résiduelle de telle manière que tous les objets créés soient instances de cette classe. Cette approche résulte donc en un ensemble de classes résiduelles différent de celui de l'approche précédente. Notons que la classe englobante devra conserver, comme la structure résiduelle, tous les champs de la classe d'origine `Point2D` puisque ils sont tous annotés comme dynamiques dans les instances créées lors de l'exécution du programme. Pour cette raison, nous modifions la classe `Point2D` comme le montre la figure 5.5(a).

5.1.2.2 Langages à objets avec héritage

L'inclusion de l'héritage entre les classes du programme implique que nous aurons besoin de marquer la différence entre le type *déclaré* d'une variable, qui n'est que la classe utilisée lors de la déclaration de la variable, et le type *concret* d'une variable, qui correspond à la classe de l'instance affectée à la variable lors de l'exécution.

Cas de base. Nous allons d'abord considérer le cas où la redéfinition d'une méthode (*method overriding*) n'est pas permise. Basée sur l'hypothèse qu'aucune classe n'est créée,

(a) Classe englobante `Point2D`.

```
(1) class Point2D {
(2)  int x;
(3)  int y;
(4)  int offset;

(5)  Point2D(int x,
(6)          int y,
(7)          int offset) {
(8)     this.x = x;
(9)     this.y = y;
(10)    this.offset = offset;
(11) }

(12) Point2D next_offset_dec(int dec) {
(13)  if ((this.offset == 1) && (dec == 1))
(14)   return Point2Dpoint1.next_1_1(this);
(16)  if ((this.offset == 2) && (dec == 2))
(17)   return Point2Dpoint1.next_2_2(this);
(19)  ...
(20) }

(21) Point2D next_y(int dec) {
(22)  if (this.y == 1)
(23)   return Point2Dpoint2.next_1(this,
(24)                               dec);
(25)  ...
(26) }
(27)}
```

(b) Classe résiduelle `Point2Dpoint1`

```
(1) class Point2Dpoint1 {
(2)  static Point2D next_1_1(Point2D p){
(3)   return new Point2D(p.x + 2, p.y + 2, 1);
(6)  }
(7)  static Point2D next_2_2(Point2D p){
(8)   return new Point2D(p.x + 4, p.y + 4, 2);
(11) }
(12)}
```

(c) Classe résiduelle `Point2Dpoint2`

```
(1) class Point2Dpoint2 {
(2)  static Point2D next_y(Point2D p,int dec){
(3)   return new Point2D(p.x + p.offset + dec,
(4)                      1 + p.offset + dec,
(5)                      p.offset);
(6)  }
(7) }
```

(d) Résidualisation du programme `Program`.

```
(1) class Program2 {
(2)  void main1(int dyn, int sta) {
(3)   Point2D point1 = new Point2D(dyn,dyn,sta);
(4)   Point2D point1Next =
(5)           point1.next_offset_dec(sta);
(6)   Point2D point2 = new Point2D(dyn,sta,dyn);
(7)   Point2D point2Next = point2.next_y(dyn);
(8)   Point2D point3 =
(9)           new Point2Dpoint3(dyn,sta,dyn);
(10) }
(11)}
```

FIG. 5.5 – Spécialisation depuis le site invoqué.

et donc que la hiérarchie ne voit pas sa structure modifiée, la question est simplement de savoir quelle est la classe qui contiendra les nouvelles méthodes spécialisées. Cela dépendra du type de receveur qui, dans ce contexte, est le type concret du receveur. Si l'information concernant les types concrets n'est pas disponible alors une approximation sûre doit être choisie. Comme le type concret est un sous-type du type déclaré, la méthode devra être incluse dans la classe de la définition du receveur. Par contre, en réalisant une analyse de type sur le programme, c'est-à-dire en déterminant l'ensemble des types concrets susceptibles d'être affectés à une variable, l'élection de la classe à modifier peut être faite avec plus de précision. Si l'ensemble calculé est un singleton alors la méthode est insérée dans cette classe. Un aspect important de cette approche est qu'elle permet de spécialiser des expressions qui dépendent des types comme par exemple, l'opérateur `instanceof`. Par contre, si la variable est associée à un ensemble non singleton lors de l'analyse de type, la stratégie générale consiste à partitionner l'ensemble résultant de telle sorte qu'une méthode spécialisée est insérée pour chaque élément de la partition. Parallèlement, si une partition contient plusieurs types une unique version spécialisée est générée et insérée dans le type plus général de cet élément (i.e. superclasse commune). Afin d'illustrer cette approche nous considérons deux nouvelles classes, la classe `Point3D` qui étend la classe `Point2D` en ajoutant le champ `z` qui représente la côte du point en 3D et la classe `Point3DColore` qui étend la classe `Point3D` avec la couleur du point. Par exemple, si l'analyse de type conclut qu'un objet `p` est associé à l'ensemble {`Point2D, Point3D, Point3DColore`} et la partition sur cet ensemble est {`Point2D`}, {`Point3D, Point3DColore`}, une méthode spécialisée est ajoutée à la classe `Point2D` et une autre méthode à la classe `Point3D`. Le but est donc de générer une partition optimale afin d'éviter la duplication de code.

Redéfinition des méthodes. Considérons maintenant la possibilité de redéfinir la méthode `next` pour pendre en compte la côte `z`. La figure 5.6 montre respectivement l'implémentation des classes `Point3D` et `Point3DColore` mentionnées auparavant. Notons que, à cause de la règle d'invariance fixée pour la redéfinition des méthodes dans le langage Java, le type des paramètres d'entrée et le type de la valeur de retour doivent être exactement les mêmes. L'exemple de la figure 5.6, correspond au *problème de la méthode binaire* (*binary method problem*), décrit par Bruce et al. [BCC+95].

Dans cette approche, la tactique d'insertion décrite auparavant est encore valide sauf que la partition est contrainte par l'existence de méthodes redéfinies. Considérons un exemple dans lequel le type d'un objet receveur de la méthode `next` peut être `Point2D` ou `Point3DColore` et le paramètre `dec` est égal à `2`. Dans ce cas l'insertion d'une seule méthode spécialisée dans la classe plus générique est insuffisante. En effet, cela requerrait la génération de deux versions de la méthode spécialisées par rapport aux deux types mentionnés, liées par une conditionnelle qui permettrait de choisir la version correcte. La figure 5.7(a) montre la méthode `next_1` insérée dans la classe (la plus générique) `Point2D`. La conditionnelle dans cette méthode, utilisant l'opérateur `instanceof`, permet d'exécuter la méthode spécialisée correcte en fonction du type concret du receveur (`this`). Cette solution étant une espèce de recodification du dispacheur dynamique de la méthode, elle conduit à un double *dispatch*. Cependant, une solution plus performante se baserait plutôt sur la création de deux méthodes spécialisées partageant la même signature et insérées en

(a) Classe `Point3D`.

```
(1)  class Point3D extends Point2D{
(2)   int z;
(3)   Point3D(int x, int y, int z,
                int offset){
(4)     super(x,y,offset);
(5)     this.z = z;
(6)   }
(7)   Point3D next(int dec){
(8)     return
          new Point3D(
                this.x + this.offset + dec,
(9)             this.y + this.offset + dec,
(10)            this.z + this.offset + dec,
(11)            this.offset);
(12)  }
(13) }
```

(b) Classe `Point3DColore`.

```
(1)  class Point3DColore extends Point3D{
(2)   int color;
(3)   Point3D(int x, int y, int z,
                int offset, int color){
(4)     super(x,y,z,offset);
(5)     this.color = color;
(6)   }
(7)   Point3DColore next(int dec){
(8)     return
          new Point3DColore(
                this.x + this.offset + dec,
(9)             this.y + this.offset + dec,
(10)            this.z + this.offset + dec,
(11)            this.offset,
(12)            this.color);
(13)  }
(14) }
```

FIG. 5.6 – Héritage et redéfinition de méthodes.

chacune des classes concernées, `Point2D` et `Point3DColore`. La figure 5.7(b) montre les classes `Point2D` et `Point3DColore` contenant respectivement une version spécialisée de la méthode.

5.1.2.3 Création de nouvelles classes

Considérons, maintenant, la création de nouvelles classes pour l'encapsulation des méthodes spécialisées. Établir une relation d'héritage entre la classe d'origine et la classe résiduelle permettrait la création des instances spécialisées compatibles avec les instances génériques en ce qui concerne le typage. Dans ce contexte, donc, on peut dire que l'héritage et l'évaluation partielle coïncident. Mais, en fait, cette idée ne peut pas être généralisée. Considérons le cas présenté auparavant où un objet susceptible d'être de type `Point2D` ou `point3DColore` est le receveur de la méthode `next`. La spécialisation de la méthode `next` en fonction d'un `offset` donné implique la résidualisation de deux classes, par exemple `SPoint2D` et `SPoint3DColore`, contenant la méthode spécialisée correspondante. L'idée proposée imposerait de définir les classes résiduelles comme sous-classes des classes originales mais aussi la classe `SPoint3DColore` comme sous-classe de `SPoint2D`. Cependant, face à l'absence d'héritage multiple, la classe `SPoint3DColore` ne peut pas être sous-classe de la classe `Point3DColore` et de la classe `SPoint2D`. La figure 5.8 montre l'ensemble des classes ajoutées à la hiérarchie originale où un des liens doit être supprimé. Le choix de la relation à éliminer peut être effectué sur la base d'une analyse de type qui permettrait de savoir si lors de l'exécution deux instances des classes liées sont réellement prises comme

(a) Insertion dans la classe la plus générique.

```
(1)  class Point2D {
(2)   ...
(3)   Point2D next_1(){
(4)     if (this instanceof Point2D)
(5)     return next_1_Point2D(this);
(6)     if (this instanceof Point3DColore)
(7)     return next_1_Point3DC(this);
(8)   }
(9)   ...
(10)  static Point2D next_1_Point2D(
                              Point2D p){
(11)    return new Point2D(
(12)            this.x + this.offset + 1,
(13)            this.y + this.offset + 1,
(14)            this.z + this.offset + 1,
(15)            this.offset);
(16)  }
(17)  ...
(18)  static Point3D next_1_Point3DC(
(19)                        Point3DColore p){
(20)   return new Point3DColore(
(21)            this.x + this.offset + 1,
(22)            this.y + this.offset + 1,
(23)            this.z + this.offset + 1,
(24)            this.offset,
(25)            this.color);
(26)  }
(27)  ...
(28) }
```

(b) Insertion dans les types receveurs.

```
(1)  class Point2D{
(2)   ...
(3)   Point2D next_1(){
(4)     return
(5)       new Point2D(
(6)             this.x + this.offset + 1,
(7)             this.y + this.offset + 1,
(8)             this.z + this.offset + 1,
(9)             this.offset);
(10)  }
      ...
(11) }

(12) class Point3DColore extends Point3D{
(13)  ...
(14)  Point2D next_1(){
(15)    return
(16)      new Point3DColore(
(17)            this.x + this.offset + 1,
(18)            this.y + this.offset + 1,
(19)            this.z + this.offset + 1,
(20)            this.offset,
(21)            this.color);
(22)  }
(23)  ...
(24) }
```

FIG. 5.7 – Critère d'insertion de la méthode spécialisée.

sous-types. Sinon, une solution possible peut être une transformation générale qui permet de simuler l'héritage multiple.

Notons, finalement, qu'il existe des situations dans lesquelles l'utilisation de la technique d'*interclassement* (*interclassing*) [RN01] est plus adéquate. Elle permettrait d'insérer la classe résiduelle comme une superclasse de la classe originale pouvant supprimer, de cette façon, les champs qui restent invariants dans la classe spécialisée. Ici, nous mettons en évidence la différence entre la spécialisation par l'héritage et par l'évaluation partielle malgré leur ressemblance superficielle : l'évaluation partielle tend à réduire la structure d'une classe tandis que le sous-classement tend à l'augmenter.

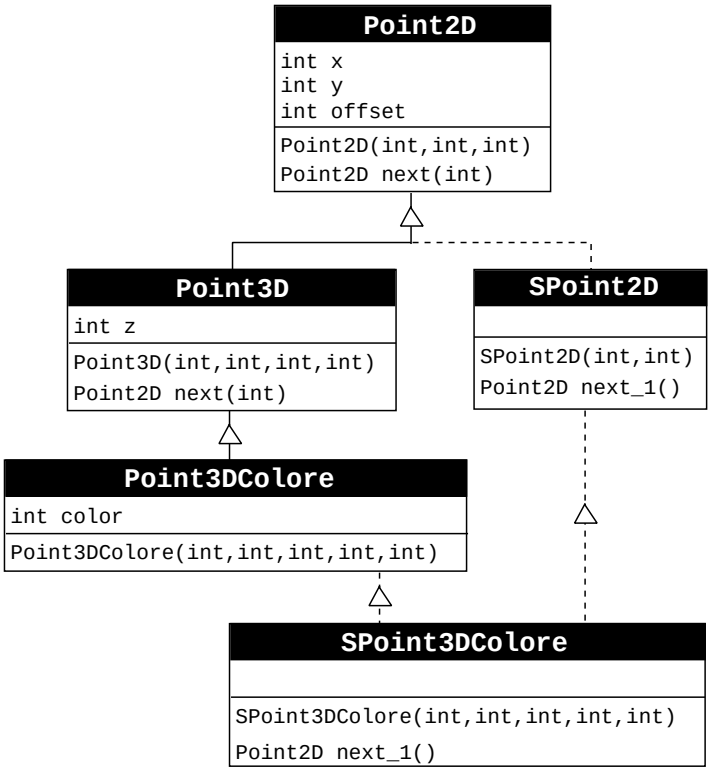


FIG. 5.8 – Héritage vs. évaluation partielle

5.1.3 Sensibilité au polymorphisme

L'héritage crée une autre complication quand il s'agit de spécialiser la définition d'une méthode pour un appel donné, ce qui nous conduit à introduire une notion de *sensibilité au polymorphisme*. Ici, nous considérons le *polymorphisme d'inclusion* comme défini par Cadelli et al. [CW85]. Dans ce contexte, un objet peut être considéré comme appartenant à des classes différentes, qui se trouvent en relation d'inclusion. Comme dans le langage Java la structure d'héritage détermine la relation de sous-typage, on dit qu'un type `T` est inclus dans un type `T'` si `T` est une sous-classe de `T'`. Afin d'illustrer cette approche, nous reprenons la définition des classes `Point2D` de la figure 5.1(a) et les classes `Point3D` et `Point3DColore`. Par exemple, une variable déclarée de type `Point2D` pourrait contenir soit une instance de la classe `Point2D` soit une instance d'une sous-classe, à savoir `Point3D` ou `Point3DColore`. Parfois, à cause du flot de données du programme et de la possibilité de redéfinition de méthodes, il est impossible d'identifier, au moment de la compilation, la méthode cible de l'appel (i.e. l'implémentation correcte). Autrement dit, dans ce contexte, il ne sera possible que de calculer un ensemble de méthodes susceptibles d'être appelées, qui dans ce cas seront toutes les méthodes qui partagent le même nom de méthode ainsi que le nombre et les types des paramètres. Un langage comme Java se base sur la liaison tardive (*late binding*) pour le choix de la méthode correcte. Elle permet de retarder la

sélection (*lookup*) de la méthode jusqu'au moment de l'exécution du programme, où le type de l'instance concernée devient connu. Pour cette raison, la sensibilité au polymorphisme permet de limiter le nombre de méthodes cibles de la spécialisation en calculant les types des instances. La figure 5.9(a) montre le programme `Poly` dont le type déclaré de la méthode `choix` est `Point2D`. Cependant, l'expression conditionnelle de la méthode implique que, selon la valeur du paramètre `is3D`, la valeur retournée peut être du type `Point2D` ou `Point3D`. Considérons aussi l'appel de la méthode `next` dans la ligne `(3)` de la figure 5.9(b).

(a) Classe `Poly`.

```
(1)  class Poly {
(2)   ...
(3)   Point2D choix(boolean is3D){
(4)     if (is3D)
(5)      return new Point3D(1,1,1,10);
(6)     else
(7)      return new Point2D(0,0,1);
(8)   }
(9)   ...
(10) }
```

(b) Appel polymorphique.

```
(1)  ...
(2)  Point2D point = (new Poly()).choix(b);
(3)  point.next(d);
(4)  ...
```

(c) Classe `Natural`.

```
(1)  class Natural {
(2)   ...
(3)   int next(int dec){
(4)     return this.num + dec;
(5)   }
(6)   ...
(7)  }
```

FIG. 5.9 – Sensibilité au polymorphisme.

Une analyse insensible au polymorphisme ne permet pas de connaître les types des instances possibles affectées à la variable `point`. En conséquences, afin de couvrir tous les cas possibles il sera nécessaire de générer une version spécialisée pour chaque méthode `next` définie dans la classe `Point2D` et dans les sous-classes, c'est-à-dire `Point3D` et `Point3DColore`. Notez que, en l'absence d'information supplémentaire, il peut être nécessaire d'inclure dans l'ensemble des méthodes une méthode `next` qui appartient à une classe qui se trouve dans l'espace de noms de la classe `Poly` mais qui n'a aucune relation avec la classe `Point2D`. D'ailleurs, cette méthode pourrait retourner un type qui n'est pas en relation d'inclusion avec la classe `Point2D`, type primitif inclus (voir figure 5.9(c)).

Par contre, dans le cadre d'une approche sensible au polymorphisme, les méthodes spécialisées ne sont générées que pour les types concrets possibles. Pour l'exemple de la figure 5.9, seules les méthodes `next` des classes `Point2D` et `Point3D` sont spécialisées. Pour cela, l'étape d'analyse de temps de liaison devra être précédée par une analyse qui permet d'annoter le programme avec l'information de type concret.

5.2 Notre approche

Nous détaillons maintenant l'approche prise en compte dans notre travail pour la construction d'un évaluateur partiel hors ligne pour la spécialisation des programmes Java. La structure de cet évaluateur est présenté dans la figure 5.10. Pour des raisons de simplicité un sous-ensemble de Java appelé EFJ (voir section 5.3) est utilisé.

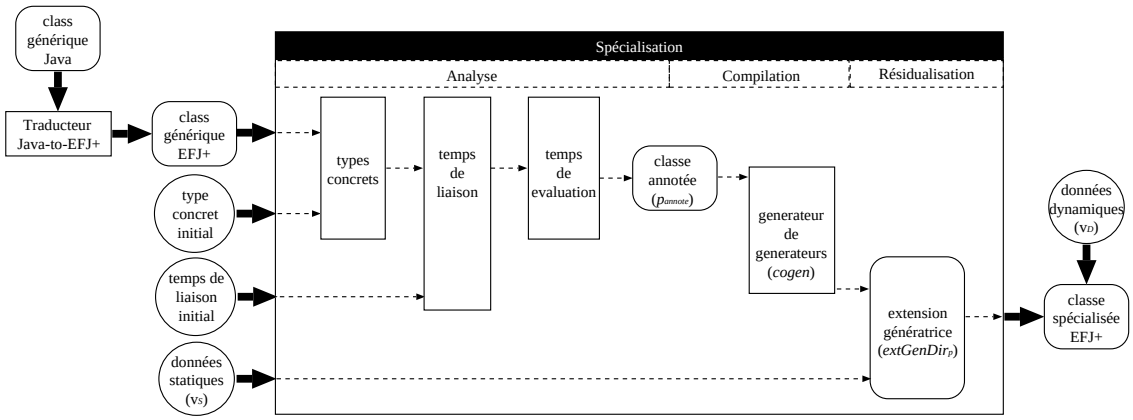


FIG. 5.10 – Approche pour la spécialisation

5.2.1 L'analyse

Afin d'éviter les problèmes mentionnés dans la section 5.1, en particulier ceux liés au polymorphisme, nous incluons dans le processus de spécialisation hors ligne classique (voir section 2.1.2) une analyse de types qui permettra d'annoter les constructions du programme analysé avec une approximation des types concrets. L'information sur les types concrets est utilisée, ensuite, pour l'analyse de temps de liaison ce qui implique une amélioration dans le résultat final de cette phase de l'analyse. Finalement, une dernière phase d'analyse est implémentée afin de calculer le temps d'évaluation des constructions qui permettra de savoir si une construction doit être évaluée à la spécialisation et/ou résidualisée. Dans tous les cas, l'analyse est monovariante, c'est-à-dire qu'il existe une seule annotation pour chaque construction analysée.

5.2.1.1 Règles de bonne annotation des programmes

Dans le domaine des langages de programmation statiquement typés, les règles de typage permettent de vérifier si un programme est bien typé, ce qui veut dire que l'on est sûr que les constructions du programme sont associées à un type compatible à leurs utilisations, par exemple, que les valeurs intervenant dans une addition sont de type entier, que le receveur d'une expression d'accès à un champ est d'un type compatible avec le champ. Ces règles sont la base de la vérification statique de types (*static type checking*) réalisée avant le processus de compilation qui utilise l'information de type déclaré exprimée dans le code source des programmes. Comme, traditionnellement, cette information n'intervient

pas lorsque le programme est exécuté, elle peut être vue comme une sorte d'annotation sur les constructions du programme.

Dans le cadre de ce document, nous nous appuyons sur cette idée pour la formulation de chacune des analyses mentionnées ci-dessus (voir figure 5.10), à savoir l'analyse de types concrets, l'analyse de temps de liaison et l'analyse de temps d'évaluation. Concrètement, nous allons vérifier et calculer, à l'aide d'un ensemble de règles de bonne annotions (*well-annotatedness rules*) [JGS93], les annotations données sur un programme.

5.2.1.2 Analyse par contraintes

Les règles de bonne annotation mentionnées ci-dessus ne constituent qu'une partie de l'analyse. En effet, elles ne disent rien sur le calcul proprement dit des annotations correspondantes. De plus, il est possible d'associer plusieurs annotations qui satisfont les règles proposées. Ici, nous considérons l'analyse de programmes comme un processus de propagation de valeurs abstraites (types, temps de liaison,...) en suivant le flot de donnée et de contrôle du programme analysé où l'exécution du programme est modélisé par rapport à une expression initiale fournie (*i.e.*, valeurs initiales de l'annotation). Dans ce contexte, nous ne sommes pas intéressés par toutes les annotations possibles sur le programme mais par celle qui représente le *mieux* un contexte d'exécution donné. L'optimisation de programmes étant une des contributions de ce document, la meilleure annotation est l'annotation qui donne la meilleure approximation possible de l'exécution du programme analysé.

Ceci se traduit dans chaque analyse de manière différente. D'une part, dans l'inférence de types nous somme intéressés sur l'association, aux constructions du programme, d'un ensemble singleton de types tel qu'il soit possible de déterminer de manière univoque quel est la méthode (où champ) appelée. Au pire, quand il n'est pas possible d'approximer le flot de contrôle, l'annotation résultante pour une expression donnée sera l'ensemble des types constitué par le type déclaré plus tous les sous-types (sous-classes) possibles. Ce dernier résultat, évidemment, ne conduira pas à une bonne spécialisation du programme (voir section 5.1.2) mais cependant, une annotation qui associe à chaque expression l'ensemble complet des types possibles est sans aucun doute validée par les règles de bonne annotation car il constitue la solution la plus sûre. D'autre part, dans l'analyse de temps de liaison, il est souhaitable d'obtenir l'annotation la plus statique (ou la moins dynamique) possible. Dans ce cas, une solution possible et aussi l'annotation la plus sûre sera celle qui affecte à chaque construction le temps de liaison le plus sûr, c'est-à-dire la valeur dynamique. En termes généraux, nous cherchons dans tous les cas *la plus petite annotation.*

Nous proposons une approche d'analyse de programmes à base de contraintes. Il s'agit d'une technique relativement puissante et bien comprise, modulaire, pour laquelle il existe des outils réutilisables. L'analyse de programmes par contraintes est divisée en deux étapes, la *génération* et la *résolution* des contraintes [Aik99]. Lors de la génération, un programme est représenté par un ensemble de contraintes qui décrivent une propriété souhaitée sur les constructions du programme analysé (par exemple, le flot de données) pour lesquelles une solution est cherchée durant l'étape de résolution.

Concrètement, nous partons des règles de bonne annotation de type déclaré pour la définition des règles de bonne annotation de type concret. Pour ceci, nous présentons une

notion informelle pour la conversion de la relation de typage vers la relation d'inclusion (voir section 6.1). Ensuite, un ensemble de contraintes est généré pour représenter les règles obtenues. La même procédure est appliquée pour l'analyse de temps d'évaluation. L'idée sous-jacente est de produire des analyses simples afin de réaliser un système complet et homogène pour l'analyse de programmes. La preuve formelle qui permet d'affirmer que les règles de bonne annotation (*i.e.*, type concret, temps de liaison, temps d'évaluation) ainsi que l'ensemble de contraintes générées sont correctes est mentionné dans la section de perspectives de cette document (voir section 11.1.1).

Définition des contraintes. Dans le cadre de notre travail, nous considérons les contraintes de la forme $V_{c_a} \oplus exp$, où V_{c_a} est une variable qui représente l'annotation sur la construction annotable de programme c_a (voir section 5.3.3) et $\oplus$ exprime la relation existante entre l'annotation sur la construction donnée et une expression *exp*. La relation $\oplus$ dépendra du domaine des annotations visées. Une expression peut être une constante, une variable ou un *opérateur conditionnel*. Les contraintes créées à partir des deux premiers types d'expressions sont appelées *contraintes simples*, en opposition aux *contraintes conditionnelles* créées à partir des opérateurs conditionnels.

Les contraintes conditionnelles ont été introduites par Palsberg *et al.* [PS94] pour exprimer le flot de contrôle des langages à objets qui devient particulièrement compliqué du fait de la liaison tardive (*late-binding*). À cause du polymorphisme, l'annotation d'une expression comme celle d'un accès à un champ, par exemple, dépendra du type de tous les receveurs possibles. Les contraintes conditionnelles sont de la forme $cond \rightarrow cstr$ où *cond* est un test de la forme $v \in V_{c_{a_1}}$ et *cstr* est une contrainte simple, par exemple $V_{c_{a_2}} \subseteq V_{c_{a_3}}$. L'évaluation de la contrainte conditionelle implique simplement que la contrainte $V_{c_{a_2}} \subseteq V_{c_{a_3}}$ sera incluse dans l'ensemble de contraintes seulement si v appartient aux valeurs de $V_{c_{a_1}}$. Une variante des contraintes conditionnelles, appelée *contrainte conditionnelle quantifiée*(CCQ) a été définie par Eluard et Jensen [EJ04], où une CCQ est définie par $\forall\ v \in V : cond(v) \rightarrow cstr(v)$ où V est une variable ensembliste définie par le système de contraintes, *cstr* est un ensemble de contraintes simples paramétrées par les valeurs v et *cond* représente les conditions à respecter par les valeurs v pour que les contraintes simples soient valides.

Génération des contraintes. Pour la génération des contraintes correspondantes on définit un générateur de contraintes dont la structure correspond aux catégories syntaxiques trouvées dans les programmes analysés (voir section 5.3). Les contraintes sont dérivées directement des règles de bonne annotation définies préalablement.

Résolution de contraintes. Lors de la résolution, une solution $\mathcal{I}$ satisfaisant l'ensemble des contraintes $\mathcal{C}$ est calculée. Comme mentionné ci-dessus, nous sommes intéressés par la plus petite solution (voir [Hei92a, Hei94]), dont l'existence est assurée par la monotonicité des opérateurs (en termes d'inclusion ensembliste) (voir [Sco76]) et par l'unicité de la solution puisque la partie droite des contrainte est une variable (voir [Hei92b]). Formellement, $\mathcal{I}$ est une interprétation de l'ensemble de contraintes $\mathcal{C}$, considérée comme une solution si

equation	::=	variable = expression | \idLambda . . . expression
expression	::=	'constante | (operateur expression) | variable

FIG. 5.11 – REQS : Syntaxe des équations.

pour chaque contrainte $V_{c_a} \oplus exp$, $\mathcal{I}(V_{c_a}) \oplus \mathcal{I}(\text{exp})$.

En plus, comme l'espace des solutions est fini, c'est-à-dire qu'il existe un nombre fini d'éléments susceptibles d'être utilisés dans les annotations (par exemple, nombre fini de types, nombre fini de valeurs de temps de liaison), la solution souhaitée peut être calculée en utilisant la méthode conventionnelle du point fixe [CC95].

Utilisation du solveur REQS Dans la recherche de la solution $\mathcal{I}$ nous utilisons le solveur d'équations récursives REQS (*Recursive Equation Solver*) [JPR99]. REQS applique une méthode itérative de point fixe générique (voir [JPR02]) pour résoudre un système d'équations décrites par le langage minimal dont la grammaire est décrite dans la figure 5.11. Les équations sont construites en affectant à une variable du système une expression qui peut être une constante du domaine, l'application d'un opérateur, une autre variable ou l'application d'une lambda variable sur une expression.

REQS fournit un ensemble de treillis en fonction du domaine des variables du système de contraintes. L'utilisation du solveur REQS demande, en premier lieu, la traduction des contraintes conditionnelles de l'ensemble $\mathcal{C}$, et en second lieu, la traduction des inégalités en égalités.

5.2.2 La spécialisation

Sur la base des analyses mentionnées ci-dessus, nous définissons un générateur d'extensions génératrices (`cogen`). Le résultat de l'application du générateur sur le programme annoté est un ensemble d'extensions génératrices : une extension pour chaque classe du programme.

Comme il a été expliqué dans la section 5.1, la hiérarchie de classes, qui dans le cas de Java correspond aussi à la hiérarchie de types (si on ignore les interfaces), et le polymorphisme intrinsèques aux langages à objets rendent la spécialisation de programmes moins évidente. Pour cette raison, nous considérons une approche conservatrice en ce qui concerne la génération du programme résiduel. La spécialisation effectuée sur les classes ne change pas la structure de la classe, c'est-à-dire que les classes résiduelles conservent les champs concernés dans la définition de la classe originale ainsi que leurs relations d'héritage. Seul le comportement de la classe se voit altéré à cause de l'inclusion des versions spécialisées des méthodes, en plus des version originales. Tandis que dans le premier cas on parle de la *spécialisation structurelle*, dans le deuxième on parle d'une approche *comportementale*.

5.3 Le langage : EFJ

Dans le cadre de la formalisation du langage Java, Igarashi *et al.* [IPW01] ont défini un noyau minimal, appelé *Featherweight Java* (*FJ*), pour la modélisation du système de typage. Le but de FJ était de fournir un cadre de modélisation de certains aspects de conception du langage Java en se concentrant sur la définition et la preuve précises des propriétés. FJ, qui favorise la compacité sur l'obsession de travailler sur une définition complète du langage, considère seulement cinq expressions : la création d'objets, l'appel de méthodes, l'accès aux champs, la coercition d'objets et les variables. En fait, il n'y a pas d'affectation de variables, ce qui permet uniquement la manipulation d'objets immutables, c'est-à-dire que la valeur des champs des objets reste inchangée tout au long de l'exécution du programme. Dans le cadre de l'évaluation partielle de langages à objets, Schultz [Sch00] a utilisé une version étendue de FJ, appelée *Extended Featherweight Java* (EFJ), dans laquelle des constructions telles que l'expression conditionnelle, des opérateurs binaires et les types primitifs `boolean` et `int` ont été ajoutées afin d'améliorer l'expressivité du langage et donc de faciliter l'écriture des exemples. En effet, l'évaluation partielle des objets, pour obtenir des objets partiellement statiques et dynamiques, est possible grâce à l'inclusion de types primitifs.

Concrètement, le choix d'un langage comme FJ permet de se concentrer sur la description du processus de spécialisation en prenant en compte les aspects intrinsèques des langages à objets, par exemple, l'encapsulation des données et des méthodes ainsi que la relation d'héritage.

5.3.1 Syntaxe

La syntaxe de EFJ est montrée dans la figure 5.12. Un programme est une collection de classes et une expression initiale (*main*). Une classe, pouvant étendre une superclasse, englobe la déclaration d'un constructeur, de champs et de méthodes. Le constructeur appelle d'abord le constructeur de la superclasse puis initialise les champs déclarés dans la classe. Comme en FJ, l'affectation des champs peut avoir lieu seulement dans le constructeur, et la forme du constructeur est donnée par l'ensemble des champs de la classe et ceux de la superclasse. Le type des champs peut être un type primitif ou une classe. La déclaration d'une méthode inclut les paramètres formels et le corps, qui consiste en une expression simple. Une expression peut être une constante, une variable, la référence à un objet, l'affectation d'un champ, l'accès à un champ, l'instanciation d'un objet, la coercition d'un objet (*casting*), une opération binaire, une opération unaire, une conditionnelle ou un appel de méthode.

Dans les définitions syntaxiques de EFJ nous utilisons la police `typewriter` pour dénoter les mots clés des programmes (par exemple, `class`, `extends`). Les métavariables utilisées sont les suivantes : t pour les types ; C et D pour la définition des classes ; X pour la définition des variables que dans le cas de FJ il s'agit d'un champ ou d'un paramètre de méthode, ; E pour les expressions ; K pour la définition du seul constructeur de la classe ; M pour les définitions des méthodes ; k pour les constantes ; op_b et op_u pour les opérateurs binaires et unaires respectivement.

$$
\begin{array}{llll}
P & \in & Program & ::= \quad \overline{C}\ t\ E \\
C & \in & Class & ::= \quad \texttt{class}\ c\ \texttt{extends}\ c\ \{\overline{X;}\ K\ \overline{M}\} \\
t & \in & Type & ::= \quad p \mid c \\
p & \in & PrimitveType & ::= \quad \texttt{int} \mid \texttt{boolean} \\
c & \in & \mathcal{C} & \\
X & \in & VariableDefinition & ::= \quad t\ x \\
K & \in & Constructor & ::= \quad c(\overline{t_n\ x_n})\{ \\
 & & & \qquad \texttt{super}(x_1,\ldots,x_i); \\
 & & & \qquad \texttt{this}.x_{i+1} = x_{i+1}; \\
 & & & \qquad \ldots; \\
 & & & \qquad \texttt{this}.x_n = x_n; \\
 & & & \} \\
M & \in & Method & ::= \quad t\ m(\overline{X,})\{\texttt{return}\ E;\} \\
E & \in & Expression & ::= \quad k \mid x \mid \texttt{this} \mid \texttt{this}.x = x \mid E.x \mid \texttt{new}\ d(\overline{E}) \mid (d)E \\
 & & & \qquad \mid E\ op_b\ E \mid op_u\ E \mid E\texttt{?}E\texttt{:}E \mid E.m(\overline{E}) \\
op_b & \in & Operator & ::= \quad \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<} \mid \texttt{>} \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \\
op_u & \in & Operator & ::= \quad \texttt{++} \mid \texttt{--} \mid \texttt{-} \mid \texttt{!} \\
k & \in & Constant & ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{0} \mid \texttt{1} \mid \texttt{-1} \mid \ldots
\end{array}
$$

FIG. 5.12 – EFJ : Syntaxe des programmes.

En ce qui concerne les identificateurs des classes, méthodes, champs et paramètres, ils sont représentés en italique et en minuscule, à savoir c pour le nom de la classe dont la définition est représentée par C, m pour le nom de la méthode dont le déclaration est représentée par M et x pour les champs ou les paramètres définis par X. Dans ce dernier cas, afin de simplifier la compréhension des définitions, on utilise aussi f pour les champs en priorisant l'utilisation de la notation x pour les paramètres.

Nous écrivons $\overline{X}$ pour la séquence éventuellement vide $X_1 \ldots X_n$ (de même pour M), $\overline{X,}$ pour la séquence éventuellement vide $X_1\texttt{,}\ \ldots X_n\texttt{,}$ (de même pour E) et $\overline{X;}$ pour la séquence éventuellement vide $X_1\texttt{;}\ \ldots X_n\texttt{;}$. Dans le cas où il est nécessaire de préciser la taille de la séquence, on conserve l'indice supérieur, par exemple $\overline{X_n}$ pour $X_1 \ldots X_n$.

Les définitions décrites dans la figure 5.13 sont utilisées pour l'extraction d'information sur les classes, champs et méthodes d'un programme EFJ. En ce qui concerne les classes, la fonction *def* retourne la définition de la classe tandis que la fonction *superClass* retourne le définition de la superclasse.

Dans les programmes Java, il est possible de surcharger (*overloading* ou *hiding*) un champ dans une classe en déclarant simplement un nouveau champ avec le même nom qu'un champ hérité. Par définition, en FJ, la signature du constructeur associé à une classe est donné par l'ensemble des champs déclarés dans la classe plus les champs hérités des superclasses qui sont initialisés en appelant le constructeur de la superclasse (*i.e.*, `super`). Dans ce contexte, l'appel imbriqué des super-constructeurs peut rendre explicite

Classes :

$$def(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\_\} \qquad \frac{\texttt{class}\ d\ \texttt{extends}\ \_\{\_\} \qquad \texttt{class}\ c\ \texttt{extends}\ d\ \{\_\}}{superClass(c) = d}$$

Champs :

$$fields(\texttt{Object}) = \emptyset \qquad \frac{\texttt{class}\ c\ \texttt{extends}\ d\ \{\_\ \overline{X}\ \_\} \qquad superClass(c) = d}{fields(c) = \overline{X}\ \cup\ fields(d)}$$

Méthodes :

$$methods(\texttt{Object}) = \emptyset \qquad \frac{\texttt{class}\ c\ \texttt{extends}\ d\ \{\_\ \overline{M}\_\ \} \qquad superClass(c) = d}{methods(c) = \overline{M} \cup \{Mt \mid \forall\ Mt\ \in\ methods(d),\ \neg overridden(mt, c)\}}$$

$$\frac{M\ \in\ methods(c) \qquad M = t\ \ m(\overline{t\ x},)\{\texttt{return}\ E;\}}{mtype(m,c) = \overline{t} \to t \qquad mbody(m,c) = ((\overline{x}), E)}$$

$$\frac{mtype(m,c) = \overline{t} \to t \qquad \overline{t} = \overline{e} \qquad t = e}{override(m, c, \overline{e} \to e)}$$

FIG. 5.13 – EFJ : Définitions auxiliaires.

l'initialisation d'un champ d'une superclasse qui en réalité est utilisé implicitement par les instances des sous-classes. Par conséquent, afin de ne pas compliquer inutilement l'explication de l'analyse nous considérons des programmes où la surcharge des champs est interdite. Les champs d'une classe, calculés par la fonction *fields*, sont tous les champs déclarés dans la classe plus tous les champs hérités des superclasses dont la classe racine *Object* ne contient aucune définition de champ.

Pour l'extraction d'information de la déclaration d'une méthode on utilise les fonctions *mtype* et *mbody* pour obtenir le type et le corps d'une méthode donnée, respectivement. La redéfinition d'une méthode dans une sous-classe (*method overriding*) est considérée correcte si elle partage le même nom et type de la déclaration que la méthode originale (*root declaration*), comme l'indique la fonction *override*. L'ensemble des méthodes d'une classe est alors constitué par toutes les méthodes déclarées dans la classe ainsi que les méthodes héritées non redéfinies. Cet ensemble est calculé par la fonction *methods*.

Notation. Afin de simplifier la description des règles de bonne annotation (développées plus tard dans ce document) quand il n'existe aucune ambiguïté dans la identification des classes ni de ses membres ils seront identifiés par leur nom syntaxique. Dans ce contexte, le champ $t\ x$ défini dans la classe C, dont est dénoté par $c.x$; le paramètre t `x` la méthode M définie dans la classe C est dénoté par $c.m.x$; la valeur de la méthode de retour est identifié par $c.m.$`return`.

5.3.1.1 Considérations pour l'analyse

Pour conserver la précision dans le résultat de l'inférence de types concrets, l'approche proposée considère que les méthodes et les champs des classes du programme analysé ont été répliqués dans les sous-classes correspondantes. Dans le premier cas, la réplication a lieu sauf si la méthode a été redéfinie comme l'indique la fonction *methods* dans la figure 5.13. Cette approche a été développée dans le contexte d'un langage dynamiquement typé, en fait un sous-ensemble de SmallTalk, par Oxhøj *et al.* [OPS92]. La réplication est effectuée à travers une transformation syntaxique du code source du programme. Cette approche permet d'associer comme type de la référence à l'objet, `this`, la classe englobante de la méthode correspondante car dans ce contexte la relation d'héritage est effectivement supprimée grâce à la réplication.

Dans un langage comme Java, implémenter une transformation comme expliquée ci-dessus est possible mais compliqué. Comme le but de cette section est de montrer une nouvelle approche pour la spécialisation de programmes à objets nous préférons, pour l'instant, éviter la discussion sur les transformations applicables sur un programme afin de faciliter l'analyse. Pour cette raison, dans notre approche, la réplication est effectuée virtuellement à travers les fonctions *methods* et $fields$ (voir figure 5.13).

5.3.2 Un exemple : programme *Point2DStepping*

Nous détaillons maintenant l'exemple utilisé précédemment pour illustrer l'idée de l'analyse proposée dans ce chapitre. Le programme *Point2DStepping*, décrit par le diagramme de classes de la figure 5.14, permet la manipulation de points à deux dimensions par l'instanciation de la classe *Point2D*. La classe *Point2DX* modélise aussi un point à deux dimensions mais composé, en plus, avec un objet de type *Stepper* qui permet de calculer le point suivant à l'aide de la méthode *next*. La classe *Point2D2* est une sous-classe de la classe *Point2DX* dont le pas retourné par la méthode *dec* duplique le pas simple avant d'ajouter le décalage spécifié par le paramètre *offset*. En ce qui concerne le pas, il est modélisé par les classe *Stepper* et *StepperBy2*. Dans le premier cas, la méthode *step* retourne simplement la valeur de la variable *step* tandis que la méthode *step* de la sous-classe *StepperBy2* duplique la valeur de cette variable.

Les figures 5.15 montrent l'implémentation du programme *Point2DStepping*. Il est à noter que, l'expressivité du langage étant limitée par la définition de EFJ, il n'est pas possible de réutiliser des méthodes des superclasses. Concrètement, la méthode *step* de la classe *StepperBy2* pourrait être redéfinie en Java en appelant la méthode de la superclasse mais, comme mentionnée auparavant, EFJ ne permet pas de référencer les membres dans la superclasse à travers le mot clé *super*.

L'intérêt d'un programme comme celui-ci réside, en particulier, dans la définition de méthodes qui retournent des valeurs de type objet ainsi que dans la possibilité de réplication et de redéfinition des méthodes. Nous verrons aussi comment l'approche choisie permet de traiter séparément l'utilisation d'un membre dans la classe qui le définit et donc les classes qui l'héritent. Par exemple, le champ *step* définie dans la classe *Stepper* sera analysée séparément, c'est-à-dire dans le contexte de la classe *Stepper* et dans le

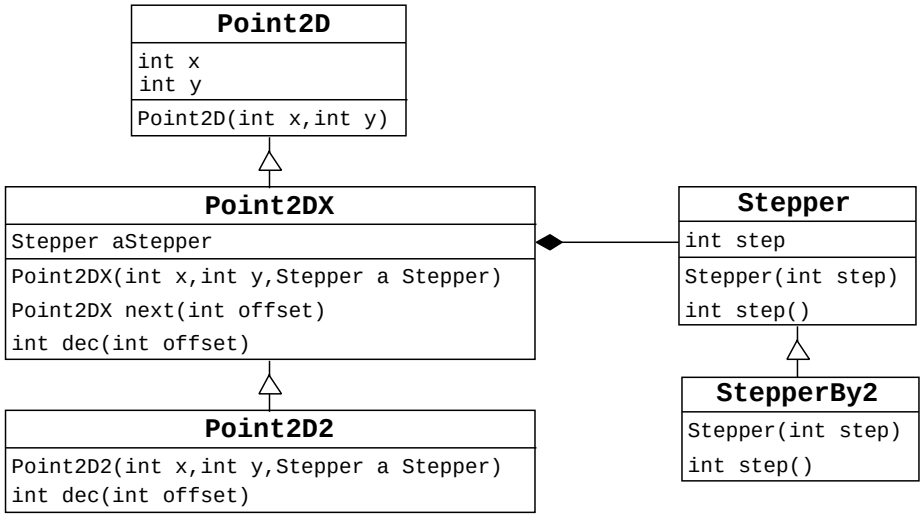


FIG. 5.14 – Exemple EFJ : Programme *Point2DStepping*

contexte de la classe $StepperBy2$.

5.3.3 Constructions annotables

Les constructions de programmes EFJ susceptibles d'être annotées sont évidemment celles dont la déclaration est associée à un type. La syntaxe de EFJ (voir figure 5.12), montre que les constructions liées aux types sont les champs et les méthodes. Ce derniere à travers leurs paramètres et l'instruction de retour ($mretnStm()$). De manière générale, les *constructions annotables* sont celles qui représentent des emplacements dans la mémoire. Les expressions sont aussi des constructions annotables simplement parce que elles peuvent être affectées aux constructions mentionnées auparavant.

L'ensemble des constructions annotables, C_A, d'un programme P est défini comme suit :

$$C_A = X \cup E \cup M$$

5.3.4 Règles de types déclarés

EFJ étant, comme FJ, un langage statiquement typé, il est soumis à un ensemble de règles de typage qui permettrent de déterminer si un programme EFJ est bien typé. Les règles de typage garantissent que l'évaluation du programme retourne une valeur, c'est-à-dire une constante ou un objet, s'arrête du fait de la cœrcition incorrecte d'un objet, c'est-à-dire incompatibilité entre le type concret de l'objet et le type cible, ou diverge (par exemple, condition d'une boucle non atteinte) [GJSB00].

Les règles sont définies sur la base des règles de sous-typage de la figure 5.16, où $<:$ dénote la relation de sous-typage ($c <: d$ indique que c est un sous-type de d).

```
class Point2D{
 int x;
 int y;
 Point2D(int x, int y){
  this.x = x;
  this.y = y;
 }
}
class Point2DX extends Point2D{
 Stepper aStepper;
 Point2DX(int x,
          int y,
          Stepper aStepper){
  super(x, y);
  this.aStepper = aStepper;
 }
 Point2DX next(int offset){
  return new Point2DX(
              this.x + this.dec(offset),
              this.y + this.dec(offset),
              this.aStepper);
 }
 int dec(int offset){
  return this.aStepper.step() + offset;
 }
}
class Point2D2 extends Point2D {
 Point2D2(int x, int y, Stepper aStepper){
  super(x, y, aStepper);
 }
 int dec(int offset){
  return this.aStepper.step() * 2 + offset;
 }
}
class Stepper {
 int step;
 Stepper(int step){
  this.step = step;
 }
 int step() {
  return this.step;
 }
}
class StepperBy2 extends Stepper {
 StepperBy2(int step){
  super(step);
 }
 int step() {
  return this.step * 2;
 }
}
```

FIG. 5.15 – Exemple EFJ : Implémentation du programme *Point2DStepping*.

Les règles de type déclaré sur les constructions de programmes EFJ sont définies dans la figure 5.17. On utilise les opérateur infixes :: et $\in$ pour dénoter respectivement la concaténation de séquences et l'appartenance à une séquence. On distingue deux formes de jugements : des jugements $\Gamma^{dt} \vdash E : e$ qui s'appliquent aux expressions (E) et qui comportent donc un type (e), et des jugements $\Gamma^{dt} \vdash C_A : OK$ qui s'appliquent aux programmes, méthodes et classes et indiquent simplement que ces constructions C_A sont bien typées.

Dans les régles, un jugement sur une séquence $\Gamma^{dt} \vdash \overline{E} : \overline{e}$ est un raccourci pour une séquence de jugements $\Gamma^{dt} \vdash E_1 : e_1, ..., \Gamma^{dt} \vdash E_n : e_n$. De même, nous écrivons $\overline{e} <: \overline{t}$ pour une séquence $e_1 <: t_1, \ldots, e_n <: t_n$.

Expressions. Le type déclaré des constantes se calcule en regardant le domaine de leur valeur (règles $\mathbf{INT}^{dt}$ et $\mathbf{BOOL}^{dt}$) tandis que le type de l'expression qui représente la référence à l'objet (`this`) est donné par l'environnement Γ^{dt} (règle $\mathbf{THIS}^{dt}$). En EFJ, la seule forme de variable est le paramètre d'une méthode, en conséquence, le type d'une variable est donné par le type déclaré du paramètre correspondant selon l'environnement Γ^{dt} (règle $\mathbf{VAR}^{dt}$). Le type de l'accès à un champ dépend du type de l'objet receveur

$$c <: c \quad [\mathbf{ID}^s] \qquad \frac{c <: d \qquad d <: b}{c <: b} \quad [\mathbf{TRANS}^s] \qquad \frac{superClass(c) = c}{c <: c} \quad [\mathbf{CLASS}^s]$$

FIG. 5.16 – EFJ : Relation de sous-typage.

(*i.e.*, E) (règle $\mathbf{FIELD}^{dt}$). Lors de la création d'une instance, les expressions associées aux arguments doivent vérifier la relation de sous-typage avec les paramètres du constructeur correspondant (règle $\mathbf{NEW}^{dt}$). La cœrcition d'un objet doit être possible soit en avant (*forward*) soit en arrière (*backward*) par rapport à la hiérarchie de classes (règle $\mathbf{CAST}^{dt}$). Un opérateur (unaire ou binaire) relie des valeurs de type primitif avec une valeur de type primitif (règles $\mathbf{OB\text{-}I\text{-}I}^{dt}$, $\mathbf{OB\text{-}I\text{-}B}^{dt}$, $\mathbf{OB\text{-}B\text{-}B}^{dt}$, $\mathbf{OU\text{-}I}^{dt}$ et $\mathbf{OB\text{-}B}^{dt}$). L'expression conditionnelle est formée par une expression booléenne pour le test et deux expressions résultant de l'évaluation de l'expression dont les types associés sont T_2 et T_3. En l'absence du résultat de l'évaluation de l'expression du test (*i.e.*, e_1), la règle peut seulement dire que le type de l'expression sera le plus petit supertype commun (ici calculé par la fonction *stc*) des sous-expressions e_2 et e_3 (règles $\mathbf{COND}^{dt}$). L'appel d'une méthode est bien typé si les arguments et l'expression de retour sont bien typés pour chaque méthode m définie dans les classes représentées par l'expression du receveur (*i.e.*, E).

Méthodes. Une méthode est bien typée par rapport à une classe c, si le type de l'expression de retour est un sous-type de le type déclaré de retour de la méthode (règle $\mathbf{MTHD}^{dt}$).

Classes. Une classe est bien typée si le constructeur et toutes les méthodes de la classe sont bien typées (règle $\mathbf{CLS}^{dt}$).

Programmes. Enfin, un programme est bien typé si toutes les classes sont bien typées ainsi que l'expression initiale (règle $\mathbf{PRG}^{dt}$).

5.4 Bilan

Dans cette section, nous avons abordé la problématique associée à la spécialisation de programmes dans les langages à objet, principalement l'héritage et le polymorphisme. Dans le premier cas nous montrons les aspects qui rendrent différent la spécialisation à travers la relation d'héritage, où la classe spécialisée (sous-classes) est un version étendue de la classe original, et la spécialisation au sens de l'évaluation partielle, où la classe spécialisée est obtenue à travers la réduction de la description de la classe originale. Dans le deuxième cas, nous expliquons la problématique associée aux langages dont l'utilisation du mécanisme de polymorphisme fait plus complexe l'analyse de flot de contrôle.

Afin de résoudre un de ces problèmes, concrètement celui associé au polymorphisme, nous proposons une approche de spécialisation de programmes qui inclut une étape d'ana-

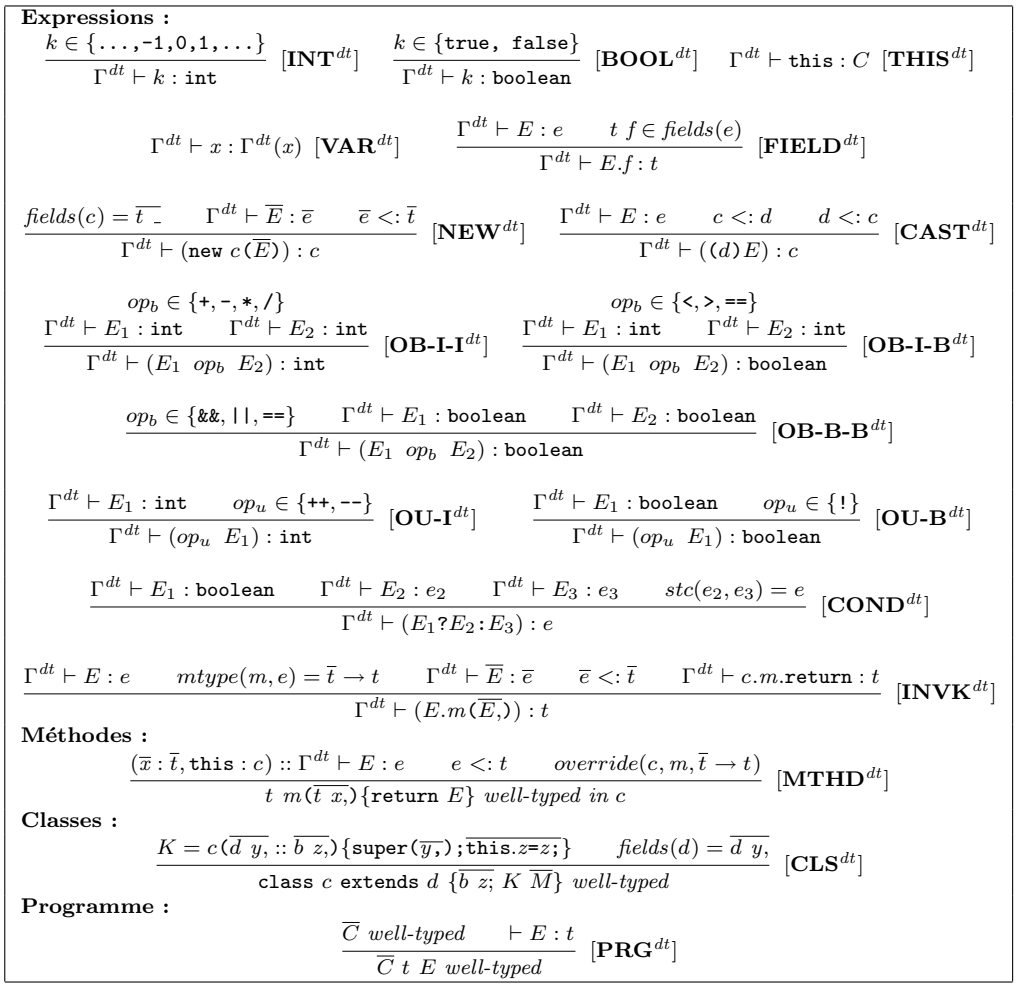


FIG. 5.17 – Type déclaré : règles sur les programmes EFJ.

lyse de types concrets. Cela permet améliorer le processus de spécialisation en calculant de manière plus précise le flot de contrôle des programmes analysés.

Finalement, nous présentons le langage à objets EFJ qui est le langage d'implémentation choisi pour illustrer et formaliser l'approche de spécialisation dans le langage à objets proposée.

Chapitre 6

Analyse et spécialisation

Sommaire

6.1 Analyse de types concrets **121**
6.1.1 Règles de bonne annotation de type concret 123
6.1.2 Analyse par contraintes 125
6.2 Analyse de temps de liaison **134**
6.2.1 Règles de bonne annotation de temps de liaison 136
6.2.2 Analyse par contraintes 138
6.3 Analyse de temps d'évaluation **145**
6.3.1 Règles de bonne annotation de temps d'évaluation 147
6.3.2 Analyse par contraintes 151
6.4 Spécialisation **159**
6.4.1 Les extensions génératrices 159
6.4.2 Générateur d'extensions génératrices pour EFJ 166
6.5 Bilan **168**

Ce chapitre détaille la spécialisation d'un programme EFJ. Cette spécialisation s'effectue en deux étapes : une étape d'analyse et une étape de génération d'extensions génératrices. L'étape d'analyse comporte une analyse de types concrets (section 6.1), une analyse de temps de liaison (section 6.2) et une analyse de temps d'évaluation (section 6.3). Ces analyses sont à la fois décrites précisément sous la forme d'une analyse par contraintes et illustrées sur le programme *Point2DStepping* (voir section 5.3.2). L'exécution de l'extension génératrice conduit à l'obtention du programme spécialisé. L'étape de génération d'extensions génératrices est décrite dans la section 6.4.

6.1 Analyse de types concrets

Partant de l'hypothèse qu'un programme est bien typé, on a la certitude que toutes les constructions du programme seront associées, durant l'exécution, à des valeurs de types compatibles avec les types déclarés. L'inférence des types concrets par analyse statique permet de connaître l'ensemble des types concrets d'un objet. L'idée sous-jacente

est d'obtenir une approximation du programme, en ce qui concerne les types concrets des constructions, en raisonnant sur l'ensemble des valeurs associées à chaque construction pendant l'exécution. En partant des règles de type déclaré on obtient un ensemble de règles tel que l'environnement reliant les constructions avec leurs types déclarés est systématiquement remplacé par un nouvel environnement reliant les constructions avec des ensembles de types possibles (concrets), en fait une approximation des types concrets. Pour ceci, la relation de sous-typage ($<:$) est remplacée, dans les règles de type déclaré, par la relation ensembliste d'inclusion ($\subseteq$) (voir section 6.1.1).

Domaine des annotations. L'environnement Γ^{dt} des types déclarés sur des constructions annotables (C_A) est remplacé par l'environnement Γ^{ct} reliant les constructions avec l'annotation de type concret : un ensemble de types dénoté par t^{ct}. Les annotations de type concret pour un élément c_a de C_A dépendront du type déclaré de c_a, c'est-à-dire $\Gamma^{dt}(c_a)$. En conséquence, soit le programme P et l'ensemble C_A des constructions annotables du programme P, nous définissons l'environnement Γ^{ct} de la façon suivante :

$$\text{Soit } \mathcal{T} : \{\texttt{int}, \texttt{boolean}\} \cup \mathcal{C} \text{ et } \Gamma^{ct} : C_A \rightarrow \mathcal{P}(\mathcal{T}) \text{ tel que}$$

$$\Gamma^{ct}(c_a) \subset \{t / t \in \mathcal{T}, t <: \Gamma^{dt}(c_a)\} \tag{6.1}$$

Comme le sous-typage est par définition une relation d'ordre partiel (voir [Cas97, Pie02]) l'image de la fonction $\Gamma^{ct}(c_a)$ est un sous-ensemble de $\mathcal{P}(\mathcal{T})$. En effet, l'ensemble des annotations possibles sur un programme donné est défini par l'élimination de tous les éléments (ensembles) de $\mathcal{P}(\mathcal{T})$ contenant des types incompatibles par rapport à la relation de sous-typage (voir figure 5.16).

Par exemple, soit le programme :

$$\begin{array}{lcl} \mathit{Point2DStepping} & = & \mathit{Point2D}\ \mathit{Point2DX}\ \mathit{Point2D2} \\ & & \mathit{Stepper}\ \mathit{StepperBy2}\ \texttt{int}\ E_{main} \end{array}$$

avec la relation de sous-typage décrite par le diagramme de la figure 5.14 et pour lequel l'ensemble *Type* de types du programme *Point2DStepping* est :

$$\begin{array}{lcl} \mathcal{T} & = & \{\texttt{int}, \texttt{boolean}\} \cup \\ & & \{\texttt{Point2D}, \texttt{Point2DX}, \texttt{Point2D2}, \texttt{Stepper}, \texttt{StepperBy2}\} \end{array}$$

Alors, les annotations possibles des types concrets du programme *Point2DStepping* sont :

$$\begin{array}{lcl} cod_{\Gamma^{ct}(\mathit{Point2DStepping})} = & & \{\ \{\}, \\ & & \{\texttt{int}\}, \{\texttt{boolean}\}, \\ & & \{\texttt{Point2D}, \texttt{Point2DX}, \texttt{Point2D2}\}, \\ & & \{\texttt{Point2D}, \texttt{Point2DX}\}, \{\texttt{Point2DX}, \texttt{Point2D2}\}, \\ & & \{\texttt{Point2D}\}, \{\texttt{Point2DX}\}, \{\texttt{Point2D2}\}, \\ & & \{\texttt{Stepper}, \texttt{StepperBy2}\}, \\ & & \{\texttt{Stepper}\}, \{\texttt{StepperBy2}\}\} \end{array}$$

6.1.1 Règles de bonne annotation de type concret

Les règles de bonne annotation de typage concret définissent la correction du typage concret d'un programme par rapport à un environnement de typage concret initial Γ_0^{ct} qui associe à chaque construction annotable du programme une annotation de typage concret (un ensemble de types concrets). Notons que le programme est supposé bien typé en terme de type déclaré. Une règle générale de correction du typage concret par rapport au typage déclaré est que tout type concret d'une expression est un sous-type (au sens large) du type déclaré de l'expression. Nos règles de bonne annotation ne prennent pas ce point en compte explicitement. Toutefois, la définition des types concrets des constantes et des instanciations, ainsi que l'inclusion systématique de l'ensemble des types concrets de la source dans l'ensemble des types concrets de la destination lors d'une affectation garantissent que cette règle générale est respectée.

Dans cette section nous abordons les règles qui permettent de déterminer si les annotations sont correctes (*well-ct-annotated*). Ces règles, détaillées dans la figure 6.1, sont dérivées des règles de types déclarés (voir section 5.3.4) et basées sur l'environnement Γ^{ct} tel que le jugement $\Gamma^{ct} \vdash E : t^{ct}$ indique que l'annotation de type concret de l'expression E est t^{ct}.

Expressions. Le type concret des constantes, entier ou booléen, est calculé à partir du type déclaré lié par l'environnent Γ^{dt} (règles $\textsc{int}^{ct}$ et $\textsc{bool}^{ct}$). La règle $\textsc{var}^{ct}$ exprime le fait que les type concrets d'une variable sont directement donnés par l'environnement de typage courant (qui est constitué de l'environnement de typage initial Γ_0^{ct}, étendu avec le typage de `this`, voir la règle pour les méthodes).

Pour un accès à un champ $E_0.f$, l'annotation de l'expression inclut au moins tous les types concrets des champs f accessibles au travers des types concrets de la sous-expression E_0 (règle $\textsc{field}^{ct}$).

Le seul type concret possible d'une instanciation est la classe dont est issue l'instance (règle $\textsc{new}^{ct}$). De plus, les annotations des paramètres du constructeur sont incluses dans les annotations des champs de la classe. C'est une traduction directe de l'affectation des paramètres actuels de l'instantiation aux paramètres formels du constructeur.

Un programme spécialisé doit avoir un comportement similaire au programme original. Dans ce contexte, si un programme lance une exception à cause d'une cœrcition incorrecte, le programme spécialisé doit réagir de la même façon. Concrètement, dans le cas où l'expression de coercition est spécialisée, c'est-à-dire qu'elle n'est pas résidualisée, les conditions d'exécution originelles du programme ne peuvent pas être assurées. Pour cette raison, nous définissons l'annotation de cette expression comme l'ensemble des types de l'expression de la cœrcition qui sont un sous-type du type cible (règle $\textsc{cast}^{ct}$).

Le type concret des opérations binaires est calculé directement à partir des règles des types déclarés du fait de l'absence de sous-typage entre les types primitifs. Les types déclarés des expressions sont remplacés par l'ensemble singleton du type déclaré (règles $\textsc{ob-i-i}^{ct}$, $\textsc{ob-i-b}^{ct}$ et $\textsc{ob-b-b}^{ct}$). La même procédure est appliquée pour les opérations unaires (règles $\textsc{ou-i}^{ct}$ et $\textsc{ou-b}^{ct}$).

Dans le cas de l'expression conditionnelle, tout comme dans le cas des règles de

Expressions :

$$\frac{\Gamma^{dt} \vdash k : \texttt{int}}{\Gamma^{ct} \vdash k : \{\texttt{int}\}}\ [\mathbf{INT}^{ct}] \qquad \frac{\Gamma^{dt} \vdash k : \texttt{boolean}}{\Gamma^{ct} \vdash k : \{\texttt{boolean}\}}\ [\mathbf{BOOL}^{ct}] \qquad \Gamma^{ct} \vdash \texttt{this} : \{c\}\ \ [\mathbf{THIS}^{ct}]$$

$$\Gamma^{ct} \vdash x : \Gamma^{ct}(x)\ \ [\mathbf{VAR}^{ct}] \qquad \frac{\Gamma^{ct} \vdash E_0 : t^{ct}_{E_0} \qquad t^{ct}_E \supseteq \bigcup_{c\ \in\ t^{ct}_{E_0}} \Gamma^{ct}(c.f)}{\Gamma^{ct} \vdash (E_0.f) : t^{ct}_E}\ [\mathbf{FIELD}^{ct}]$$

$$\frac{\mathit{fields}(c) = \overline{\_\ f} \qquad \Gamma^{ct} \vdash \overline{E} : \overline{t^{ct}_E} \qquad \overline{t^{ct}_E} \subseteq \overline{\Gamma^{ct}(c.f)}}{\Gamma^{ct} \vdash (\texttt{new}\ c(\overline{E})) : \{c\}}\ [\mathbf{NEW}^{ct}] \qquad \frac{\Gamma^{ct} \vdash E : t^{ct}_{E_0} \qquad t^{ct}_E = \{b \in t^{ct}_{E_0} \mid b <: d\}}{\Gamma^{ct} \vdash ((d)\,E) : t^{ct}_E}\ [\mathbf{CAST}^{ct}]$$

$$\frac{op_b \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\} \qquad \Gamma^{ct} \vdash E_1 : \{\texttt{int}\} \qquad \Gamma^{ct} \vdash E_2 : \{\texttt{int}\}}{\Gamma^{ct} \vdash (E_1\ \ op_b\ \ E_2) : \{\texttt{int}\}}\ [\mathbf{OB\text{-}I\text{-}I}^{ct}] \qquad \frac{op_b \in \{\texttt{<}, \texttt{>}, \texttt{==}\} \qquad \Gamma^{ct} \vdash E_1 : \{\texttt{int}\} \qquad \Gamma^{ct} \vdash E_2 : \{\texttt{int}\}}{\Gamma^{ct} \vdash (E_1\ \ op_b\ \ E_2) : \{\texttt{boolean}\}}\ [\mathbf{OB\text{-}I\text{-}B}^{ct}]$$

$$\frac{op_b \in \{\texttt{\&\&}, \texttt{||}, \texttt{==}\} \qquad \Gamma^{ct} \vdash E_1 : \{\texttt{boolean}\} \qquad \Gamma^{ct} \vdash E_2 : \{\texttt{boolean}\}}{\Gamma^{ct} \vdash (E_1\ \ op_b\ \ E_2) : \{\texttt{boolean}\}}\ [\mathbf{OB\text{-}B\text{-}B}^{ct}]$$

$$\frac{\Gamma^{ct} \vdash E_1 : \{\texttt{int}\} \qquad op_u \in \{\texttt{++}, \texttt{--}\}}{\Gamma^{ct} \vdash (op_u\ \ E_1) : \{\texttt{int}\}}\ [\mathbf{OU\text{-}I}^{ct}] \qquad \frac{\Gamma^{ct} \vdash E_1 : \{\texttt{boolean}\} \qquad op_u \in \{\texttt{!}\}}{\Gamma^{ct} \vdash (op_u\ \ E_1) : \{\texttt{boolean}\}}\ [\mathbf{OU\text{-}B}^{ct}]$$

$$\frac{\Gamma^{ct} \vdash E_1 : \{\texttt{boolean}\} \qquad \Gamma^{ct} \vdash E_2 : t^{ct}_{E_2} \qquad \Gamma^{ct} \vdash E_3 : t^{ct}_{E_3} \qquad t^{ct}_E \supseteq t^{ct}_{E_2} \qquad t^{ct}_E \supseteq t^{ct}_{E_3}}{\Gamma^{ct} \vdash (E_1\texttt{?}E_2\texttt{:}E_3) : t^{ct}_E}\ [\mathbf{COND}^{ct}]$$

$$\frac{\Gamma^{ct} \vdash E_0 : t^{ct}_{E_0} \qquad \Gamma^{ct} \vdash \overline{E} : \overline{t^{ct}_x} \qquad \forall\ c\ \in\ t^{ct}_{E_0}\ (\overline{t^{ct}_x} \subseteq \Gamma^{ct}(c.m.x),\ t^{ct}_E \supseteq \Gamma^{ct}(c.m.\texttt{return}))}{\Gamma^{ct} \vdash (E_0.m(\overline{E},)) : t^{ct}_E}\ [\mathbf{INVK}^{ct}]$$

Méthodes :

$$\frac{(\texttt{this} : \{c\}) :: \Gamma^{ct} \vdash E : t^{ct}_E \qquad t^{ct}_E \subseteq \Gamma^{ct}(c.m.\texttt{return})}{\Gamma^{ct} \vdash t\ \ m(\overline{t\ x},)\{\texttt{return}\ E\}\ \textit{well-ct-annotated in } c}\ [\mathbf{MTHD}^{ct}]$$

Classes :

$$\frac{\Gamma^{ct} \vdash \overline{M}\ \textit{well-ct-annotated in } c}{\Gamma^{ct} \vdash \texttt{class}\ c\ \texttt{extends}\ \_\ \{\_\ \_\ \overline{M}\}\ \textit{well-ct-annotated}}\ [\mathbf{CLS}^{ct}]$$

Programme :

$$\frac{\Gamma^{ct}_0(\overline{x}) : \overline{t} \qquad \Gamma^{ct}_0 \vdash \overline{C}\ \textit{well-ct-annotated} \qquad \Gamma^{ct}_0 \vdash E : t}{\Gamma^{ct}_0 \vdash \overline{C}\ t\ E\ \overline{t\ x}\ \textit{well-ct-annotated}}\ [\mathbf{PRG}^{ct}]$$

FIG. 6.1 – Type concret : règles sur les programmes EFJ.

type déclaré, l'annotation dépend de la relation existant entre les annotations des sous-expressions E_2 et E_3. L'annotation de l'expression est donnée par l'union des annotations des sous-expressions. Autrement dit, tous les types concernant l'annotation de l'expression E_2 devront être compris dans l'annotation de l'expression conditionnelle, de même pour l'expression E_3.

Dans le cas d'une invocation d'une méthode m, on considère le type concret du receveur. Pour chaque classe possible c les annotations des paramètres de l'invocation sont incluses dans les annotations des paramètres formels de la définition de la méthode pour cette classe ($c.m$). De plus, l'annotation du retour pour la méthode $c.m.\texttt{return}$, est incluse dans l'annotation de l'expression.

Méthodes. Une méthode m est typée dans le contexte d'une classe c donnée, ce qui donne le type concret de `this`. L'annotation de l'expression retournée est incluse dans l'annotation de $c.m.$`return`. En fait, comme l'instruction `return` est unique, on pourrait aussi choisir d'exclure les typages où il n'y a pas égalité (règle $\textsc{mthd}^{ct}$).

Classes. Une classe est bien typée si ses méthodes sont bien typées dans son contexte. À partir du moment où les constructeurs sont syntaxiquement bien formés, ils sont bien typés (typage déclaré) et ne nécessitent pas de règle de bonne annotation particulière. Toute l'information est contenue dans les annotations des champs de leur classe ($\textsc{class}^{ct}$).

Programmes. Un programme est bien typé si ses classes ainsi que son expression principale sont bien typées par rapport à un environnement initial Γ_0^{ct} initialisé par les annotations sur les variables libres $\overline{t\ x}$ ($\textsc{prg}^{ct}$).

6.1.2 Analyse par contraintes

Dans cette section nous décrivons la définition, la génération et, enfin, la résolution des contraintes pour l'inférence de types concrets.

6.1.2.1 Définition des contraintes

Dans le contexte de l'inférence des types concrets étudié ici, les variables représentent les annotations associées aux constructions du programme, soit des ensembles de types. En particulier, nous appliquons l'approche de l'analyse par contraintes ensemblistes (*set-based constraint analysis*) pour générer l'ensemble des contraintes qui permettra d'effectuer l'analyse mentionnée [Hei92b].

Une variable de contrainte liée à la construction de programme c_a est dénotée par $V_{c_a}^{ct}$. Les variables sont donc de la forme suivante :

- $V_{c.x}^{ct}$: variable représentant l'annotation du champ libellé x de la class libellée c.
- $V_{c.m.x}^{ct}$: variable représentant l'annotation du paramètre libellé x de la méthode libellée m définie dans la classe libellée c.
- $V_{c.m.\texttt{return}}^{ct}$: variable représentant l'annotation du retour de la méthode libellée m définie dans la classe libellée c.
- V_E^{ct} et $V_{E_i}^{ct}$: variable représentant les annotations sur une expression E et celles sur les sous-expressions E_i, si nécessaire.

Notez que les variables correspondent à chacune des catégories de constructions annotables définies auparavant.

Dans le cas des variables de contraintes liées aux expressions, l'identificateur d'une sous-expression est construit à partir d'un index associé à l'expression englobante et préfixé par le nom de la méthode et de la classe auquel elle appartient. La figure 6.2 montre la méthode `dec` de la classe `Point2D2` annotée avec les variables de contraintes correspondantes.

Les contraintes sont de la forme $V_{c_a}^{ct} \supseteq en$ où en représente une expression ensembliste. La relation d'inclusion implique que l'ensemble en doit être inclus dans l'ensemble exprimé par la variable $V_{c_a}^{ct}$ [HJ94, CJ01]. La figure 6.3 montre la syntaxe utilisée pour la

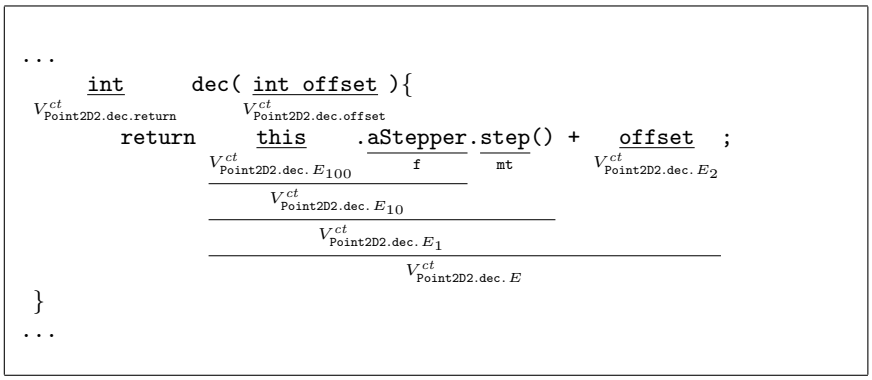


FIG. 6.2 – Variables de contraintes : annotation sur la méthode `Point2D2.dec`.

génération des contraintes où les opérateurs $fieldAccess^{ct}$ et $methodApp^{ct}$ sont utilisés pour la définition des contraintes conditionnelles (voir section 5.2.1.2). Ici, nous appelons les expressions ensemblistes définies par les opérateurs mentionnés des *expressions ensemblistes conditionnelles*.

$V^{ct}_{c_a}$	$\supseteq$	en	contrainte
en	::=	T	ensemble de types
	|	$V^{ct}_{c_a}$	variable de contrainte
	|	$fieldAccess^{ct}(V^{ct}_{c_{a_0}}, x)$	accès à un champ
	|	$methodApp^{ct}(V^{ct}_{c_{a_0}}, m, (V^{ct}_{c_{a_1}}, \ldots, V^{ct}_{c_{a_n}}))$	appel d'une méthode

FIG. 6.3 – Type concret : syntaxe des contraintes.

Sémantique des contraintes. La sémantique formelle des expressions ensemblistes décrite dans la figure 6.4 est définie par une interprétation $\mathcal{I}^{ct}$ qui lie les expressions ensemblistes en avec les ensembles de valeurs. L'interprétation $\mathcal{I}^{ct}$ est une solution de l'ensemble de contraintes $\mathcal{C}^{ct}$ si pour chaque contrainte $V^{ct}_{c_a} \supseteq en \in \mathcal{C}^{ct}$, $\mathcal{I}^{ct}(V^{ct}_{c_a}) \supseteq \mathcal{I}^{ct}(V^{ct}_{en})$.

$\mathcal{I}^{ct}(\bot)$	=	$\{\}$
$\mathcal{I}^{ct}(T)$	=	T
$\mathcal{I}^{ct}(V^{ct}_{c_a})$	$\subseteq$	$Type$
$\mathcal{I}^{ct}(fieldAccess^{ct}(V^{ct}_{c_{a_0}}, f))$	=	$\{\mathcal{I}^{ct}(V^{ct}_{c.f}) \mid c \in \mathcal{I}^{ct}(V^{ct}_{c_{a_0}}),\ \_\ f \in fields(c)\}$
$\mathcal{I}^{ct}(methodApp^{ct}(V^{ct}_{c_{a_0}}, m, (V^{ct}_{c_{a_1}}, \ldots, V^{ct}_{c_{a_n}})))$	=	$\{\mathcal{I}^{ct}(V^{ct}_{c.m.\texttt{return}}) \mid c \in \mathcal{I}^{ct}(V^{ct}_{c_{a_0}}),$ $\_\ m(\overline{\_\ x_n\ ,})\{\_\} \in methods(c)$ $\forall\ i \in [1,n]\ \mathcal{I}^{ct}(V^{ct}_{\texttt{c}.m.x_i}) \supseteq \mathcal{I}^{ct}(V^{ct}_{c_{a_i}})\}$
$\mathcal{I}^{ct}(\top)$	=	$Type$

FIG. 6.4 – Type concret : sémantique des contraintes.

Par exemple, l'expression $methodApp^{ct}(V^{ct}_{c_{a_0}}, m, (V^{ct}_{c_{a_1}}, \ldots, V^{ct}_{c_{a_n}}))$ représente les types de la valeur de retour des méthodes libellées m et définies dans les classes incluses dans l'ensemble $V^{ct}_{c_{a_0}}$ avec les arguments correspondants ($V^{ct}_{c_{a_1}}, \ldots, V^{ct}_{c_{a_n}}$). Ceci implique donc que les types concrets de la valeur de retour sont conditionnés par les types concrets du receveur.

6.1.2.2 Génération des contraintes

$C^{ct}_{Exp}(c, m, E)$	$=$	case E of
k	$\Rightarrow$	$\{V^{ct}_E = \{\Gamma^{dt}(k)\}\}$
`this`	$\Rightarrow$	$\{V^{ct}_E = \{c\}\}$
x	$\Rightarrow$	$\{V^{ct}_E = V^{ct}_{c.m.x}\}$
$E_0.x$	$\Rightarrow$	$\{V^{ct}_E = fieldAccess^{ct}(V^{ct}_{E_0}, x)\} \bigcup C^{ct}_{Exp}(c, m, E_0)$
`new` d`(`$E_1, \ldots, E_n$`)`	$\Rightarrow$	$\{V^{ct}_E = \{d\}\} \bigcup_{i\in[1,n]} \{V^{ct}_{d.f_i} \supseteq V^{ct}_{E_i}\} \bigcup_{i\in[1,n]} C^{ct}_{Exp}(c, m, E_i)$
`(`d`)`E_1	$\Rightarrow$	$\{V^{ct}_E = V^{ct}_{E_1}\} \bigcup C^{ct}_{Exp}(c, m, E_1)$
$E_1\ op_b\ E_2$	$\Rightarrow$	$(op_b \in \{$`+, -, /, *`$\}?\{V^{ct}_E = \{$`int`$\}\}:\{V^{ct}_E = \{$`boolean`$\}\})$ $\bigcup_{i\in[1,2]} C^{ct}_{Exp}(c, m, E_i)$
$op_u\ E_1$	$\Rightarrow$	$\{V^{ct}_E = V^{ct}_{E_1}\} \bigcup C^{ct}_{Exp}(c, m, E_1)$
E_1`?`E_2`:`E_3	$\Rightarrow$	$\{V^{ct}_E \supseteq V^{ct}_{E_2}, V^{ct}_E \supseteq V^{ct}_{E_3}\} \bigcup_{i\in[1,3]} C^{ct}_{Exp}(c, m, E_i)$
E_0`.`mt`(`$E_1, \ldots, E_n$`)`	$\Rightarrow$	$\{V^{ct}_E = methodApp^{ct}(V^{ct}_{E_0}, mt, (V^{ct}_{E_1}, \ldots, V^{ct}_{E_n}))\}$ $\bigcup_{i\in[0,n]} C^{ct}_{Exp}(c, m, E_i)$
$C^{ct}_{Mthd}(c, \_\ m$`(`$\overline{X,}$`){return` E`;})`	$=$	$\{V^{ct}_{c.m.\texttt{return}} \supseteq V^{ct}_E\} \bigcup C^{ct}_{Exp}(c, m, E)$
C^{ct}_{Class}(`class` $c \ldots$ {$\_\ \_\ \overline{M}$})	$=$	$\bigcup_{M\ \in\ \overline{M}} C^{ct}_{Mthd}(c, M)$
$C^{ct}_{Prg}(\overline{C}\ \_\ E\ \_\ \overline{x_n})$	$=$	$\bigcup_{C\ \in\ \overline{C}} C^{ct}_{Class}(C) \bigcup C^{ct}_{Exp}(\_, \_, E_{main}) \bigcup_{i\in[1,n]} V^{ct}_{\_.\_.x_i} = t^{ct}_{\_.\_.x_i}$

FIG. 6.5 – Type concret : générateur de contraintes.

Nous détaillons maintenant la génération de contraintes pour le calcul des annotations de types concrets. La figure 6.5 présente le générateur de contraintes C^{ct}_{Prg} applicable aux programmes EFJ. Les contraintes sont dérivées des règles des types concrets de la section 6.1.1.

Expressions (C^{ct}_{Exp}). Les contraintes sur les constantes sont générées en fonction du type déclaré de l'expression (*i.e.*, donné par $\Gamma^{dt}(E)$). L'annotation sur la référence à l'instance (`this`) est donnée par la classe de l'instance. L'accès à une variable est lié à l'annotation du paramètre correspondant de la méthode englobante.

Les contraintes associées à l'accès aux champs sont construites à l'aide de l'expression ensembliste $fieldAccess^{ct}$.

Les contraintes générées à partir de l'expression d'instanciation permettent d'annoter l'expression avec la classe référencée comme le type concret ainsi que d'annoter les champs de la classe avec les valeurs des paramètres du constructeur.

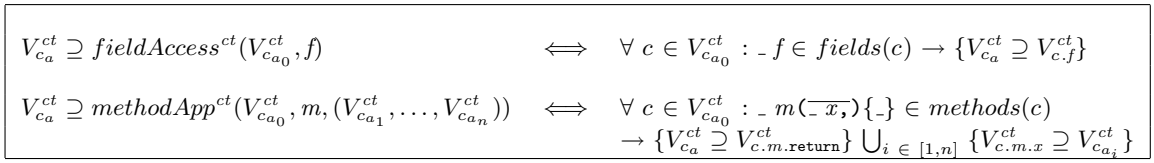


FIG. 6.6 – Type concret : règles de résolution des contraintes.

La contrainte sur les expressions de cœrcition d'un objet est directement donnée par la règle $\textsc{cast}^{ct}$.

Les contraintes sur les opérations binaires sont générées en regardant l'opérateur. Du fait que dans notre approche nous ne considérons pas de relation de sous-typage entre les types primitifs, l'annotation liée à l'expression sera l'ensemble singleton {`int`} ou l'ensemble singleton {`boolean`}. Les contraintes sur les opérations unaires dépendent, par contre, de l'annotation de l'opérande.

Dans le cas de l'expression conditionnelle, en considérant la définition de la règle $\textsc{cond}^{ct}$, le générateur doit lier l'annotation de l'expression à l'annotation des sous-expressions.

La génération de contraintes pour les appels de méthodes est effectuée de manière identique à l'accès à un champ, à savoir en appliquant l'opérateur $methodApp^{ct}$.

Méthodes (C_{Mthd}^{ct}). À partir de la définition d'une méthode M appartenant à une classe C, la contrainte générée par le générateur C_{Mthd}^{ct} exprime simplement la relation d'inclusion stipulée par la règle de typage $\textsc{mthd}^{ct}$.

Classes (C_{Class}^{ct}). Les contraintes générées par le générateur de contraintes sur une classe sont celles résultant de l'analyse des méthodes de la classe.

Programmes (C_{Prg}^{ct}). Le générateur principal C_{Prg}^{ct} se base sur les contraintes générées par le générateur C_{Class}^{ct} sur toutes les classes du programme ainsi que sur celles générées à partir de l'expression initiale.

6.1.2.3 Résolution des contraintes

L'ensemble initial de contraintes, obtenu par application du générateur C_{Prg}^{ct} sur le programme P, est simplifié en appliquant les règles de résolution de la figure 6.6. Ces règles traduisent les contraintes conditionnelles basées sur des opérateurs en contraintes simples. Notamment, les règles de résolution décomposent les contraintes sur la base de la sémantique des contraintes décrite dans la figure 6.4. Nous donnons la définition des ces règles sous la forme $\forall\, v \in V : cond(v) \rightarrow cstr(v)$ (voir section 5.2.1.2).

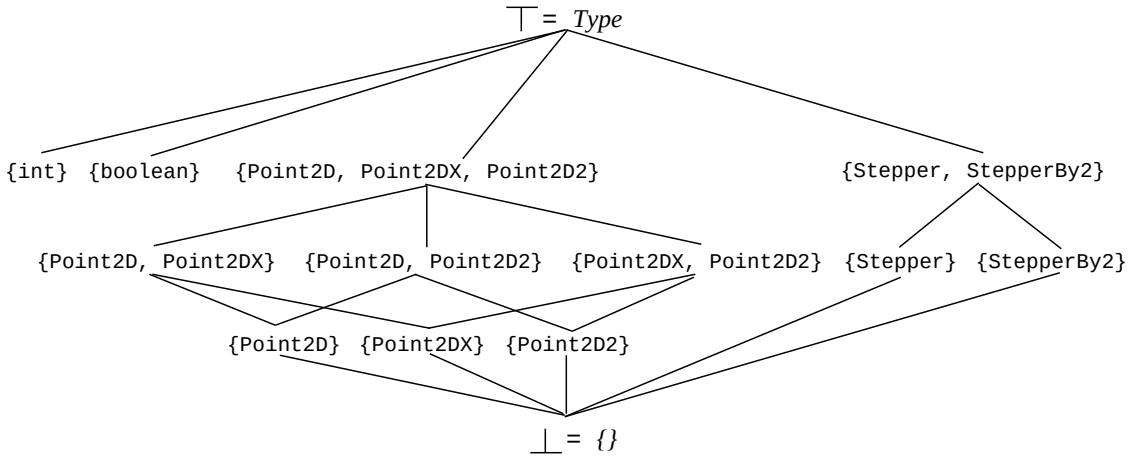


FIG. 6.7 – Treillis d'ensemble de parties des types du programme *Point2DStepping*.

6.1.2.4 Utilisation du solveur REQS

Nous expliquons maintenant l'utilisation du solveur REQS pour l'obtention d'une solution pour l'ensemble de contraintes généré.

Treillis. Dans le cas des annotations de types concrets, le domaine des variables est ordonné par un treillis avec comme relation d'ordre l'inclusion ensembliste. Concrètement, nous allons utiliser le treillis ensemble de parties (*power set*) fourni par REQS, appliqué à l'ensemble $Type$. La figure 6.7 montre le treillis associé au programme *Point2DStepping*.

$$\frac{V_{c_a}^{ct} \supseteq fieldAccess^{ct}(V_{c_{a_0}}^{ct}, f)}{V_{c_a}^{ct} \supseteq \texttt{(IF (belongsTo 'c } V_{c_{a_0}}^{ct}\texttt{) } V_{c.f}^{ct}\texttt{ 'empty)}} \quad [\mathbf{FIELD}_{\overline{req\$}}^{ct}] \qquad \forall\ \texttt{class}\ c\ \ldots\{\ldots\} \in P \mid \_\, f \in fields(c)$$

$$\frac{V_{c_a}^{ct} \supseteq methodApp^{ct}(V_{c_{a_0}}^{ct}, m, (V_{c_{a_1}}^{ct}, \ldots, V_{c_{a_n}}^{ct}))}{\begin{array}{c} V_{c_a}^{ct} \supseteq \texttt{(IF (belongsTo 'c } V_{c_{a_0}}^{ct}\texttt{) } V_{c.m.\texttt{return}}^{ct}\texttt{ 'empty)} \\ \forall\ i \in [1,n] \\ V_{c.m.x_i}^{ct} \supseteq \texttt{(IF (belongsTo 'c } V_{c_{a_0}}^{ct}\texttt{) } V_{c_{a_i}}^{ct}\texttt{ 'empty)} \end{array}} \quad [\mathbf{INVK}_{\overline{req\$}}^{ct}] \qquad \forall\ \texttt{class}\ c\ \ldots\{\ldots\} \in P \mid \_\, m(\overline{\_\, x,})\{\_\} \in methods(c)$$

FIG. 6.8 – Type concret : règles de conversion vers REQS.

Règles de conversion vers REQS. Les règles présentées par la figure 6.8 permettent de réécrire les contraintes en termes de la syntaxe de REQS. Ces règles de conversion se basent sur les règles de résolution décrites dans la figure 6.6. En particulier, ces règles expriment simplement les contraintes conditionnelles (*i.e.*, $fieldAccess^{ct}$ et $methodApp^{ct}$) sous la syntaxe reconnue par le solveur REQS. Comme expliqué auparavant, les contraintes conditionnelles se basent sur les ensembles de classes auxquelles pourrait appartenir le

membre concerné. Cette conditionnalité est exprimée dans REQS par l'opérateur `IF` applicable dans le domaine des ensembles de parties. La syntaxe de l'opérateur est présentée dans la figure 6.9.

$$V_{c_a}^{ct} \quad = \quad \texttt{(IF (belongsTo } V_{c_{a_1}}^{ct} \; V_{c_{a_2}}^{ct} \texttt{)} \; V_{c_{a_3}}^{ct} \; \texttt{'empty)}$$

FIG. 6.9 – REQS : opérateur conditionnel `IF`.

L'interprétation de la contrainte conclut que si l'ensemble représenté par la variable $V_{c_{a_1}}^{ct}$ est inclus dans l'ensemble $V_{c_{a_2}}^{ct}$ alors la variable $V_{c_a}^{ct}$ reçoit la valeur de la variable $V_{c_{a_3}}^{ct}$. Elle reçoit sinon la constante `'empty` utilisée ici pour représenter l'ensemble vide.

Notez que les contraintes résultant de l'application des règles de conversion mentionnées ci-dessus sont exprimées en termes de la relation d'inclusion. Cependant, l'utilisation de REQS implique la transformation des contraintes en termes d'équations selon la syntaxe décrite dans la figure 5.11.

Considérons l'ensemble de contraintes $\mathcal{C}_1^{ct}$: $\{ V_{c.f}^{ct} \supseteq V_{E_1}^{ct}$ et $V_{c.f}^{ct} \supseteq V_{E_2}^{ct}\}$. Il exprime le fait que la construction $c.f$ est affectée par deux expressions. Concrètement, $c.f$ recevra des valeurs de type égal à l'annotation de l'expression E_1 et à celle de l'expression E_2. Dans un contexte monovariant, les inégalités servent à modéliser l'annotation de type concret d'une construction comme l'ensemble de types des valeurs affectées à la construction lors de l'exécution. L'analyse locale (individuelle) de chaque contrainte justifie, dans ce cas, l'utilisation des inégalités par rapport à l'utilisation des égalités. Autrement dit, la contrainte $V_{c.f}^{ct} \supseteq V_{E_1}^{ct}$ exprime le fait que $V_{c.f}^{ct}$ contiendra la valeur liée à la variable $V_{E_1}^{ct}$ mais ne dit rien sur le fait d'affecter d'autres valeurs à travers d'autres contraintes.

Nous illustrons dans la figure 6.10 comment les inégalités peuvent être transformées en égalités afin de pouvoir utiliser REQS pour trouver la plus petite solution qui satisfasse l'ensemble $\mathcal{C}_1^{ct}$. En REQS, les variables de contraintes sont initialisées à la valeur associée à $\perp$ du treillis. Dans le cas du treillis ensemble de parties, cette valeur est l'ensemble vide ($\{\}$). La figure 6.10(a) montre un exemple de résolution des contraintes basées sur des inégalités, où la solution trouvée est la plus petite des solutions possibles. La partie droite du tableau montre l'évolution de la solution à chaque ajout de contrainte ce qui sert à mieux comprendre l'impact de chaque contrainte sur la solution. En remplaçant simplement la relation d'inégalité par une égalité dans toutes les contraintes nous obtenons l'ensemble de contraintes de la figure 6.10(b). En particulier, REQS travaille sur des équations où la partie gauche de l'égalité est composée d'une variable simple telle que si la même variable se trouve plusieurs fois dans la partie gauche le solveur ne prend en compte que la dernière définition. Pour cette raison, le résultat de la résolution de ce dernier ensemble de contraintes ne constitue pas une solution valide pour l'ensemble initial des contraintes car la valeur finale associée à la variable $V_{c.f}^{ct}$ ne satisfait pas la contrainte *(3)* de la figure 6.10(a). Cependant, si on considère les contraintes *(3)* et *(4)* comme les seules contraintes sur la construction $c.f$ alors ces deux contraintes peuvent être réécrites en regroupant les expressions correspondantes. Dans ce cas, la solution cherchée doit associer

(a) Contraintes.

	Contraintes	Plus petite solution
(1)	$V^{ct}_{E_1} \supseteq \{T\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{}, $V^{ct}_{c.\mathtt{x}}$:{}
(2)	$V^{ct}_{E_2} \supseteq \{E\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{}
(3)	$V^{ct}_{c.\mathtt{x}} \supseteq V^{ct}_{E_1}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{T}
(4)	$V^{ct}_{c.\mathtt{x}} \supseteq V^{ct}_{E_2}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{T, E}

(b) Contraintes sur la forme d'égalités.

	Equations	Solution
(1)	$V^{ct}_{E_1} = \{T\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{}, $V^{ct}_{c.\mathtt{x}}$:{}
(2)	$V^{ct}_{E_2} = \{E\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{}
(3)	$V^{ct}_{c.\mathtt{x}} = V^{ct}_{E_1}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{T}
(4)	$V^{ct}_{c.\mathtt{x}} = V^{ct}_{E_2}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{E}

(c) Contraintes sur la forme d'équations REQS.

	Equations REQS	Solution
(1)	$V^{ct}_{E_1} = \{T\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{}, $V^{ct}_{c.\mathtt{x}}$:{}
(2)	$V^{ct}_{E_2} = \{E\}$	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{}
(3)	$V^{ct}_{c.\mathtt{x}}$ = (UNION $V^{ct}_{E_1}$ $V^{ct}_{E_2}$)	$V^{ct}_{E_1}$:{T}, $V^{ct}_{E_2}$:{E}, $V^{ct}_{c.\mathtt{x}}$:{T}$\cup${E} :{T,E}

FIG. 6.10 – Type concret : conversion des inégalités en équations REQS.

à la variable $V^{ct}_{c.f}$ le plus petit ensemble incluant les ensembles $V^{ct}_{E_1}$ et $V^{ct}_{E_1}$. Par conséquent, il devient naturel de redéfinir la partie droite de la contrainte comme l'union ensembliste des expressions correspondantes, c'est-à-dire $V^{ct}_{E_1} \cup V^{ct}_{E_2}$. Le solveur REQS fournit, à travers le treillis d'ensemble de parties, l'opérateur d'union. La figure 6.10(c) montre la solution proposée par REQS par rapport au système d'équations obtenu à partir de l'ensemble $\mathcal{C}^{ct}_1$.

La conversion d'inégalités en égalités est effectuée en appliquant la règle **I-TO-E**$^{\overline{reqs}}$ dont la définition est présentée dans la figure 6.11.

$$\frac{V^{ct}_{c_a} \supseteq en_1, \ldots, V^{ct}_{c_a} \supseteq en_i, \ldots, V^{ct}_{c_a} \supseteq en_n}{V^{ct}_{c_a} = (\mathtt{UNION}\ en_1 \ldots (\mathtt{UNION}\ en_i \ldots)\ en_n) \ldots)} \quad [\textbf{I-TO-E}^{\overline{reqs}}]$$

FIG. 6.11 – Type concret : conversion d'inégalités en égalités.

6.1.2.5 Inférence des types concrets sur le programme *Point2DStepping*

Dans cette section, nous appliquons l'approche décrite pour l'inférence de types concrets sur le programme *Point2DStepping* introduit dans la section 5.3.2.

Réplication de membres hérités. Pour mieux comprendre la réplication des membres des classes, il faut reprendre la définition des fonctions *fields* et *methods* proposées dans la figure 5.13. En appliquant ces fonctions avec comme argument, par exemple, la classe `Point2D2` nous obtenons le résultat donné figure 6.12.

$$
\begin{array}{lcl}
\mathit{fields}(\texttt{Point2DX}) & = & \{\texttt{int x, int y, Stepper aStepper}\} \\
\mathit{methods}(\texttt{Point2DX}) & = & \{\texttt{Point2DX next(int offset)\{...\}, int step()\{...\}}\}
\end{array}
$$

FIG. 6.12 – Réplication de membres hérités : membres de la classe `Point2DX`

Comme mentionné dans la section 5.3.1.1, le code source du programme n'est soumis à aucune transformation pour rendre explicite la réplication des champs et méthodes. Néanmoins, du point de vue de l'analyse, les champs et méthodes hérités par une classe sont analysés comme appartenant à cette classe. Par exemple, le champ libellé `x` défini dans les classes `Point2DX` et `Point2D2` étant hérité de la superclasse `Point2D`, il est identifié comme le champ `Point2D.x`, `Point2DX.x` et `Point2D2.x`, respectivement. Ceci permet de séparer, par exemple, l'analyse d'instances de la classe `Point2D` de celle de la classe `Point2D2`. La figure 6.13 montre la perspective de l'analyse par rapport au programme *Point2DStepping* où les membres répliqués sont propagés (dans la figure en police ***italique-gras***) sur la classe correspondante.

Génération des contraintes Soit le programme *Point2DStepping* :

$$\mathit{Point2DStepping} \quad = \quad \mathit{Point2D}\ \mathit{Point2DX}\ \mathit{Point2D2}\ \mathit{Stepper}\ \mathit{StepperBy2}\ \texttt{int}\ E_{main}$$

tel que l'expression initiale E_{main} :

$$
\begin{array}{lcl}
E_{main} & = & \texttt{new Point2DX(x}_1\texttt{, y}_1\texttt{, new Stepper(step}_1\texttt{)).next(offset}_1\texttt{).x +} \\
 & & \texttt{new Point2D2(x}_2\texttt{, y}_2\texttt{, new StepperBy2(step}_2\texttt{)).next(offset}_2\texttt{).y}
\end{array}
$$

où les variables libres ont pour type :

$$
\begin{array}{lcl}
\Gamma_1^{ct} & = & \{\texttt{x}_1 \mapsto \texttt{int}, \texttt{y}_1 \mapsto \texttt{int}, \texttt{step}_1 \mapsto \texttt{int}, \texttt{offset}_1 \mapsto \texttt{int}, \\
 & & \texttt{x}_2 \mapsto \texttt{int}, \texttt{y}_2 \mapsto \texttt{int}, \texttt{step}_2 \mapsto \texttt{int}, \texttt{offset}_2 \mapsto \texttt{int}\}
\end{array}
$$

Du fait du nombre de contraintes générées pour le programme *Point2DStepping*, nous ne détaillons qu'un sous-ensemble du résultat de l'analyse. Concrètement, nous nous concentrons sur la génération des contraintes à partir des expressions d'instanciation de l'expression initiale.

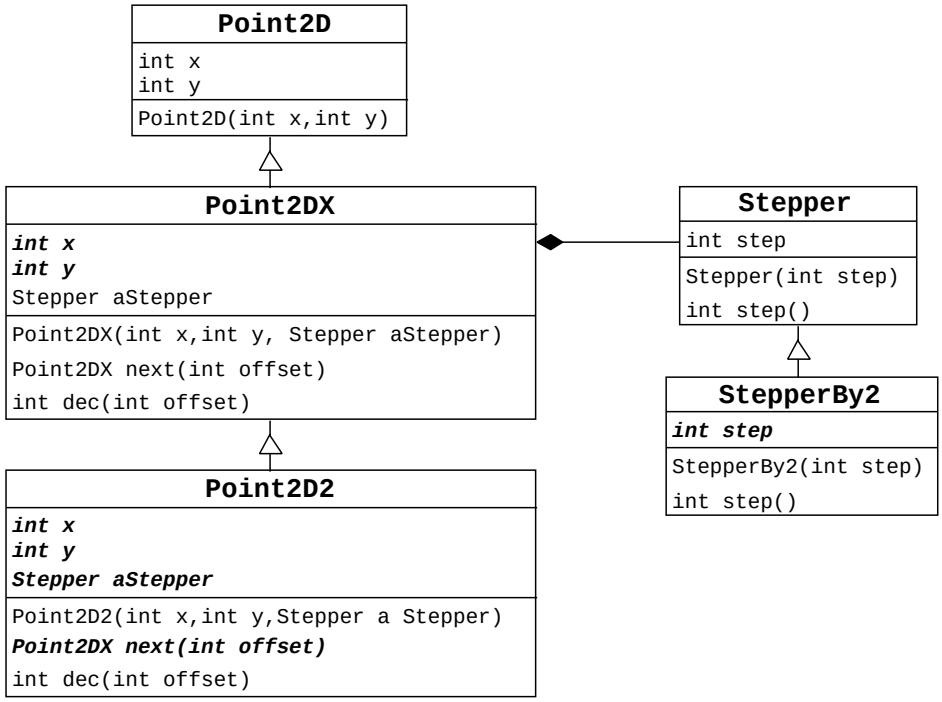


FIG. 6.13 – Réplication de membres hérités : perspective de l'analyse.

$$
\begin{aligned}
C^{ct}_{Prg}(Point2DStepping) \quad = \quad & C^{ct}_{Class}(Point2D) \cup \ldots \cup \\
& C^{ct}_{Class}(Point2DX) \cup \ldots \cup \\
& C^{ct}_{Class}(Point2D2) \cup \ldots \cup \\
& \ldots \\
& C^{ct}_{Exp}(\_, \_, E_{main})
\end{aligned}
$$

Le but est d'illustrer la propagation de types concrets tout au long de la création d'objets, surtout dans les cas des champs de type objet comme, par exemple, le champ libellé `aStepper` dont la définition appartient à la classe `Point2DX`.

Le tableau 6.1 contient l'ensemble des contraintes généré à partir de l'analyse de l'expression E_{main}.

Des contraintes (f), (g), (o) et (p) nous concluons que la variable $V^{ct}_{\texttt{Point2DX.aStepper}} = \{\texttt{Stepper}\}$ tandis que la variable $V^{ct}_{\texttt{Point2D2.aStepper}} = \{\texttt{StepperBy2}\}$. Concrètement, nous annotons deux constructions (le champs `Point2DX.aStepper` et `Point2D2.aStepper`) qui ne sont pas définies explicitement dans le programme, elles sont héritées de la superclasse `Point2D`.

En ce qui concerne l'expression initiale E_{main}, la figure 6.14 montre l'annotation de type concret résultant de la résolution des contraintes générées et des valeurs initiales données par l'environnement Γ^{ct}_1 mentionné auparavant.

Générateur	Éléments de $\mathcal{C}^{ct}$	
$C^{ct}_{Exp}(\_,\_,E_{main})$		
$\quad C^{ct}_{Exp}(\_,\_,E_1)$		
$\quad E_1 = E_{10}.\texttt{x}$	$V^{ct}_{\_,\_,E_1} \supseteq fieldAccess^{ct}(V^{ct}_{\_,\_,E_{10}}, \texttt{x})$	(a)
$\quad E_{10} = E_{100}.\texttt{next}(E_{101})$	$V^{ct}_{\_,\_,E_{10}} \supseteq methodApp^{ct}(V^{ct}_{\_,\_,E_{100}}, \texttt{next}, (V^{ct}_{\_,\_,E_{101}}))$	(b)
$\quad E_{100} = \texttt{new Point2DX}(E_{1001},$	$V^{ct}_{\_.\_.E_{100}} = \{\texttt{Point2DX}\}$	(c)
$\qquad\qquad E_{1001},$	$V^{ct}_{\texttt{Point2DX.x}} \supseteq V^{ct}_{\_.\_.E_{1001}}$	(d)
$\qquad\qquad E_{1002},$	$V^{ct}_{\texttt{Point2DX.y}} \supseteq V^{ct}_{\_.\_.E_{1002}}$	(e)
$\qquad\qquad E_{1003})$	$V^{ct}_{\texttt{Point2DX.aStepper}} \supseteq V^{ct}_{\_.\_.E_{1003}}$	(f)
$\quad \ldots$		
$\quad E_{1003} = \texttt{new Stepper}(E_{10031})$	$V^{ct}_{\_.\_.E_{1003}} = \{\texttt{Stepper}\}$	(g)
	$V^{ct}_{\texttt{Stepper.step}} \supseteq V^{ct}_{\_.\_.E_{10031}}$	(h)
$\quad \ldots$		
$\quad C^{ct}_{Exp}(\_,\_,E_2)$		
$\quad E_2 = E_{20}.\texttt{y}$	$V^{ct}_{\_,\_,E_2} \supseteq fieldAccess^{ct}(V^{ct}_{\_,\_,E_{20}}, \texttt{y})$	(i)
$\quad E_{20} = E_{200}.\texttt{next}(E_{201})$	$V^{ct}_{\_,\_,E_{20}} \supseteq methodApp^{ct}(V^{ct}_{\_,\_,E_{200}}, \texttt{next}, (V^{ct}_{\_,\_,E_{201}}))$	(j)
$\quad E_{200} = \texttt{new Point2D2}(E_{2001},$	$V^{ct}_{\_.\_.E_{200}} = \{\texttt{Point2D2}\}$	(k)
$\qquad\qquad E_{2001},$	$V^{ct}_{\texttt{Point2D2.x}} \supseteq V^{ct}_{\_.\_.E_{2001}}$	(m)
$\qquad\qquad E_{2002},$	$V^{ct}_{\texttt{Point2D2.y}} \supseteq V^{ct}_{\_.\_.E_{2002}}$	(n)
$\qquad\qquad E_{2003})$	$V^{ct}_{\texttt{Point2D2.aStepper}} \supseteq V^{ct}_{\_.\_.E_{2003}}))$	(o)
$\quad \ldots$		
$\quad E_{2003} = \texttt{new StepperBy2}(E_{20031})$	$V^{ct}_{\_.\_.E_{2003}} = \{\texttt{StepperBy2}\}$	(p)
	$V^{ct}_{\texttt{StepperBy2.step}} \supseteq V^{ct}_{\_.\_.E_{20031}}$	(q)

TAB. 6.1 – Type concret : génération des contraintes sur l'expression initiale E_{main}.

6.2 Analyse de temps de liaison

Nous avons vu dans la section précédente l'annotation de types concrets sur un programme, dont les constructions sont annotées avec des ensembles de types. Nous savons aussi si ces constructions sont de type primitif ou de type objet. Le domaine classique des temps de liaison, dénoté ici par $\pi = \{\mathcal{S}, \mathcal{D}\}$ [JGS93], est applicable aux constructions de type primitif.

Une construction de type objet est utilisée pour l'accès à ses membres (*i.e.*, un champ ou une méthode) qui peuvent être eux-mêmes aussi d'un des deux types mentionnés. Dans le cas d'un membre de type primitif, l'annotation est le résultat de l'application de l'opérateur plus petit majorant $\sqcup$ (voir équations 2.5) aux annotations de temps de liaison (selon de domaine π) du membre défini dans tous le types concrets calculés.

Dans le cas d'un membre de type objet, comme il sera utilisé ultérieurement pour l'accès à un de ses membres, le temps de liaison est calculé en fonction des types concrets (voir section 5.2.1) associés au membre. Pour cette raison, dans cette thèse, le temps de liaison des constructions de type objet se résume aux annotations de type concret calculées dans l'analyse de temps concret abordée dans la section précédente.

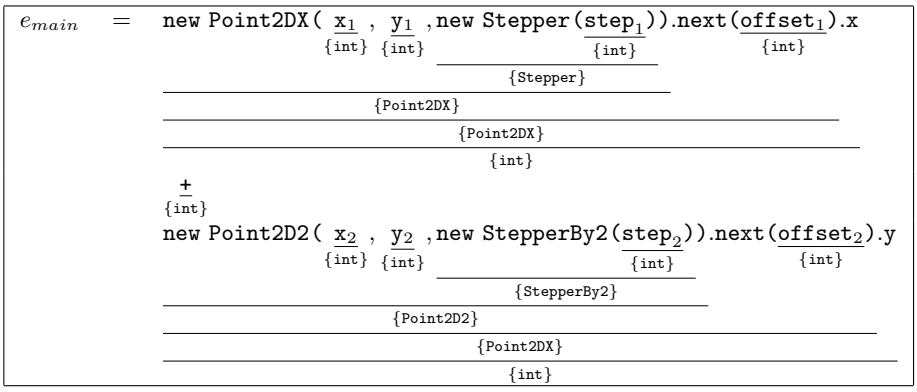


FIG. 6.14 – Type concret : annotation de l'expression initiale E_{main}.

Domaine des annotations. En conséquence, dans le contexte de notre approche, les annotations résultant de l'analyse de temps de liaison seront des valeurs dans le domaine π, pour les constructions primitives, et des valeurs dans l'ensemble de parties $\mathcal{P}(\mathcal{C})$, pour les constructions de type objet. Les domaines des annotations de temps de liaison sont unifiés sur l'ensemble BT défini comme suit :

$$\pi = \{\mathcal{S}, \mathcal{D}\}$$

$$BT = (\pi \cup \mathcal{P}(\mathcal{C})) \tag{6.2}$$

Afin de pouvoir identifier le type des constructions on applique la fonction *primitive* définie dans la figure 6.15.

$$\frac{\Gamma^{dt} \vdash c_a : t \qquad t \in p}{primitive(c_a)} \quad [\mathbf{PRIM}^{bt}]$$

FIG. 6.15 – Temps de liaison : définitions auxiliaires.

Les annotations de temps de liaison sont liées aux constructions correspondantes par l'environnement Γ^{bt}. Nous dénotons $t^{bt}_{c_a}$ le temps de liaison de la construction c_a, tel que le jugement $\Gamma^{bt} \vdash c_a : t^{bt}_{c_a}$ détermine que « la construction c_a est associée à l'annotation $t^{bt}_{c_a}$ selon l'environnement Γ^{bt} ».

Nous définissons l'environnement Γ^{bt} comme suit :

$$\Gamma^{bt}(c_a) : C_A \rightarrow BT \text{ tel que}$$

$$\Gamma^{bt}(c_a) \in \begin{cases} \pi & \text{si } primitive(c_a), \\ \mathcal{P}(\mathcal{T}) & \text{si } \neg\ primitive(c_a). \end{cases} \tag{6.3}$$

L'inférence de type concret se sert de la relation d'inclusion ensembliste pour établir les annotations sur un programme. Concrètement, l'expression $t^{ct}_{E_i} \subseteq t^{ct}_{E_j}$ implique que la bonne annotation des expressions concernées ne dépend que de l'annotation de type concret sur l'expression E_i, qui doit être incluse dans l'annotation liée à l'expression E_j. Dans le contexte de l'analyse de temps de liaison, la relation d'ordre entre les valeurs $\mathcal{S}$ et $\mathcal{D}$ est donnée par la relation *moins dynamique que* définie par l'équation 2.4 de la section 2.1.2.1, où la relation d'ordre est établie par l'expression $\mathcal{S} \sqsubset \mathcal{D}$.

6.2.1 Règles de bonne annotation de temps de liaison

En appliquant la même procédure que pour les annotations de types concrets (voir section 6.1.1), nous décrivons dans cette section les règles qui permettent de savoir si un programme est bien annoté par rapport au temps de liaison de ses constructions. Pour les constructions de type objet on utilise la règle de la figure 6.16.

$$\frac{\neg primitive(c_a) \qquad \Gamma^{ct} \vdash c_a : t^{ct}_{c_a}}{\Gamma^{bt} \vdash c_a : t^{ct}_{c_a}} \quad [\mathbf{Obj}^{bt}]$$

FIG. 6.16 – Temps de liaison : constructions de type objet.

La figure 6.17 inclut les règles de bonne annotation sur les expressions de type primitif ainsi que celles pour les expressions de type objet. Dans ce dernier cas, une règle est uniquement nécessaire quand l'expression est composée de sous-expressions, lesquelles doivent être aussi annotées.

Expressions. Le temps de liaison des constantes, entier ou booléen, est toujours associé à la valeur primitive statique $\mathcal{S}$ (règles $\textbf{INT}^{bt}$ et $\textbf{BOOL}^{bt}$).

En ce qui concerne l'accès à un champ (règle $\textbf{FIELD-Prim}^{bt}$), le temps de liaison dépend des temps de liaison des champs *f*, définis dans les classes éléments de l'annotation de type concret $t^{ct}_{E_0}$ dont E_0 représente la sous-expression du receveur. Si le champ est de type objet, la règle $\textbf{FIELD-Obj}^{bt}$ permet d'analyser l'annotation de la sous-expression du receveur.

Comme l'expression d'instanciation d'un objet est uniquement de type objet, la règle déterminant la bonne annotation de cette expression (règle $\textbf{NEW-Prim}^{bt}$) ne concerne que le temps de liaison des paramètres de type primitif du constructeur. Concrètement, les annotations sur les champs de la classe instanciée auront la valeur la plus dynamique entre les annotations associées aux paramètres par rapport à tous les appels du constructeur dans

Expressions :

$$\frac{\Gamma^{ct} \vdash k : \{\texttt{int}\}}{\Gamma^{bt} \vdash k : \mathcal{S}}\ [\mathbf{INT}^{bt}] \qquad \frac{\Gamma^{ct} \vdash k : \{\texttt{boolean}\}}{\Gamma^{bt} \vdash k : \mathcal{S}}\ [\mathbf{BOOL}^{bt}] \qquad \Gamma^{bt} \vdash x : \Gamma^{bt}(x)\ \ [\mathbf{VAR}^{bt}]$$

$$\frac{\mathit{primitive}(E_0.f) \qquad \Gamma^{bt} \vdash E_0 : t^{bt}_{E_0} \qquad \forall\ c\ \in\ t^{ct}_{E_0} \qquad (t^{bt}_{E} \sqsupseteq \Gamma^{bt}(c.f))}{\Gamma^{bt} \vdash (E_0.f) : t^{bt}_{E}}\ [\mathbf{FIELD\text{-}Prm}^{bt}]$$

$$\frac{\neg\mathit{primitive}(E_0.f) \qquad \Gamma^{ct} \vdash E_0.f : t \qquad \Gamma^{bt} \vdash E_0 : \_}{\Gamma^{bt} \vdash (E_0.f) : t}\ [\mathbf{FIELD\text{-}Obj}^{bt}]$$

$$\frac{\mathit{fields}(c) = \overline{\_\ f_n} \qquad \forall\ i\ \in [1,n] \qquad (\mathit{primitive}(c.f_i) \qquad \Gamma^{bt} \vdash E_i : t^{bt}_{E_i} \qquad t^{bt}_{E_i} \sqsubseteq \Gamma^{bt}(c.f_i))}{(\texttt{new}\ c(\overline{E}))\ \textit{well-bt-annotated}}\ [\mathbf{NEW\text{-}Prim}^{bt}]$$

$$\frac{\mathit{fields}(c) = \overline{\_\ f_n} \qquad \forall\ i\ \in [1,n] \qquad (\neg\mathit{primitive}(c.f_i) \qquad \Gamma^{bt} \vdash E_i : \_)}{(\texttt{new}\ c(\overline{E}))\ \textit{well-bt-annotated}}\ [\mathbf{NEW\text{-}Obj}^{bt}]$$

$$\frac{\Gamma^{bt} \vdash E : \_}{((d)E) : t^{ct}_{E}\ \textit{well-bt-annotated}}\ [\mathbf{CAST\text{-}Obj}^{bt}]$$

$$\frac{\Gamma^{bt} \vdash E_1 : t^{bt}_{E_1} \qquad \Gamma^{bt} \vdash E_2 : t^{bt}_{E_2} \qquad t^{bt}_{E} \sqsupseteq t^{bt}_{E_1} \qquad t^{bt}_{E} \sqsupseteq t^{bt}_{E_2}}{\Gamma^{bt} \vdash (E_1\ op_b\ E_2) : t^{bt}_{E}}\ [\mathbf{OB}^{bt}] \qquad \frac{\Gamma^{bt} \vdash E_1 : t^{bt}_{E_1}}{\Gamma^{bt} \vdash (op_b\ E_1) : t^{bt}_{E_1}}\ [\mathbf{OU}^{bt}]$$

$$\frac{\begin{array}{c}\mathit{primitive}(E_1\texttt{?}E_2\texttt{:}E_3) \qquad \Gamma^{bt} \vdash E_1 : t^{bt}_{E_1} \qquad \Gamma^{bt} \vdash E_2 : t^{bt}_{E_2} \qquad \Gamma^{bt} \vdash E_3 : t^{bt}_{E_3} \\ t^{bt}_{E} \sqsupseteq t^{bt}_{E_1} \qquad t^{bt}_{E} \sqsupseteq t^{bt}_{E_2} \qquad t^{bt}_{E} \sqsupseteq t^{bt}_{E_3}\end{array}}{\Gamma^{bt} \vdash (E_1\texttt{?}E_2\texttt{:}E_3) : t^{bt}_{E}}\ [\mathbf{COND\text{-}Prim}^{bt}]$$

$$\frac{\neg\mathit{primitive}(E_1\texttt{?}E_2\texttt{:}E_3) \qquad \Gamma^{bt} \vdash E_1 : t^{bt}_{E_1} \qquad \Gamma^{bt} \vdash E_2 : \_ \qquad \Gamma^{bt} \vdash E_3 : \_}{\Gamma^{bt} \vdash (E_1\texttt{?}E_2\texttt{:}E_3) : t^{bt}_{E}}\ [\mathbf{COND\text{-}Obj}^{bt}]$$

$$\frac{\begin{array}{c}\Gamma^{ct} \vdash E_0 : t^{bt}_{E_0} \qquad \forall\ c\ \in\ t^{ct}_{E_0} \qquad (\mathit{mbody}(m,c) = ((\overline{x_n})E) \\ \forall\ i\ \in [1,n] \qquad (\Gamma^{bt} \vdash E_i : t^{bt}_{x_i} \qquad \mathit{primitive}(c.m.x_i) \Rightarrow t^{bt}_{x_i} \sqsubseteq \Gamma^{bt}(c.m.x_i)) \\ \mathit{primitive}(c.m.\texttt{return}) \Rightarrow \Gamma^{bt}(c.m.\texttt{return}) \sqsupseteq \Gamma^{bt}(E))\end{array}}{\Gamma^{bt} \vdash (E_0.m(\overline{E,})) : t^{bt}_{E}}\ [\mathbf{INVK\text{-}Prm}^{bt}]$$

$$\frac{\begin{array}{c}\Gamma^{ct} \vdash E_0 : t^{bt}_{E_0} \qquad \forall\ c\ \in\ t^{ct}_{E_0} \qquad (\mathit{mbody}(m,c) = ((\overline{x_n})E) \\ \forall\ i\ \in [1,n] \qquad (\neg\mathit{primitive}(c.m.x_i) \qquad \Gamma^{bt} \vdash c.m.x_i : \_) \\ \neg\mathit{primitive}(c.m.\texttt{return}) \qquad \Gamma^{ct} \vdash c.m.\texttt{return} : t) \qquad \Gamma^{bt} \vdash E_0 : \_\end{array}}{\Gamma^{bt} \vdash (E_0.m(\overline{E,})) : t}\ [\mathbf{INVK\text{-}Obj}^{bt}]$$

Méthodes :

$$\frac{(\texttt{this} : \{c\}) :: \Gamma^{bt} \vdash c.m.\texttt{return} : t^{bt}_{c.m.\texttt{return}} \qquad t^{bt}_{c.m.\texttt{return}} \sqsupseteq \Gamma^{bt}(E)}{\Gamma^{bt} \vdash \_\ m(\_\ \_)\{\texttt{return}\ E\}\ \textit{well-bt-annotated in } c}\ [\mathbf{MTHD}^{bt}]$$

Classes :

$$\frac{\Gamma^{bt} \vdash \overline{M}\ \textit{well-bt-annotated in } c}{\Gamma^{bt} \vdash \texttt{class}\ c\ \texttt{extends}\ \_\ \{\_\ \_\ \overline{M}\}\ \textit{well-bt-annotated}}\ [\mathbf{CLS}^{bt}]$$

Programmes :

$$\frac{\Gamma^{bt}_0 \vdash \overline{C}\ \textit{well-bt-annotated} \qquad \Gamma^{bt}_0(\overline{x}) : \overline{t^{bt}_x} \qquad \Gamma^{bt}_0 \vdash E : t^{bt}_E}{\Gamma^{bt}_0 \vdash \overline{C}\ t\ E\ \overline{t\ x}\ \textit{well-bt-annotated}}\ [\mathbf{PRG}^{bt}]$$

FIG. 6.17 – Temps de liaison : règles sur les programmes EFJ.

le programme donné. Dans le cas de paramètres de type objet, c'est la règle **NEW-Obj**^{bt} qui permet de vérifier leur bonne annotation.

L'expression de coercition d'un objet est bien annotée si l'expression représentant l'objet est aussi bien annotée (règle **CAST-Obj**^{bt}.

Les opérations binaires comme les opérations unaires ne propagent que des valeurs de type primitif. Par conséquent, dans les deux cas, la valeur de temps de liaison des expressions dépend du temps de liaison des sous-expressions (règles **OB**^{bt} et **OU**^{bt}). Notez que, à la différence des règles précédentes pour ce type d'expressions, l'opérateur concerné n'est pas pris en compte dans le calcul de la règle, d'où la raison de l'existence d'une seule règle pour chacune.

L'expression conditionnelle dépend du temps de liaison associé à la sous-expression de la condition (E_1) ainsi que du temps de liaison des sous-expressions qui représentent les branches (E_2 et E_3) (règle **COND-Prim**^{bt}). Par contre, si l'expression conditionnelle est de type objet la bonne annotation de l'expression est uniquement fonction du temps de liaison de la sous-expression de la condition. La règle **COND-Obj**^{bt} permet d'analyser la bonne annotation des branches de l'expression.

Pour ce qui est des appels de méthodes, la règle exprime la bonne annotation sur les paramètres ainsi que sur l'expression elle-même (règle **INVK-Prim**^{bt}). La relation entre le temps de liaison des paramètres et celui des sous-expressions E_i est définie de manière similaire aux paramètres du constructeur. L'annotation sur l'expression dépend du temps de liaison de l'expression de retour correspondante, c'est-à-dire l'annotation de l'expression de retour de la méthode m des classes c éléments de l'annotation de type concret de l'expression du receveur. Pour ce qui est des paramètres de type objet de la méthode, la bonne annotation de temps de liaison est assurée par la régle **INVK-Obj**^{bt}. Cette règle est aussi applicable à la valeur de retour de type objet de la méthode analysée.

Notons que la figure 6.17 n'inclut pas de règles associées à la référence à l'objet (`this`). Dans ce cas, l'expression est toujours de type objet et donc on applique simplement la règle **Obj**^{bt} (voir figure 6.16).

Méthodes. On considère que l'annotation de temps de liaison d'une méthode est correcte si l'expression de retour est aussi bien annotée (règle **MTHD**^{bt}).

Classes. Dans le cas des classes, une bonne annotation est conditionnée par la bonne annotation des méthodes correspondantes (**CLASS**^{bt}).

Programmes. Finalement, la bonne annotation d'un programme dépend de la bonne annotations des classes et de l'expression initiale par rapport à l'annotation de temps de liaison donnée par l'environnement Γ_0^{bt}. Cet environnement est initialisé par les annotations sur les variables libres contenues dans l'expression initiale.

6.2.2 Analyse par contraintes

Dans cette section nous décrivons la définition, la génération et, enfin, la résolution des contraintes pour le calcul de temps de liaison.

6.2.2.1 Définition des contraintes

Une variable de contrainte lieé à la construction c_a est dénotée par $V_{c_a}^{bt}$. Tout comme dans l'inférence de types concrets, les variables de contraintes prennent une forme différente selon la construction référencée, soit $V_{c.f}^{bt}$, $V_{c.m.\texttt{return}}^{bt}$, $V_{c.m.x}^{bt}$ et V_E^{bt} pour les champs, la valeur de retour des méthodes, les paramètres et les expressions, respectivement.

Puisque l'annotation du temps de liaison sur les constructions de type objet est celle obtenue par l'inférence de types concrets, on ne génère pas de contraintes sur les constructions de type objet.

$V_{c_a}^{bt}$	$\sqsupseteq$	bt	contrainte
bt	:=	$\mathcal{S}$	constante *statique*
	|	$\mathcal{D}$	constante *dynamique*
	|	$V_{c_a}^{bt}$	variable de contrainte
	|	$fieldAccess^{bt}(t_{c_{a_0}}^{ct}, f)$	accès à un champ
	|	$methodApp^{bt}(t_{c_{a_0}}^{ct}, m, (V_{c_{a_1}}^{bt}, \ldots, V_{c_{a_n}}^{bt}))$	appel d'une méthode

FIG. 6.18 – Temps de liaison : syntaxe des contraintes.

Les contraintes simples sont créées à partir des constantes de temps de liaison ou d'une autre variable de contrainte, tandis que les contraintes conditionnelles sont formées en utilisant les opérateurs $fieldAccess^{bt}$ et $methodApp^{bt}$.

Sémantique des contraintes. La sémantique formelle des expressions des contraintes de la figure 6.18 est définie par une interprétation $\mathcal{I}^{bt}$ liant les expressions bt avec la valeur de temps de liaison correspondante. De même l'interprétation sur les expressions ensemblistes $\mathcal{I}^{ct}$, $\mathcal{I}^{bt}$ représente une solution sur l'ensemble de contraintes $\mathcal{C}^{bt}$ si pour chaque contrainte $V_{c_a}^{bt} \sqsupseteq bt \in \mathcal{C}^{bt}$, $\mathcal{I}^{bt}(V_{c_a}^{bt}) \sqsupseteq \mathcal{I}^{bt}(bt)$. Notons, par exemple, que la définition de l'opérateur $methodApp^{bt}$ se base sur l'annotation de type concret, $t_{c_{a_0}}^{ct}$, de l'expression du receveur de l'appel de la méthode. Ceci montre clairement que l'analyse de temps de liaison dépend du résultat de l'analyse des temps concrets. La figure 6.19 donne la sémantique des contraintes en utilisant l'interprétation $\mathcal{I}^{bt}$.

6.2.2.2 Génération des contraintes

Les contraintes de temps de liaison sont dérivées des règles de bonne annotation décrites dans la section 6.2.1. L'ensemble de contraintes $\mathcal{C}^{bt}$ est généré en appliquant un générateur de contraintes sur la définition des classes d'un programme. La figure 6.20 montre la structure du générateur de contraintes C_{Prg}^{bt} basé sur les sous-générateurs associés aux différents types des constructions analysées.

Les générateurs de contraintes prennent en compte le fait qu'une construction puisse être de type primitif ou pas en appliquant la fonction *primitive* (voir figure 6.15).

$$
\begin{array}{lcl}
\mathcal{I}^{bt}(\bot) & = & \emptyset \\
\mathcal{I}^{bt}(\mathcal{S}) & = & \mathcal{S} \\
\mathcal{I}^{bt}(\mathcal{D}) & = & \mathcal{D} \\
\mathcal{I}^{bt}(V^{bt}_{c_a}) & \sqsubseteq & \mathcal{D} \\
\mathcal{I}^{bt}(fieldAccess^{bt}(t^{ct}_{c_{a_0}}, f)) & = & \{\mathcal{I}^{bt}(V^{bt}_{c.f}) \mid c \in t^{ct}_{c_{a_0}}, \_\, f \in fields(c), primitive(f)\} \\
\mathcal{I}^{bt}(methodApp^{bt}(t^{ct}_{c_{a_0}}, m, (V^{bt}_{c_{a_1}}, \ldots, V^{bt}_{c_{a_n}}))) & = & \{\mathcal{I}^{bt}(V^{bt}_{c.m.\texttt{return}}) \mid c \in t^{ct}_{c_{a_0}}, \_\, m(\overline{\_\, x_n\,,})\{\_\} \in methods(c) \\
 & & \forall\, i \in [1,n],\, primitive(c_{a_i}),\, \mathcal{I}^{bt}(V^{bt}_{c.m.x_i}) \sqsupseteq \mathcal{I}^{bt}(V^{bt}_{c_{a_i}})\} \\
\mathcal{I}^{bt}(\top) & = & \mathcal{D}
\end{array}
$$

FIG. 6.19 – Temps de liaison : sémantique des contraintes.

$$
\begin{array}{lcl}
C^{bt}_{Exp}(c, m, E) & = & \text{case } E \text{ of} \\
c & \Rightarrow & \{V^{bt}_E = \mathcal{S}\} \\
x & \Rightarrow & \{V^{bt}_E = V^{bt}_{c.m.x}\} \\
E_0.f & \Rightarrow & \{V^{bt}_E = fieldAccess^{bt}(t^{ct}_{E_0}, f)\} \bigcup C^{bt}_{Exp}(c, m, E_0) \\
\texttt{new}\ d\texttt{(}E_1, \ldots, E_n\texttt{)} & \Rightarrow & \bigcup_{i\in[1,n],\ primitive(E_i)} \{V^{bt}_{d.f_i} \sqsupseteq V^{bt}_{E_i}\} \bigcup_{i\in[1,n]} C^{bt}_{Exp}(c, m, E_i) \\
\texttt{(}d\texttt{)}E & \Rightarrow & C^{bt}_{Exp}(c, m, E) \\
E_1\ op_b\ E_2 & \Rightarrow & \{V^{bt}_E \sqsupseteq V^{bt}_{E_1},\, V^{bt}_E \sqsupseteq V^{bt}_{E_2}\} \bigcup_{i\in[1,2]} C^{bt}_{Exp}(c, m, E_i) \\
op_u\ E_1 & \Rightarrow & \{V^{bt}_E = V^{bt}_{E_1}\} \bigcup C^{bt}_{Exp}(c, m, E_1) \\
E_1\texttt{?}E_2\texttt{:}E_3 & \Rightarrow & \{primitive(E),\, V^{bt}_E \sqsupseteq V^{bt}_{E_1}, V^{bt}_E \sqsupseteq V^{bt}_{E_2},\, V^{bt}_E \sqsupseteq V^{bt}_{E_3}\} \bigcup_{i\in[1,3]} C^{bt}_{Exp}(c, m, E_i) \\
E_0.mt\texttt{(}E_1, \ldots, E_n\texttt{)} & \Rightarrow & \{V^{bt}_E = methodApp^{bt}(t^{ct}_{E_0}, mt, (V^{bt}_{E_1}, \ldots, V^{bt}_{E_n}))\} \bigcup_{i\in[0,n]} C^{bt}_{Exp}(c, m, E_i) \\
 & & \\
C^{bt}_{Mthd}(c, \_\, m(\overline{X\texttt{,}})\texttt{\{return}\ E\texttt{;\}}) & = & \{primitive(E),\, V^{bt}_{c.m.\texttt{return}} \sqsupseteq V^{bt}_E\} \bigcup C^{bt}_{Exp}(c, m, E) \\
 & & \\
C^{bt}_{Class}(\texttt{class}\ c \ldots \texttt{\{}\_\ \_\ \overline{M}\texttt{\}}) & = & \bigcup_{M\ \in\ \overline{M}} C^{bt}_{Mthd}(c, M) \\
 & & \\
C^{bt}_{Prg}(\overline{C}\ \_\ E\ \_\ \overline{x_n}) & = & \bigcup_{C\ \in\ \overline{C}} C^{bt}_{Class}(C) \bigcup C^{bt}_{Exp}(\_, \_, E_{main}) \bigcup_{i\in[1,n]} V^{bt}_{\_.\_.x_i} = t^{bt}_{\_.\_.x_i}
\end{array}
$$

FIG. 6.20 – Temps de liaison : Générateur de contraintes.

Expressions (C^{bt}_{Exp}). Les constantes, étant de type primitif, sont annotées comme statiques, c'est-à-dire $\mathcal{S}$. Pour une variable de type primitif, le temps de liaison est égal au temps de liaison du paramètre de la méthode correspondante.

Tout comme dans la génération des contraintes $\mathcal{C}^{ct}$, les contraintes sur les expressions d'accès aux champs sont générées en appliquant l'opérateur $fieldAccess^{bt}$ (voir figure 6.19).

L'expression d'instanciation d'une classe est une construction de type objet donc seul les contraintes sur les paramètres de type primitif sont générées

Dans le cas des expressions qui représentent les opérations binaires et unaires, en tant qu'opérateurs appliquées sur des sous-expressions uniquement de type primitif, le temps de liaison dépend de celui des sous-expressions correspondantes.

L'annotation de l'expression conditionnelle dépend du temps de liaison de la condition. Celle-ci étant du type primitif `boolean`, la valeur de temps de liaison se trouve dans le domaine π, c'est-à-dire soit statique ($\mathcal{S}$), soit dynamique ($\mathcal{D}$). Si la sous-expression représentant la condition est statique l'annotation sur l'expression englobante dépend des

annotations associées aux branches, à savoir $V^{bt}_{E_2}$ et $V^{bt}_{E_3}$. Autrement, l'expression conditionnelle est annotée comme dynamique.

Les contraintes générées à partir de l'expression d'appel de méthodes sont obtenues en appliquant l'opérateur $methodApp^{bt}$ qui permet aussi la génération des contraintes associées aux paramètres des méthodes (voir figure 6.19).

Méthodes (C^{bt}_{Mthd}). En ce qui concerne l'instruction de retour d'une méthode le temps de liaison est lié au temps de liaison de l'expression associée au corps de la méthode.

Classes (C^{bt}_{Class}). L'ensemble des contraintes produites par le générateur C^{bt}_{Class} est l'ensemble des contraintes générées pour toutes les méthodes de la classe.

Programmes (C^{bt}_{Prg}). Finalement, l'ensemble de contraintes généré est constitué par les contraintes générées pour chacune des classes du programme, celles produites à partir de l'expression initiale et les contraintes représentant les annotations sur les variables libres de l'expression initiale.

6.2.2.3 Résolution des contraintes

Dans le contexte de l'analyse de temps de liaison, nous sommes intéressés par la solution la moins dynamique qui satisfasse l'ensemble de contraintes.

$$V^{bt}_{c_a} \sqsupseteq fieldAccess^{bt}(t^{ct}_{c_{a_0}}, f) \iff \forall\, c \in t^{ct}_{c_{a_0}} \rightarrow \{V^{bt}_{c_a} \sqsupseteq V^{bt}_{c.f}\}$$

$$V^{bt}_{c_a} \sqsupseteq methodApp^{bt}(t^{ct}_{c_{a_0}}, m, (V^{bt}_{c_{a_1}}, \ldots, V^{bt}_{c_{a_n}})) \iff \forall\, c \in V^{ct}_{c_{a_0}},\ \_\ m(\overline{\_\ x_n\ ,})\{S;\} \in methods(c) \rightarrow \{V^{bt}_{c_a} \sqsupseteq V^{bt}_{c.m.\texttt{return}}\} \bigcup_{i\in[1,n]} \{V^{bt}_{c.m.x_i} \sqsupseteq V^{bt}_{c_{a_i}}\}$$

FIG. 6.21 – Temps de liaison : règles de résolution des contraintes.

L'ensemble initial de contraintes, produit par le générateur de la figure 6.20, est simplifié en appliquant les règles de résolution de contraintes de la figure 6.21 sur les contraintes formées à partir des opérateurs conditionnels (voir figure 6.18). Cela permet d'obtenir un nouvel ensemble de contraintes dans lequel toutes les contraintes sont de la forme $V^{bt}_{c_a} \sqsupseteq bt$, comme mentionné dans la section 6.2.2.1. La décomposition proposée par ces règles se base sur la sémantique des contraintes donnée figure 6.19.

6.2.2.4 Utilisation du solveur REQS

Tout comme dans l'analyse de type concret, les annotations de temps de liaison d'un programme peuvent être calculées en utilisant le solveur REQS.

Treillis. Rappelons que le domaine des annotations du temps de liaison BT est l'union du domaine des annotation de type concrets et du domaine π (voir équation 6.2). Pour cette raison, il suffit de trouver un treillis qui est conforme à l'ordonnancement stipulé par la relation $\sqsupseteq$ pour les variables de contraintes dont le domaine est π. Parmi les treillis prédéfinis, REQS inclut le treillis $T2$, lequel a deux éléments $\bot$ et $\top$, tel que $\bot \sqsupseteq \top$.

Règles de conversion vers REQS. Ici, nous détaillons les règles qui permettent de traduire les contraintes conditionnelles vers une syntaxe traitable par le solveur REQS comme le montre la figure 6.22. La définition des règles $\textbf{FIELD}^{bt}_{\overrightarrow{reqs}}$, $\textbf{INVK}^{bt}_{\overrightarrow{reqs}}$ et $\textbf{NEW}^{bt}_{\overrightarrow{reqs}}$ reprend la forme des règles de résolution correspondantes présentées dans la figure 6.21.

$$\frac{V^{bt}_{c_a} = {fieldAccess}^{bt}(t^{ct}_{c_{a_0}}, f)}{V^{bt}_{c_a} \sqsupseteq V^{bt}_{c.f}} \quad [\textbf{FIELD}^{bt}_{\overrightarrow{reqs}}] \qquad \forall\, c \in t^{ct}_{c_{a_0}}$$

$$\frac{V^{bt}_{c_a} = {methodApp}^{bt}(t^{ct}_{c_{a_0}}, m, (V^{bt}_{c_{a_1}}, \ldots, V^{bt}_{c_{a_n}}))}{V^{bt}_{c_a} \sqsupseteq V^{bt}_{c.m.\texttt{return}} \qquad V^{bt}_{c.m.x_i} \sqsupseteq V^{bt}_{c_{a_i}} \;\; \forall\, i \in [1,n]} \quad [\textbf{INVK}^{bt}_{\overrightarrow{reqs}}] \quad \forall\, c \in t^{ct}_{c_{a_0}}$$

FIG. 6.22 – Temps de liaison : règles de conversion vers REQS.

Tout comme dans le contexte de l'analyse de types concrets (voir figure 6.8), les règles de conversion conservent la relation d'inégalité spécifiée originellement. Notez, par contre, que la contrainte résultante n'inclut pas l'opérateur conditionnel fourni par REQS. Cela est une conséquence de l'approche de résolution des contraintes lors de l'analyse de programmes. Comme la solution liée aux contraintes de l'analyse de types concrets est disponible lors de la recherche d'une solution pour l'ensemble $\mathcal{C}^{bt}$, la condition $\forall\, c \in t^{ct}_{c_{a_0}}$ est prise en compte au moment de la conversion des contraintes. Autrement dit, la condition ne fait pas partie de la définition de la contrainte résultante car, contrairement à l'analyse de types concrets, l'ensemble des classes concernées dans la condition, représenté par $t^{ct}_{c_{a_0}}$, est connu avant la résolution des contraintes $\mathcal{C}^{bt}$.

Dans la section 11.1.3, nous verrons comment les ensembles de contraintes obtenus lors de différentes étapes de l'analyse peuvent être intégrés au sein d'un même processus de résolution. Afin de rendre l'ensemble des contraintes résultant de la conversion proposée traitable par le solveur REQS, toutes les contraintes doivent être exprimées sous la forme d'équations. Ceci demande à réécrire les inégalités en égalités de le même façon que pour les contraintes $\mathcal{C}^{ct}$ (voir section 6.1.2.4).

Reprenons l'ensemble de contraintes $V^{bt}_{c.f} \sqsupseteq V^{bt}_{E_1}$ et $V^{bt}_{c.f} \sqsupseteq V^{bt}_{E_2}$, mais dans ce cas en reliant les variables de temps de liaison. Cet ensemble exprime le fait que le temps de liaison de la construction $c.f$ doit être plus dynamique ou égal au temps de liaison affecté aux constructions E_1 et E_2.

La figure 6.23(a) montre la résolution des contraintes mentionnées auparavant basées sur la relation d'inégalité. Observons que le temps affecté à la construction $c.f$ est la valeur $\mathcal{D}$ malgré la contrainte (4) où la variable de contrainte $V^{bt}_{E_1}$ reçoit la valeur $\mathcal{S}$. Cela est

(a) Contraintes.

	Contraintes	Solution la plus statique
(1)	$V_{E_1}^{bt} \sqsupseteq \mathcal{D}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\_$, $V_{c.f}^{bt} :\_$
(2)	$V_{E_2}^{bt} \sqsupseteq \mathcal{S}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\_$
(3)	$V_{c.f}^{bt} \sqsupseteq V_{E_1}^{bt}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\mathcal{D}$
(4)	$V_{c.f}^{bt} \sqsupseteq V_{E_2}^{bt}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\mathcal{D}$

(b) Contraintes sous la forme d'égalités.

	Équations	Solution REQS
(1)	$V_{E_1}^{bt} = \mathcal{D}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\_$, $V_{c.f}^{bt} :\_$
(2)	$V_{E_2}^{bt} = \mathcal{S}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\_$
(3)	$V_{c.f}^{bt} = V_{E_1}^{bt}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\mathcal{D}$
(4)	$V_{c.f}^{bt} = V_{E_2}^{bt}$	$V_{E_1}^{bt} :\mathcal{D}$, $V_{E_2}^{bt} :\mathcal{S}$, $V_{c.f}^{bt} :\mathcal{S}$

(c) Contraintes sous la forme des équations REQS.

	Équations	Solution REQS
(1)	$V_{E_1}^{bt} =$ `'T`	$V_{E_1}^{bt} :$`'T`, $V_{E_2}^{bt} :\_$, $V_{c.f}^{bt} :\_$
(2)	$V_{E_2}^{bt} =$ `'B`	$V_{E_1}^{bt} :$`'T`, $V_{E_2}^{bt} :$`'B`, $V_{c.f}^{bt} :\_$
(3)	$V_{c.f}^{bt =}$ `(UNION` $V_{E_1}^{bt}$ $V_{E_1}^{bt}$`)`	$V_{E_1}^{bt} :$`'T`, $V_{E_2}^{bt} :$`'B`, $V_{c.f}^{bt} :$`'T` $\sqcup$ `'B` $=$ `'T`

FIG. 6.23 – Temps de liaison : conversion des inégalités en égalités.

une conséquence de la recherche de la solution la plus statique qui satisfasse l'ensemble de contraintes par rapport au treillis mentionné ci-dessus (voir figure 6.23). En remplaçant la relation $\sqsupseteq$ par une inégalité, afin de convertir les contraintes en équations REQS, on obtient le système d'équations de la figure 6.23(b). Rappelons que si la même variable apparaît dans la partie gauche de plusieurs équations alors REQS ne prend en compte que la dernière équation fournie. Pour cette raison, la solution calculée par REQS par rapport aux contraintes de la figure 6.23(b) ne correspond pas à la solution souhaitée car la construction $c.f$ est finalement affectée à la valeur $\mathcal{S}$. Rappelons aussi qu'une construction est statique si elle ne dépend que de constructions statiques. En effet, la valeur de la construction $c.f$ n'est plus que le plus petit majorant des valeurs affectées, sur la base du treillis mentionné ci-dessus, c'est-à-dire $V_{E_1}^{bt} \sqcup V_{E_2}^{bt}$. La figure 6.23(c) montre l'ensemble d'équations résultant de l'union des contraintes qui affectent une même construction à travers l'opérateur $\sqcup$, qui dans la syntaxe de REQS est assuré par l'opérateur `UNION` du treillis prédéfini T2. Dans la syntaxe proposée par REQS, les valeurs $\bot$ (statique) et $\top$ (dynamique) sont représentées par les constantes `'B` et `'T`.

Pour cette raison, les contraintes modifiées par les règles de conversion de la figure 6.22 sont finalement traduites en appliquant la règle **I-TO-E**$^{bt}_{\overline{reqs}}$ de la figure 6.24, afin d'éliminer les inégalités des contraintes.

$$\frac{V_{ca}^{bt} \sqsupseteq bt_1, \ldots, V_{ca}^{bt} \sqsupseteq bt_i, \ldots, V_{ca}^{bt} \sqsupseteq bt_n}{V_{ca}^{bt} = \texttt{(UNION } bt_1 \ldots \texttt{(UNION } bt_i \ldots\texttt{) } bt_n\texttt{)} \ldots} \quad [\textbf{I-TO-E}^{bt}_{\overrightarrow{reqs}}]$$

FIG. 6.24 – Temps de liaison : conversion d'inégalités en égalités.

6.2.2.5 Temps de liaison sur le programme *Point2DStepping*

Dans cette section nous illustrons l'analyse de temps de liaison sur le programme *Point2DStepping*. Pour cette raison, nous allons reprendre le résultat de l'analyse de types concrets sur ce programme effectuée dans la section 6.1.2.5.

Génération des contraintes. Reprenons l'expression initiale E_{main} donnée dans la section 6.1.2.5 :

$$\begin{array}{rcl} E_{main} & = & \texttt{new Point2DX(x}_1\texttt{, y}_1\texttt{, new Stepper(step}_1\texttt{)).next(offset}_1\texttt{).x +} \\ & & \texttt{new Point2D2(x}_2\texttt{, y}_2\texttt{, new StepperBy2(step}_2\texttt{)).next(offset}_2\texttt{).y} \end{array}$$

où les variables libres ont initialement les temps de liaison suivants :

$$\begin{array}{rcl} \Gamma_1^{bt} & = & \{\texttt{x}_1 \mapsto \mathcal{S}, \texttt{y}_1 \mapsto \mathcal{D}, \texttt{step}_1 \mapsto \mathcal{D}, \texttt{offset}_1 \mapsto \mathcal{S}, \\ & & \texttt{x}_2 \mapsto \mathcal{D}, \texttt{y}_2 \mapsto \mathcal{S}, \texttt{step}_2 \mapsto \mathcal{S}, \texttt{offset}_2 \mapsto \mathcal{S}\} \end{array}$$

Ici, nous décrivons la génération et la résolution des contraintes concernant les méthodes afin de mieux comprendre l'utilisation de l'information de type concret lorsque les contraintes pour l'analyse de temps de liaison sont générées. La figure 6.25 présente partiellement les classes `Point2DX` et `Point2D2` annotées selon le résultat de l'inférence de types concrets.

Les contraintes sont générées à partir de l'expression suivante :

$$\begin{array}{rcl} C_{Prg}^{bt}(\mathit{Point2DStepping}) & = & \ldots \\ & & \ldots \\ & & \ldots \cup C_{Mthd}^{bt}(\texttt{Point2DX}, \texttt{Point2DX.dec}) \\ & & \ldots \cup C_{Mthd}^{bt}(\texttt{Point2D2}, \texttt{Point2D2.dec}) \\ & & \ldots \end{array}$$

De même que dans l'analyse de types concrets, le nombre de contraintes est assez élevé donc nous allons étudier seulement les contraintes générées à partir des méthodes `dec` définies dans les classes `Point2DX` et `Point2D2`. Ces contraintes sont données dans le tableau 6.2.

De l'application de la règle de conversion vers REQS [$\textbf{INVK}^{bt}_{\overrightarrow{reqs}}$] de la figure 6.22, on obtient les équations du tableau 6.3. En plus, à partir de l'expression initiale, on sait que seule une instance des classes `Stepper` et `StepperBy2` est créée et associée, ensuite, aux instances des classes `Point2DX` et `Point2D2`, respectivement. Ceci permet de déduire (sans décrire l'ensemble des contraintes dans sa totalité) que l'appel de la méthode `step` dans

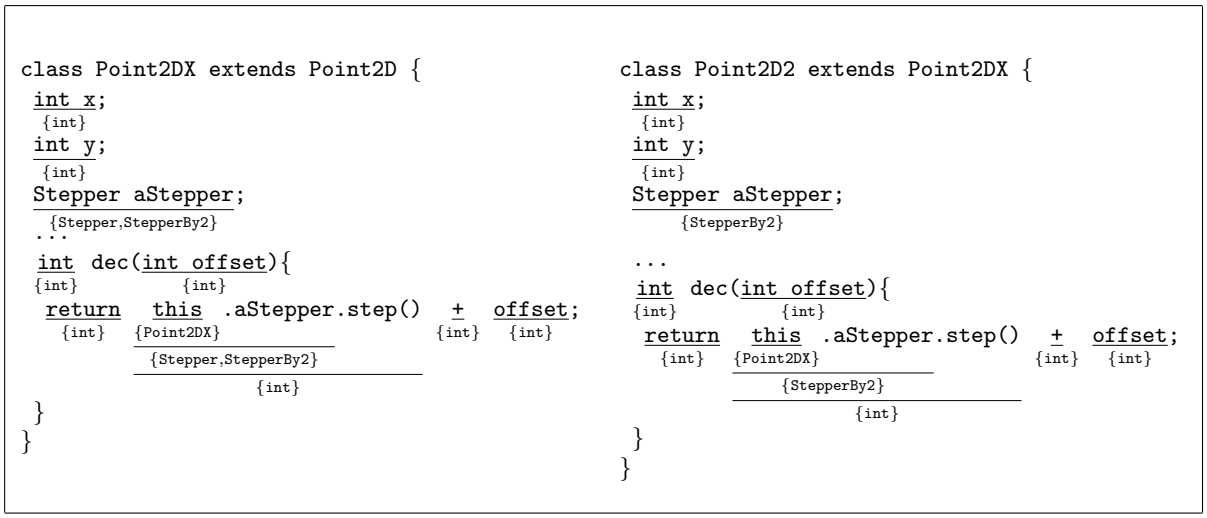


FIG. 6.25 – Type concret : annotations des classes `Point2DX` et `Point2D2`.

la définition des méthodes `dec` sera annoté comme dynamique si le receveur est de type `Stepper` et comme statique si le receveur est de type `StepperBy2`. Pour cela il suffit de prendre en compte le fait que les instances sont initialisées avec une valeur dynamique dans le premier cas (*i.e.*, $\Gamma_1^{bt}(\texttt{step}_1) = \mathcal{D}$) et une valeur statique dans le deuxième cas (*i.e.*, $\Gamma_1^{bt}(\texttt{step}_2) = \mathcal{S}$).

Notez que l'annotation du receveur de l'appel de la méthode `step` dans la méthode `dec` de la classe `Point2DX` est l'ensemble $\{\texttt{Stepper}, \texttt{StepperBy2}\}$. Par conséquent, comme le montre l'équation ($d_{1\cup 2}$) du tableau 6.3, cet appel est annoté dynamique, ce qui affecte ultérieurement l'annotation sur la définition *initiale* de la méthode `next` de la classe `Point2DX`. En ce qui concerne la définition *héritée* de cette méthode dans la classe `Point2D2`, l'annotation reste statique. La figure 6.26 montre les annotations de temps de liaison sur les classes `Point2DX` et `Point2D2` calculées à travers la résolution des contraintes résultant de l'analyse.

6.3 Analyse de temps d'évaluation

La spécialisation d'un programme basée sur l'analyse de temps de liaison est effectuée, traditionnellement, en évaluant les constructions statiques et en résidualisant les constructions dynamiques. Cependant, l'application de cette approche sur le programme annoté de la figure 6.26 s'avère *a priori* impossible en l'absence d'une telle annotation (statique ou dynamique) pour les expressions de type objet.

Ici, nous considérons la création d'objets comme une expression statique puisque toutes les instances peuvent être créées au moment de la spécialisation. Dans le cas où l'instance créée contient des champs dynamiques et des champs statiques, l'instance est traitée comme étant, à la fois, statique et dynamique. En conséquence, l'expression sera exécutée

Générateur	Éléments de $\mathcal{C}^{bt}$	
$C^{bt}_{Mthd}(\texttt{Point2DX}, \texttt{int dec}\{\texttt{return } E\})$	$V^{bt}_{\texttt{Point2DX.dec.return}} \sqsupseteq V^{bt}_{E}$	(a)
$E = E_1 + E_2$	$V^{bt}_{\texttt{Point2DX.dec.}E} \sqsupseteq V^{bt}_{\texttt{Point2DX.dec.}E_1}$,	(b)
	$V^{bt}_{\texttt{Point2DX.dec.}E} \sqsupseteq V^{bt}_{\texttt{Point2DX.dec.}E_2}$	(c)
$C^{bt}_{Exp}(\texttt{Point2DX}, M_{Point2DX.dec}, E_1)$		
$E_1 = E_{10}\texttt{.step()}$	$V^{bt}_{\texttt{Point2DX.dec.}E_1} = methodApp^{bt}(t^{ct}_{\texttt{Point2DX.dec.}E_{10}}, \texttt{step}, ())$	(d)
$C^{bt}_{Exp}(\texttt{Point2DX}, M_{Point2DX.dec}, E_{10})$		
$E_{10} = E_{100}\texttt{.aStepper}$	$V^{bt}_{\texttt{Point2DX.dec.}E_{10}} = t^{ct}_{\texttt{Point2DX.dec.}E_{10}}$	(e)
$C^{bt}_{Exp}(\texttt{Point2DX}, M_{Point2DX.dec}, E_{100})$		
$E_{100} = \texttt{this}$	$V^{bt}_{\texttt{Point2DX.dec.}E_{100}} = t^{ct}_{\texttt{Point2DX.dec.}E_{100}}$	(f)
$C^{bt}_{Exp}(\texttt{Point2DX}, M_{Point2DX.dec}, E_2)$		
$E_2 = \texttt{offset}$	$V^{bt}_{\texttt{Point2DX.dec.}E_2} = V^{bt}_{\texttt{Point2DX.dec.offset}}$	(g)
$C^{bt}_{Mthd}(\texttt{Point2D2int dec}\{\texttt{return } E\})$	$V^{bt}_{\texttt{Point2D2.dec.return}} \sqsupseteq V^{bt}_{E}$	(h)
$E = E_1 + E_2$	$V^{bt}_{\texttt{Point2D2.dec.}E} \sqsupseteq V^{bt}_{\texttt{Point2D2.dec.}E_1}$,	(i)
	$V^{bt}_{\texttt{Point2D2.dec.}E} \sqsupseteq V^{bt}_{\texttt{Point2D2.dec.}E_2}$	(j)
$C^{bt}_{Exp}(\texttt{Point2D2}, M_{Point2D2.dec}, E_1)$		
$E_1 = E_{11} * E_{12}$	$V^{bt}_{\texttt{Point2D2.dec.}E_1} \sqsupseteq V^{bt}_{\texttt{Point2D2.dec.}E_{11}}$,	(k)
	$V^{bt}_{\texttt{Point2D2.dec.}E} \sqsupseteq V^{bt}_{\texttt{Point2D2.dec.}E_{12}}$	(m)
$C^{bt}_{Exp}(\texttt{Point2D2}, M_{Point2D2.dec}, E_{11})$		
$E_{11} = E_{110}\texttt{.step()}$	$V^{bt}_{\texttt{Point2D2.dec.}E_{11}} = methodApp^{bt}(t^{ct}_{\texttt{Point2D2.dec.}E_{110}}, \texttt{step}, ())$	(n)
$C^{bt}_{Exp}(\texttt{Point2D2}, M_{Point2D2.dec}, E_{110})$		
$E_{110} = E_{1100}\texttt{.aStepper}$	$V^{bt}_{\texttt{Point2D2.dec.}E_{110}} = t^{ct}_{\texttt{Point2D2.dec.}E_{110}}$	(o)
$C^{bt}_{Exp}(\texttt{Point2D2}, M_{Point2D2.dec}, E_{1100})$		
$E_{1100} = \texttt{this}$	$V^{bt}_{\texttt{Point2D2.dec.}E_{1100}} = t^{ct}_{\texttt{Point2D2.dec.}E_{1100}}$	(p)
$C^{bt}_{Exp}(\texttt{Point2D2}, M_{Point2D2.dec}, E_2)$		
$E_2 = \texttt{offset}$	$V^{bt}_{\texttt{Point2D2.dec.}E_2} = V^{bt}_{\texttt{Point2D2.dec.offset}}$	(q)

TAB. 6.2 – Temps de liaison : génération des contraintes à partir des méthodes `dec` des classes `Point2DX` et `Point2D2`.

à la spécialisation ainsi que résidualisée. Dans le premier cas, l'expression est recréée en utilisant des valeurs quelconques pour l'initialisation des champs dynamiques car ils ne peuvent qu'être utilisés dans des contexte dynamiques et, donc, leur valeur ne sera jamais *liftée*. Rappelons que nous considérons une spécialisation comportementale et que, en conséquence, la signature des constructeurs reste inchangée (voir section 5.2.2).

Pour cette raison, nous proposons d'appliquer l'analyse de temps d'évaluation (voir section 2.1.2.2) afin d'annoter les constructions du programme comme purement statique ($\mathcal{S}$) si elles doivent être évaluées à la spécialisation, comme purement dynamique ($\mathcal{D}$) si elles doivent être évaluées à l'exécution du programme résiduel, et comme à la fois statique et dynamique ($\mathcal{SD}$) si elle doivent être évaluées deux fois, à la spécialisation et à l'exécution. Comme expliqué dans la section 2.1.2.2, l'analyse de temps d'évaluation est une analyse arrière qui propage l'information de temps de liaison des utilisations des variables vers leur définition. Ainsi, chaque variable est annotée en prenant en compte le temps de liaison de la variable et du contexte où elle est utilisée. Elle permet de calculer le temps d'évaluation des définitions de variables. Rappelons que, dans le cadre du langage choisi EFJ, la définition de variables (affectation) n'existe que lors du passage de valeurs

Variable		Expression	
		// $t^{ct}_{\texttt{Point2DX.Point2DX.}E_{10}} = \{\texttt{Stepper,StepperBy2}\}$	
$V^{bt}_{\texttt{Point2DX.Point2DX.}E_1}$	$\sqsupseteq$	$V^{bt}_{\texttt{Stepper.step.return}}$	(d_1)
$V^{bt}_{\texttt{Point2DX.Point2DX.}E_1}$	$\sqsupseteq$	$V^{bt}_{\texttt{StepperBy2.step.return}}$	(d_2)
$V^{bt}_{\texttt{Point2DX.Point2DX.}E_1}$	$=$	(UNION $V^{bt}_{\texttt{StepperBy2.step.return}}$ $V^{bt}_{\texttt{Stepper.step.return}}$)	$(\mathrm{d}_{1\cup 2})$
		// $t^{ct}_{\texttt{Point2D2.Point2D2.}E_{110}} = \{\texttt{StepperBy2}\}$	
$V^{bt}_{\texttt{Point2D2.Point2D2.}E_{11}}$	$=$	$V^{bt}_{\texttt{StepperBy2.step.return}}$	(n_1)

TAB. 6.3 – Temps de liaison : conversion vers REQS de la contrainte conditionnelle *(d)* et *(n)* du tableau 6.2

à travers des paramètres.

Tandis que le temps de liaison détermine quelles sont les constructions qui *peuvent* être spécialisées ou résidualisées, le temps d'évaluation détermine quelles sont les constructions qui *doivent* être spécialisées et/ou résidualisées.

Domaine des annotations. Le domaine des annotations de temps d'évaluation est défini par l'ensemble des parties de l'ensemble π avec comme relation d'ordre l'inclusion [HN00].

$$\pi = \{\mathcal{S}, \mathcal{D}\} \tag{6.4}$$

$$ET = \mathcal{P}(\pi) = \{\{\}, \{\mathcal{S}\}, \{\mathcal{D}\}, \{\mathcal{S}, \mathcal{D}\}\} \tag{6.5}$$

Les annotations de temps d'évaluation sont liées aux constructions en utilisant l'environnement Γ^{et}. Ainsi, le temps d'évaluation $t^{et}_{c_a}$ est associé à la construction c_a par l'environnement $\Gamma^{et} : C_A \rightarrow ET$ utilisé dans des jugements de la forme $\Gamma^{et} \vdash c_a : t^{et}_{c_a}$.

6.3.1 Règles de bonne annotation de temps d'évaluation

Dans cette section nous présentons les règles qui expriment les relations des annotations de temps d'évaluation d'un programme. Cette analyse permet, concrètement, de prendre en compte le contexte d'utilisation des constructions d'un programme.

Dans le cas des constructions de type primitif l'analyse devient extrêmement simple. Quelque soit le temps d'évaluation du contexte de la construction, le temps d'évaluation associé à la construction sera le temps de liaison correspondant. En conséquence, si le contexte est dynamique, alors que le temps de liaison de la construction est statique et donc que la construction *peut* être calculée, sa valeur sera finalement *liftée* afin d'être résidualisée. Par contre, dans le cas où le temps de liaison de la construction est dynamique, la construction sera simplement résidualisée. Notez qu'il n'est pas possible d'avoir une construction dont le temps de liaison est dynamique dans un contexte statique parce que, simplement, on ne peut pas forcer l'évaluation d'une construction dont on ignore la valeur correspondante au moment de la spécialisation.

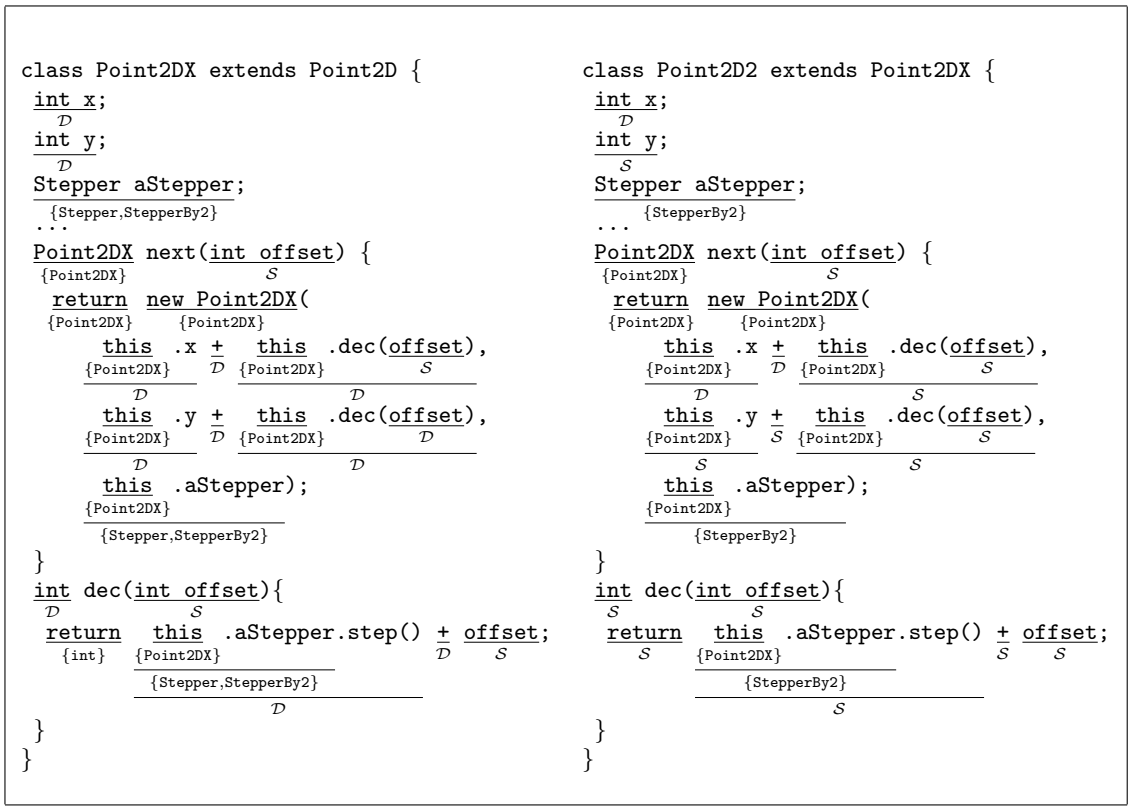


FIG. 6.26 – Temps de liaison : annotations des classes `Point2DX` et `Point2D2`.

Par exemple, considérons les annotations de temps de liaison de la méthode `m` de la figure 6.27(a) on peut calculer le temps d'évaluation comme le montre la figure 6.27(b). Dans ce cas, il n'y a pas de propagation proprement dite. Il s'agit d'expressions primitives et donc le temps d'évaluation dépend du temps de liaison. Par exemple, l'utilisation du paramètre `x` dans l'expression de retour sera remplacée par la valeur correspondante lors de la résidualisation.

Au contraire des constructions primitives, les constructions de type objet seront annotées en fonction du temps d'évaluation de leur contexte qui est à son tour propagé en arrière. Prenons, par exemple, la méthode `m` de la figure 6.28(a). Dans ce cas on observe que le paramètre `p` de type objet est utilisé dans un contexte dynamique lorsque le champ `x` est accédé et dans un contexte statique lorsque le champ `y` est accédé. En conséquence, le paramètre `p` est annoté comme $\{\mathcal{S}, \mathcal{D}\}$. Cela implique que celui-ci devra être résidualisé dans le premier cas, afin de pouvoir résidualiser l'accès au champ, et évalué dans le deuxième cas, pour remplacer l'expression par la valeur concrète du champ (*i.e.*, champ `y`).

Dans les exemples ci-dessus, on observe que le temps d'évaluation peut être calculé, d'abord, en calculant l'annotation des expressions primitives et, ensuite, en propageant ces annotations si les expressions font partie d'une expression englobante comme, par exemple, l'accès à un champ. Les annotations sur les constructions de type primitif sont,

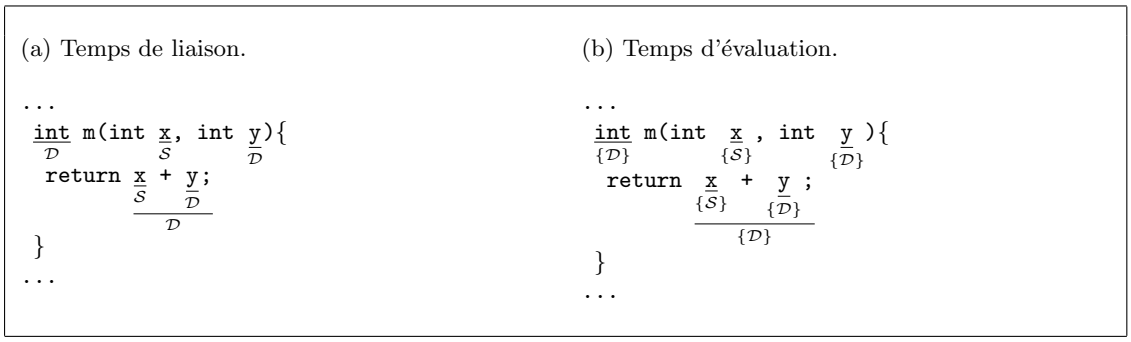


FIG. 6.27 – Temps d'évaluation : annotations de constructions de type primitif.

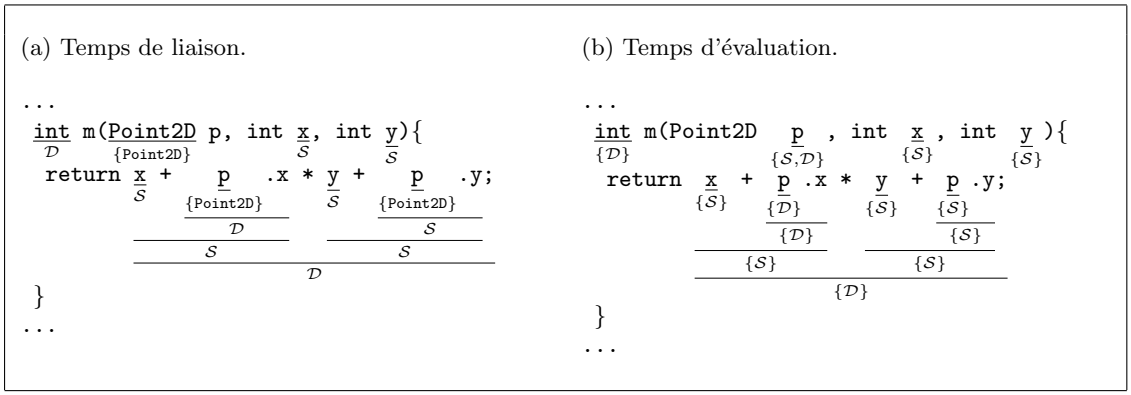


FIG. 6.28 – Temps d'évaluation : annotations de constructions de type objet.

de manière générale, modélisées par la règle $\mathbf{Prm}^{et}$ de la figure 6.29. Cette règle indique, concrètement, le passage du domaine BT (voir définition 6.2) vers le domaine ensembliste ET (voir définition 6.5). Les règles de bonne annotation de temps d'évaluation sur les constructions de type objet sont décrites dans la figure 6.30.

$$\frac{primitive(c_a) \qquad \Gamma^{bt} \vdash c_a : t^{bt}_{c_a} \qquad t^{bt}_{c_a} \in t^{et}_{c_a}}{\Gamma^{et} \vdash c_a : t^{et}_{c_a}} \quad [\mathbf{Prm}^{et}]$$

FIG. 6.29 – Temps d'évaluation : règles sur les constructions de type primitif.

Expressions. Une fois le temps d'évaluation d'une expression calculé, il doit être propagé en arrière, c'est-à-dire vers les sous-expression selon le cas. Dans les règles suivantes, le temps d'évaluation des expressions représenté par t^{et}_E fait partie des prémisses des règles et est donc le point de départ de la propagation en arrière. Notez que, à l'exception des

règles sur l'accès à un champ (règle **FIELD**^{et}) et l'appel d'une méthode (règle **INVK**^{et}), les règles restantes ne concernent que les constructions de type objet. De plus, car il s'agit d'expressions de type primitif avec sous-expressions de type primitif, la figure n'inclut pas de règles pour les expressions représentant les opérations unaires ou binaires. Dans ces derniers cas, l'annotation de temps d'évaluation se base uniquement sur la règle **Prm**^{et}.

Expressions :

$$\Gamma^{et} \vdash \texttt{this} : t^{et}_{E} \quad [\textbf{THIS}^{et}]$$

$$\frac{\neg \mathit{primitive}(c.m.x) \qquad \Gamma^{et} \vdash x : t^{et}_{x} \qquad t^{et}_{c.m.x} \supseteq t^{et}_{x}}{\Gamma^{et} \vdash c.m.x : t^{et}_{c.m.x}} \quad [\textbf{VAR}^{et}]$$

$$\frac{\Gamma^{ct} \vdash E_0 : t^{ct}_{E_0} \qquad \Gamma^{et} \vdash (E_0.f) : t^{et}_{E} \qquad t^{ct}_{E_0} \supseteq t^{et}_{E} \qquad \forall\, c \;\in\; t^{ct}_{E_0}\ (t^{et}_{c.f} \supseteq t^{et}_{E})}{\Gamma^{et} \vdash (E_0.f) : t^{et}_{E}} \quad [\textbf{FIELD}^{et}]$$

$$\frac{\Gamma^{et} \vdash (\texttt{new}\ c(\overline{E})) : t^{et}_{E} \qquad \neg \mathit{primitive}(E_i) \qquad t^{et}_{E_i} \supseteq t^{et}_{E}}{\Gamma^{et} \vdash E_i : t^{et}_{E_i}} \quad [\textbf{NEW}^{et}] \qquad \frac{\Gamma^{et} \vdash ((c)E_1) : t^{et}_{E} \qquad t^{et}_{E_1} \supseteq t^{et}_{E}}{\Gamma^{et} \vdash E_1 : t^{et}_{E_1}} \quad [\textbf{CAST}^{et}]$$

$$\frac{\neg \mathit{primitive}(E_1 \texttt{?} E_2 \texttt{:} E_3) \qquad \Gamma^{et} \vdash (E_1 \texttt{?} E_2 \texttt{:} E_3) : t^{et}_{E} \qquad t^{et}_{E_2} \supseteq t^{et}_{E} \qquad t^{et}_{E_3} \supseteq t^{et}_{E}}{\Gamma^{et} \vdash E_2 : t^{et}_{E_2} \qquad \Gamma^{et} \vdash E_3 : t^{et}_{E_3}} \quad [\textbf{COND}^{et}]$$

$$\frac{\begin{array}{c}\Gamma^{ct} \vdash E_0 : t^{ct}_{E_0} \qquad \Gamma^{et} \vdash (E_0.m(\overline{E,})) : t^{et}_{E} \qquad t^{ct}_{E_0} \supseteq t^{et}_{E} \\ \forall\, c \in t^{ct}_{E_0}(t^{et}_{c.m.\texttt{return}} \supseteq t^{et}_{E} \mid \neg \mathit{primitive}(c.m.\texttt{return}), t^{et}_{E_i} \supseteq t^{et}_{E} \mid \neg \mathit{primitive}(E_i))\end{array}}{\Gamma^{et} \vdash c.m.\texttt{return} : t^{et}_{c.m.\texttt{return}}} \quad [\textbf{INVK}^{et}]$$

Méthodes :

$$\frac{\Gamma^{et} \vdash E : t^{et}_{E} \qquad t^{et}_{E} \supseteq \Gamma^{et}(c.m.\texttt{return})}{\Gamma^{et} \vdash t\ m(\overline{t\ x,})\{\texttt{return}\ E\}\ \textit{well-et-annotated in}\ c} \quad [\textbf{MTHD}^{et}]$$

Classes :

$$\frac{\Gamma^{et} \vdash \overline{M}\ \textit{well-et-annotated in}\ c}{\Gamma^{et} \vdash \texttt{class}\ c\ \texttt{extends}\ \_\ \{\_\ \_\ \overline{M}\}\ \textit{well-et-annotated}} \quad [\textbf{CLS}^{et}]$$

Programmes :

$$\frac{\Gamma^{et}_{0}(E) : t^{et}_{E} \qquad \Gamma^{et}_{0} \vdash \overline{C}\ \textit{well-et-annotated}}{\Gamma^{et}_{0} \vdash \overline{C}\ t\ E\ \overline{t\ x}\ \textit{well-et-annotated}} \quad [\textbf{PRG}^{et}]$$

Fig. 6.30 – Temps d'évaluation : règles sur les programmes EFJ.

L'annotation sur la référence à l'objet (`this`) dépend de la propagation résultant des utilisations dans le code de la méthode analysée (règle **THIS**^{et}). Dans le cas des variables, la règle correspondante ne concerne que les variables de type objet (règle **VAR**^{et}). Une variable est bien annotée si le temps d'évaluation est contenu dans le temps d'évaluation de la définition du paramètre correspondant (*i.e.*, $t^{et}_{c.m.x}$). Tout comme dans les règles restantes cette propagation est exprimée par la relation d'inclusion entre les annotations correspondantes.

Dans le cas de l'expression d'accès à un champ, elle est considérée bien annotée si le temps d'évaluation associé est contenu dans le temps d'évaluation de la sous-expression du receveur (*i.e.*, E_0), comme le montre la règle **FIELD**^{et}.

Comme l'indique la règle **NEW**^{et}, le temps d'évaluation de l'expression d'instanciation

influe directement sur le temps d'évaluation des arguments du constructeur (*i.e.*, E_i).

Dans le cas de la coercition, la bonne annotation de la sous-expression cible de la coercition dépend du temps d'évaluation de l'expression (règle $\textsc{cast}^{et}$).

Le temps d'évaluation de l'expression conditionnelle doit être propagée vers les sous-expressions pour être conforme à la règle de bonne annotation (règle $\textsc{cond}^{et}$). Comme la sous-expression de la condition est toujours de type primitif, le temps d'évaluation se calcule en appliquant la règle $\textbf{Prm}^{et}$.

Finalement, le temps d'évaluation de l'appel d'une méthode affecte la sous-expression qui représente l'objet receveur de l'appel ainsi que les sous-expressions représentant les arguments de l'appel (règle $\textsc{invk}^{et}$). Dans les deux cas, cette règle s'applique si le retour de la méthode concernée est de type objet.

Méthodes. Une méthode est considérée bien annotée si l'expression de l'instruction de retour est bien annotée (règle $\textsc{mthd}^{et}$).

Classes. Une classe est considérée bien annotée si ses méthodes sont bien annotées (règle $\textsc{cls}^{et}$).

Programmes. Finalement, la bonne annotation d'un programme dépend de la bonne annotation des classes dont la propagation est déclenchée par l'annotation associée à l'expression initiale E_{main} donnée par l'environnement Γ_0^{et}.

6.3.2 Analyse par contraintes

Dans cette section nous détaillons l'analyse de temps d'évaluation à base de contraintes en utilisant l'information obtenue par l'analyse de temps de liaison décrite dans la section 6.2.

6.3.2.1 Définition des contraintes

Une variable de contrainte liée à la construction c_a est dénotée $V_{c_a}^{et}$. Tout comme dans les analyses décrites auparavant, les variables de contraintes prennent une forme dépendant de la construction : $V_{c.x}^{et}$, $V_{c.m.\texttt{return}}^{et}$, $V_{c.m.x}^{et}$ et V_E^{et} pour les champs, la valeur de retour des méthodes, les paramètres et les expressions, respectivement.

Les contraintes générées sont de la forme $V_{c_a}^{et} \supseteq et$, avec $et \in ET$ et c_a une construction (de type primitif ou de type objet). Une contrainte $V_{c_a}^{et} \supseteq et$ signifie que l'ensemble de valeurs et doit être un sous-ensemble de celui représenté par la variable $V_{c_a}^{et}$. La syntaxe des contraintes est présentée dans la figure 6.31.

Les contraintes sont créées à partir de constantes de temps d'évaluation ou d'autres variables de contraintes uniquement. Dans cette analyse, les contraintes conditionnelles ne sont pas nécessaires car le flot de contrôle a déjà été pris en compte lors de l'analyse de temps de liaison.

$V_{c_a}^{et}$	$\supseteq$	et	contrainte
et	$:=$	a	constante (élément de ET)
	$\mid$	$V_{c_a}^{et}$	variable de contrainte

FIG. 6.31 – Temps d'évaluation : syntaxe des contraintes.

Sémantique des contraintes. La sémantique formelle des expressions de contraintes de la figure 6.31 est définie par une interprétation $\mathcal{I}^{et}$ reliant les expressions et avec l'annotation de temps d'évaluation correspondante. De même que l'interprétation sur les expressions ensemblistes $\mathcal{I}^{ct}$ des contraintes pour l'inférence de types concrets, $\mathcal{I}^{et}$ représentera une solution sur l'ensemble de contraintes $\mathcal{C}^{et}$ si pour chaque contrainte $V_{c_a}^{et} \supseteq et \in \mathcal{C}^{et}$, $\mathcal{I}^{et}(V_{c_a}^{et}) \supseteq \mathcal{I}^{et}(et)$. La figure 6.32 donne la sémantique des contraintes en utilisant l'interprétation $\mathcal{I}^{et}$.

$\mathcal{I}^{et}(\bot)$	$=$	$\emptyset$
$\mathcal{I}^{et}(a)$	$=$	$\{a\}$
$\mathcal{I}^{et}(V_{c_a}^{et})$	$\subseteq$	ET
$\mathcal{I}^{bt}(\top)$	$=$	ET

FIG. 6.32 – Temps d'évaluation : sémantique des contraintes.

À la différence des analyses abordées auparavant, où il existe aussi des contraintes conditionnelles, les contraintes sont créées uniquement à partir d'une constante de temps d'évaluation ou d'autres variables de contraintes. Par contre, comme nous le verrons dans la définition du générateur (C_{Exp}^{et}), les contraintes générées pour les expressions d'accès à un champ et l'appel à une méthode sont définies en fonction des annotations de type concret des expressions du receveur. L'annotation de type concret est utilisée pour déterminer la propagation du temps d'évaluation. Autrement dit, elles sont définies de la même façon qu'une contrainte conditionnelle mais, comme expliqué auparavant, dans ce cas la propagation va de l'expression proprement dite vers les sous expressions. C'est la raison pour laquelle, afin d'éviter de rendre la définitions des contraintes plus complexe, nous décidons d'inclure la condition dans la définition du générateur.

6.3.2.2 Génération des contraintes

Les contraintes de l'analyse de temps d'évaluation sont dérivées des règles de bonne annotation de la section 6.3.1. L'ensemble de contraintes $\mathcal{C}^{et}$ est généré en appliquant un générateur de contraintes sur la définition des classes d'un programme donné P. La figure 6.33 montre la structure du générateur de contraintes C_{Prg}^{et} basée sur les sous-générateurs associés aux différents types des constructions analysées.

De même que dans l'analyse de temps de liaison, afin de simplifier la description des générateurs, les générateurs de contraintes prennent en compte seulement les constructions

de type objet.

$$
\begin{array}{lcl}
C^{et}_{Exp}(c,m,E) & = & \text{case } E \text{ of} \\
x & \Rightarrow & \{V^{et}_{c.m.x} \supseteq V^{et}_{E}\} \\
E_0.\texttt{x} & \Rightarrow & \{V^{et}_{E_0} \supseteq V^{et}_{E}\} \bigcup_{c \,\in\, t^{ct}_{E_0}} \{V^{et}_{c.x} \supseteq V^{et}_{E}\} \bigcup C^{et}_{Exp}(c,m,E_0) \\
\texttt{new}\ d\texttt{(}E_1,\ldots,E_n\texttt{)} & \Rightarrow & \{V^{et}_{E_i} \supseteq V^{bt}_{E}\} \bigcup C^{et}_{Exp}(c,m,E_i) \\
\texttt{(}d\texttt{)}E_1 & \Rightarrow & \{V^{et}_{E_1} \supseteq V^{et}_{E}\} \bigcup C^{et}_{Exp}(c,m,E_1) \\
E_1\texttt{?}E_2\texttt{:}E_3 & \Rightarrow & \{V^{et}_{E_2} \supseteq V^{et}_{E}, V^{et}_{E_3} \supseteq V^{et}_{E}\} \bigcup C^{et}_{Exp}(c,m,E_i) \\
E_0.mt\texttt{(}E_1,\ldots,E_n\texttt{)} & \Rightarrow & \bigcup \{V^{et}_{E_i} \supseteq V^{et}_{E}\} \bigcup_{c \,\in\, t^{ct}_{E_0}} \{V^{et}_{c.mt.\texttt{return}} \supseteq V^{et}_{E}\} \bigcup C^{et}_{Exp}(c,m,E_i) \\
 & & \\
C^{et}_{Mthd}(C, \_\ m\texttt{(}\overline{X\texttt{,}}\texttt{)\{return}\ E\texttt{;\}}) & = & \{V^{et}_{E} \supseteq V^{et}_{c.m.\texttt{return}}\} \bigcup C^{et}_{Exp}(C,M,E) \\
 & & \\
C^{et}_{Class}(\texttt{class}\ c \ldots \texttt{\{}\_\ \_\ \overline{M}\texttt{\}}) & = & \bigcup_{M \,\in\, \overline{M}} C^{et}_{Mthd}(c,M) \\
 & & \\
C^{et}_{Prg}(P) & = & \bigcup_{C \,\in\, \overline{C}} C^{et}_{Class}(C) \bigcup C^{et}_{Exp}(\_,\_,E_{main})
\end{array}
$$

FIG. 6.33 – Temps d'évaluation : générateur des contraintes.

Expressions (C^{et}_{Exp}). Dans le cas de la référence à l'objet `this`, le temps d'évaluation dépend strictement du contexte dans lequel cette référence est utilisée. Aussi, comme il s'agit d'une expression simple, son annotation n'est propagée vers aucune sous-expression et donc aucune contrainte ne doit être générée.

Le temps d'évaluation associé à l'utilisation d'une variable sera propagé vers sa définition qui, dans le langage EFJ, est la définition du paramètre de la méthode correspondante.

Dans le cas de l'accès d'un champ, le temps d'évaluation sera propagé vers les sous-expressions correspondantes.

L'annotation de l'expression associée à l'instanciation d'une classe a une incidence directe sur le temps d'évaluation des expressions passées par les paramètres du constructeur.

Dans le cas de l'expression conditionnelle, l'annotation du test, étant une expression de type primitif, dépend du temps de liaison correspondant. Par contre, si les sous-expressions qui déterminent le résultat de l'expression englobante, E_2 et E_3, sont de type objet, alors leur temps d'évaluation dépend de celui de l'expression, c'est-à-dire V^{et}_{E}.

Finalement, les contraintes générées à partir de l'expression d'appel de méthodes permettent la propagation du temps d'évaluation de l'expression sur les annotations des expressions passées à travers les paramètres uniquement si elles sont de type objet. Dans le cas de la sous-expression du receveur, E_0, elle sera affectée par le temps d'évaluation de l'expression indépendamment du type de cette dernière.

Méthodes (C^{et}_{Mthd}). En ce qui concerne l'instruction de retour d'une méthode, le temps d'évaluation se propage de la valeur de retour, $\texttt{C.}m\texttt{.return}$, vers l'expression de l'instruction, E. Notez que, dans le cas de cette analyse, c'est cette instruction qui est le point

de départ de la propagation de valeurs abstraites, non pas les paramètres comme dans les analyses précédentes.

Classes (C^{et}_{Class}). L'ensemble des contraintes produites par le générateur C^{et}_{Class} est l'ensemble des contraintes générées pour toutes les méthodes de la classe.

Programmes (C^{et}_{Prg}). Finalement, l'ensemble de contraintes $\mathcal{C}^{et}$ est l'ensemble des contraintes produites par le générateur C^{et}_{Prg}. Elle inclut les contraintes générées pour toutes les classes et celles de l'expression initiale.

6.3.2.3 Résolution des contraintes

Contrairement aux analyses étudiées dans les sections précédentes (voir section 6.1 et 6.2), la définition des contraintes de temps d'évaluation ne considère pas l'utilisation d'opérateurs conditionnels (voir section 6.1.2.1 et 6.2.2.1). En effet, toutes les contraintes dans l'ensemble $\mathcal{C}^{et}$, générées par le générateur C^{et}_{Prg}, sont de la forme $V^{et}_{c_a} \supseteq et$ et donc aucune transformation (simplification) n'est nécessaire avant la résolution.

6.3.2.4 Utilisation du solveur REQS

Tout comme dans l'analyse de type concret, les annotations de temps de liaison d'un programme peuvent aussi être calculées en utilisant le solveur REQS.

Treillis. Reprenons maintenant la définition du domaine ET (voir définition 6.5) des annotations de temps d'évaluation. Le domaine ET est l'ensemble de parties de l'ensemble π (voir définition 6.2) avec comme relation d'ordre partielle la relation d'inclusion. Pour cette analyse nous allons aussi utiliser le treillis ensemble de parties (*power set*) fourni par REQS

Règles de conversion vers REQS. Du fait de la forme des contraintes générées (voir section 6.3.2.1) la seule transformation nécessaire est celle qui permet de traduire les inégalités en égalités. Comme ces contraintes se basent sur la même relation que celles générées pour l'analyse de types concrets, la règle appliquée prend la même forme que la règle **I-TO-E**$^{\overline{reqs}}$ de la figure 6.11. En effet, la figure 6.34 présente la définition de la règle appliquée dans cette analyse pour la conversion d'un ensemble de contraintes qui présentent la même variable (représentant la même construction annotable) dans la partie droite de la contrainte comme une seule contrainte où la partie gauche est exprimée par l'union des expressions *et* des contraintes concernées (voir figure 6.10).

6.3.2.5 Temps d'évaluation sur le programme *Point2DStepping*

Dans cette section nous illustrons l'analyse de temps d'évaluation appliquée sur le programme *Point2DStepping* (voir section 6.1.2.5).

$$\frac{V_{c_a}^{ct} \supseteq et_1, \ldots, V_{c_a}^{et} \supseteq et_i, \ldots, V_{c_a}^{et} \supseteq et_n}{V_{c_a}^{et} = \texttt{(UNION } et_1 \ldots \texttt{(UNION } et_i \ldots \texttt{)} \ et_n \texttt{)} \ldots)} \quad [\textbf{I-TO-E}_{\widehat{req}}^{et}]$$

FIG. 6.34 – Temps d'évaluation : conversion d'inégalités en égalités.

Génération des contraintes. Comme mentionné lors de la description de cette analyse, la principal différence avec les analyse décrites auparavant, à savoir l'analyse de types concrets et de temps de liaison, est le sens de la propagation des annotations. Pour le calcul du temps d'évaluation, cette propagation commence à partir de l'annotation initiale donnée sur la valeur de retour des méthodes vers l'expression de l'instruction de retour correspondante.

Rappelons que le temps d'évaluation détermine quelles constructions *doivent* être spécialisées et/ou résidualisées, ce qui est différent du temps de liaison, qui détermine quelles construction peuvent entre spécialisées et quelles constructions ne le peuvent pas. De manière générale, il existe une relation d'ordre entre les annotations de temps de liaison et celles de temps d'évaluation. D'une part, si le temps de liaison d'une construction c_a indique qu'elle peut être spécialisée (*i.e.*, annotée comme $V_{c_a}^{bt} = \mathcal{S}$) alors soit elle peut être spécialisée (*i.e.*, $V_{c_a}^{et} = \{\mathcal{S}\}$), car il existe une valeur concrète associée, soit elle peut être résidualisée (*i.e.*, $V_{c_a}^{et} = \{\mathcal{D}\}$), car elle peut être rendue au monde dynamique à travers sa représentation textuelle. En conséquence, une telle construction pourrait être annotée aussi comme $V_{c_a}^{et} = \{\mathcal{S}, \mathcal{D}\}$ pour les arguments exposés ci-dessus. D'autre part, si l'annotation résultant de l'analyse de temps de liaison sur la construction c_a est dynamique (*i.e.*, $V_{c_a}^{bt} = \mathcal{D}$), alors la seule possibilité d'annotation de temps d'évaluation compatible avec une telle annotation sera $V_{c_a}^{et} = \{\mathcal{D}\}$. En effet, il est impossible d'évaluer une construction à la spécialisation dont la valeur est inconnue. Cette relation peut être formalisée comme suit :

$$\text{Si } \forall \ et \in V_{c_a}^{et} \mid et \sqsupseteq V_{c_a}^{bt}, \text{ alors } V_{c_a}^{et} \textit{ est conforme à } V_{c_a}^{bt} \tag{6.6}$$

Ce dernier raisonnement est clairement applicable sur les constructions de type primitif. Par contre, dans le cas des expressions de type objet, il faut prendre en compte que l'annotation de temps de liaison reste dans le domaine des types et non pas dans le domaine π des temps de liaison (voir définition 6.2). Cependant, comme mentionné dans l'introduction de la section 6.3, les expressions de type objet peuvent être spécialisées car l'objet correspondant peut toujours être créé à la spécialisation même s'il existe des champs dynamiques. L'expression sera complètement résidualisée si tous les champs sont annotés dynamiques.

Ainsi, lors d'un accès à un champ ou un appel de méthode, la sous-expression receveur une expression de type objet dont l'annotation de temps d'évaluation est la cible de la propagation du temps de liaison de l'expression englobante (acceès à un champ ou appel de méthode). Par exemple, si on reprend l'expression initiale e_{main} (voir section 6.1.2.5), les expressions représentant le receveur des expressions de l'accès aux champs `x` et `y` sont annotées selon les annotations d'accès, $V_{\_,\_,e_1}^{et}$ et $V_{\_,\_,e_2}^{et}$ respectivement. Dans ce contexte,

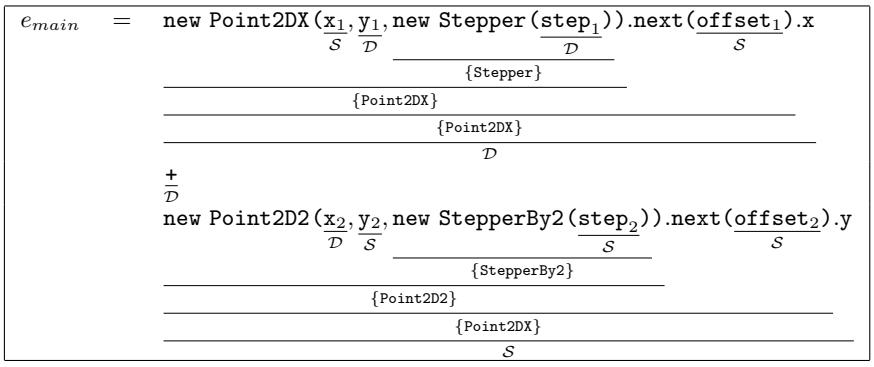


FIG. 6.35 – Temps de liaison : annotation de l'expression initiale e_{main}.

la propagation peut être déclenchée à partir de valeurs de temps de liaison calculées. Sinon, il faudrait indiquer le temps d'évaluation souhaité, toujours en respectant la relation exprimée par la définition 6.6.

Cependant, à la différence de l'exemple étudié tout au long de ce chapitre, on peut considérer une expression initiale de type objet. Dans ce cas, il se peut qu'il s'agisse d'une expression où toutes les sous-expressions sont *spécialisables* (*i.e.*, annotées comme $\{\mathcal{S}\}$) et donc aucun résidualisation n'a lieu. Autrement dit, l'expression est exécutée complètement à la spécialisation et donc le résultat de la spécialisation est vide. Pour cette raison, l'utilisateur peut forcer la résidualisation d'une expression. Cette possibilité est applicable aussi pour les expressions de type primitif. Maintenant, afin d'illustrer comment les annotations sont propagées, nous reprenons l'expression initiale E_{main} donnée dans la section 6.1.2.5. La figure 6.35 montre le résultat de l'analyse de temps de liaison calculé à partir des valeurs initiales données par l'environnent Γ_0^{bt}. Notez que cette expression a été annotée comme dynamique à la suite de l'analyse de temps de liaison. Pour cette raison, comme elle ne peut pas être spécialisée, la seule annotation initiale de temps d'évaluation conforme à la définition 6.6 est $\{\mathcal{D}\}$.

Le table 6.4 montre les contraintes générées par le générateur C_{Exp}^{et} en prenant comme argument l'expression initiale. La contrainte (a) représente le fait que le temps d'évaluation de l'expression initiale dépend de son temps de liaison.

En ce qui concerne la sous-expression d'accès au champ `x` le temps d'évaluation représenté par la variable de contrainte $V_{\_.\_.E_1}^{et}$ se propage vers la sous-expression représentant le receveur E_{10} (contrainte (b)). En effet, la sous-expression E_{10} est elle-même l'expression d'appel de la méthode `next` et donc le temps d'évaluation associé est propagé vers les sous-expressions du receveur E_{100} (contrainte (c)), ainsi que vers l'expression de retour de la méthode `next` de la classe `Point2DX` (contrainte (d)). Notons que dans le cas de l'expression de retour des méthodes, les méthodes concernées sont celles définies dans les classes associées à l'expression du receveur qui ont été calculées lors de l'analyse de type concret. Finalement, dans le cas de la sous-expression E_{100}, une expression d'instanciation, la contrainte (e) exprime la propagation des annotations des paramètres de type objet du constructeur correspondant.

Générateur	Éléments en $\mathcal{C}^{et}$	
$C^{et}_{Exp}(\_,\_,E_{main})$		
$E_{main} = E_1\ op_b\ E_2$	$V^{et}_{\_.\_.E_{main}} = t^{bt}_{\_.\_.E_{main}}$	(a)
$E_1 = E_{10}.\texttt{x}$	$V^{et}_{\_.\_.E_{10}} \supseteq V^{et}_{\_.\_.E_1}$,	(b)
	$V^{et}_{\texttt{Point2DX.x}} \supseteq V^{et}_{\_.\_.E_1}$	
$E_{10} = E_{100}.\texttt{next}(E_{101})$	$V^{et}_{\_.\_.E_{100}} \supseteq V^{et}_{\_.\_.E_{10}}$,	(c)
	$V^{et}_{\texttt{Point2DX.next.return}} \supseteq V^{et}_{\_.\_.E_{10}}$	(d)
$E_{100} = \texttt{new Point2DX}(E_{1001}, E_{1002}, E_{1003})$	$V^{et}_{\_.\_.E_{1003}} \supseteq V^{et}_{\_.\_.E_{100}}$	(e)
$E_{1001} = \texttt{x}_1$	$V^{et}_{\_.\_.E_{1001}} = t^{bt}_{\_.\_.E_{1001}}$	
$E_{1002} = \texttt{y}_1$	$V^{et}_{\_.\_.E_{1002}} = t^{bt}_{\_.\_.E_{1002}}$	
$E_{1003} = \texttt{new Stepper}(E_{10031})$	$\emptyset$	
$E_{10031} = \texttt{step}_1$	$V^{et}_{\_.\_.E_{10031}} = t^{bt}_{\_.\_.E_{10031}}$	
$E_{101} = \texttt{offset}_1$	$V^{et}_{\_.\_.E_{101}} = t^{bt}_{\_.\_.E_{101}}$	
$E_2 = E_{20}.\texttt{y}$	$V^{et}_{\_.\_.E_{20}} \supseteq V^{et}_{\_.\_.E_2}$,	(f)
	$V^{et}_{\texttt{Point2DX.y}} \supseteq V^{et}_{\_.\_.E_2}$	
$E_{20} = E_{200}.\texttt{next}(E_{201})$	$V^{et}_{\_.\_.E_{200}} \supseteq V^{et}_{\_.\_.E_{20}}$,	(g)
	$V^{et}_{\texttt{Point2D2.next.return}} \supseteq V^{et}_{\_.\_.E_{20}}$	(h)
$E_{200} = \texttt{new Point2D2}(E_{2001}, E_{2002}, E_{2003})$	$V^{et}_{\_.\_.E_{2003}} \supseteq V^{et}_{\_.\_.E_{200}}$	(i)
$E_{2001} = \texttt{x}_2$	$V^{et}_{\_.\_.E_{2001}} = t^{bt}_{\_.\_.E_{2001}}$	
$E_{2002} = \texttt{y}_2$	$V^{et}_{\_.\_.E_{2002}} = t^{bt}_{\_.\_.E_{2002}}$	
$E_{2003} = \texttt{new Stepper}(E_{20031})$	$\emptyset$	
$E_{20031} = \texttt{step}_2$	$V^{et}_{\_.\_.E_{20031}} = t^{bt}_{\_.\_.E_{20031}}$	
$E_{201} = \texttt{offset}_2$	$V^{et}_{\_.\_.E_{201}} = t^{bt}_{\_.\_.E_{201}}$	

TAB. 6.4 – Temps d'évaluation : génération des contraintes à partir de l'expression initiale (E_{main}).

Une situation similaire se présente dans le cas de l'expression E_2 où le temps d'évaluation est propagé vers l'expression de retour de la méthode `next` de la classe `Point2D2` comme exprimé par les contraintes (f), (g), (h) et (i).

Dans les cas restants, les annotations de temps d'évaluation dépendent des annotations de temps de liaison.

Maintenant, voyons comment les contraintes générées à partir de l'expression initiale affectent les annotations du programme. Pour simplifier l'explication, nous n'étudions que les contraintes de temps d'évaluation générées à partir de la classe `Point2DX` (voir figure 6.5). Dans la contrainte (a) intervient l'annotation obtenue de l'expression initiale par la contrainte (d) de la figure 6.4 en affectant l'annotation de l'expression de l'instruction de retour `Point2DX.next`.E. La propagation faite vers le corps de la méthode a son point de départ dans la contrainte (b) (voir figure figure 6.5).

L'annotation de temps d'évaluation de la méthode `dec`, quant à elle, est calculée à partir de la contrainte (c) générée dans l'analyse de la méthode `next` de la classe `Point2DX`. Dans ce cas, par contre, comme c'est une expression de type primitif, l'annotation de temps d'évaluation dépend du temps de liaison. Notons que la contrainte (d) intervient aussi sur

Générateur	Éléments en $\mathcal{C}^{et}$	
C^{et}_{Class}(`Point2DX`)		
C^{et}_{Mthd}(`Point2DX`, `int next` {`return` E})	$V^{et}_{\texttt{Point2DX.next.}E} \supseteq V^{et}_{\texttt{Point2DX.next.return}}$	(a)
C^{et}_{Exp}(`Point2DX`, `next`, E)		
$E =$ `new Point2DX(`E_1, E_2, E_3`)`	$V^{et}_{\texttt{Point2DX.next.}E_3} \supseteq V^{et}_{\texttt{Point2DX.next.}E}$	(b)
$E_1 = E_{11}$`+`E_{12}	$V^{et}_{\texttt{Point2DX.next.}E_1} = t^{bt}_{\texttt{Point2DX.next.}E_1}$	
$E_{11} = E_{110}$`.x`	$V^{et}_{\texttt{Point2DX.next.}E_{11}} = t^{bt}_{\texttt{Point2DX.next.}E_{11}}$,	
	$V^{et}_{\texttt{Point2DX.next.}E_{110}} \supseteq V^{et}_{\texttt{Point2DX.next.}E_{11}}$	
$E_{110} =$ `this`	$V^{et}_{\texttt{Point2DX.next.}E_{110}} = V^{et}_{\texttt{Point2DX.next.}E_{110}}$	
$E_{12} = E_{120}$`.dec(`E_{121}`)`	$V^{et}_{\texttt{Point2DX.next.}E_{12}} = t^{bt}_{\texttt{Point2DX.next.}E_{12}}$,	
	$V^{et}_{\texttt{Point2DX.next.}E_{120}} \supseteq V^{et}_{\texttt{Point2DX.next.}E_{12}}$,	
	$V^{et}_{\texttt{Point2DX.dec.return}} = t^{bt}_{\texttt{Point2DX.next.}E_{12}}$	(c)
$E_{120} =$ `this`	$V^{et}_{\texttt{Point2DX.next.}E_{120}} = V^{et}_{\texttt{Point2DX.next.}E_{120}}$	
$E_{121} =$ `offset`	$V^{et}_{\texttt{Point2DX.next.}E_{121}} = t^{bt}_{\texttt{Point2DX.next.}E_{121}}$	
$E_2 = E_{21}$`+`E_{22}	$V^{et}_{\texttt{Point2DX.next.}E_2} = t^{bt}_{\texttt{Point2DX.next.}E_2}$	
$E_{21} = E_{210}$`.y`	$V^{et}_{\texttt{Point2DX.next.}E_{21}} = t^{bt}_{\texttt{Point2DX.next.}E_{21}}$,	
	$V^{et}_{\texttt{Point2DX.next.}E_{210}} \supseteq V^{et}_{\texttt{Point2DX.next.}E_{21}}$	
$E_{210} =$ `this`	$V^{et}_{\texttt{Point2DX.next.}E_{210}} = V^{et}_{\texttt{Point2DX.next.}E_{210}}$	
$E_{22} = E_{220}$`.dec(`E_{221}`)`	$V^{et}_{\texttt{Point2DX.next.}E_{22}} = t^{bt}_{\texttt{Point2DX.next.}E_{22}}$,	
	$V^{et}_{\texttt{Point2DX.next.}E_{220}} \supseteq V^{et}_{\texttt{Point2DX.next.}E_{22}}$,	
	$V^{et}_{\texttt{Point2DX.dec.return}} \supseteq V^{et}_{\texttt{Point2DX.dec.}E_{22}}$	(d)
$E_{220} =$ `this`	$V^{et}_{\texttt{Point2DX.next.}E_{220}} = V^{et}_{\texttt{Point2DX.next.}E_{220}}$	
$E_{221} =$ `offset`	$V^{et}_{\texttt{Point2DX.next.}E_{221}} = t^{bt}_{\texttt{Point2DX.next.}E_{221}}$	
$E_3 = E_{30}$`aStepper`	$V^{et}_{\texttt{Point2DX.next.}E_{30}} \supseteq V^{et}_{\texttt{Point2DX.next.}E_3}$	
$E_{30} =$ `this`	$V^{et}_{\texttt{Point2DX.next.}E_{30}} = V^{et}_{\texttt{Point2DX.next.}E_{30}}$	
C^{et}_{Mthd}(`Point2DX`, `intdec`{`return` E})	$V^{et}_{\texttt{Point2DX.dec.}E} \supseteq V^{et}_{\texttt{Point2DX.dec.return}}$	(e)
C^{et}_{Exp}(`Point2DX`, `dec`, E)		
$E = E_1$`+`E_2	$V^{et}_{\texttt{Point2DX.dec.}E} = t^{bt}_{\texttt{Point2DX.dec.}E}$	
$E_1 = E_{10}$`.step()`	$V^{et}_{\texttt{Point2DX.dec.}E_1} = t^{bt}_{\texttt{Point2DX.dec.}E_1}$,	
	$V^{et}_{\texttt{Point2DX.dec.}E_{10}} \supseteq V^{et}_{\texttt{Point2DX.dec.}E_1}$	
$E_{10} = E_{100}$`.aStepper`	$V^{et}_{\texttt{Point2DX.dec.}E_{100}} \supseteq V^{et}_{\texttt{Point2DX.dec.}E_{10}}$	
$E_{100} =$ `this`	$V^{et}_{\texttt{Point2DX.dec.}E_{100}} = V^{et}_{\texttt{Point2DX.dec.}E_{100}}$	
$E_2 =$ `offset`	$V^{et}_{\texttt{Point2DX.dec.}E_2} = t^{bt}_{\texttt{Point2DX.dec.}E_2}$	

TAB. 6.5 – Temps d'évaluation : génération des contraintes à partir de la classe `Point2DX`.

l'annotation de la valeur de retour de la même méthode. Une fois appliquée la règle de conversion de la figure 6.34 l'annotation est propagée vers le corps de la méthode par la contrainte (e).

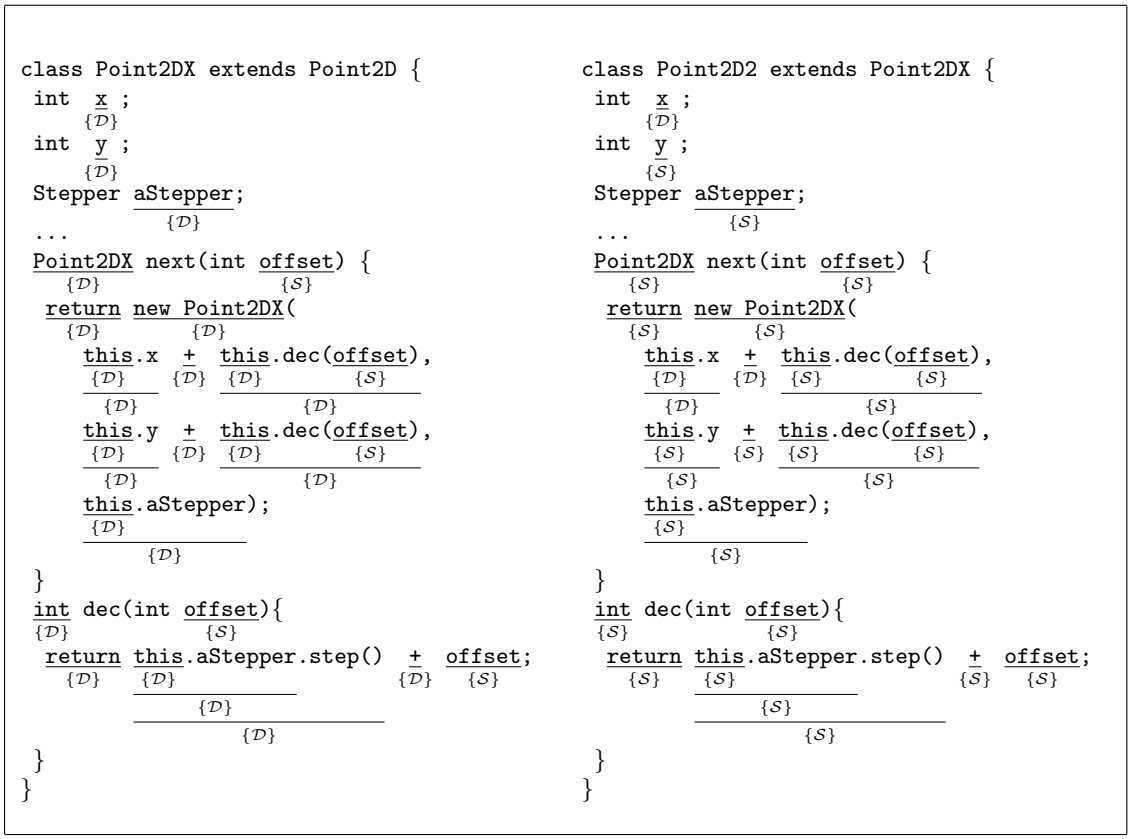


FIG. 6.36 – Temps d'évaluation : annotations des classes `Point2DX` et `Point2D2`.

6.4 Spécialisation

Cette section est consacrée à la génération de code spécialisé à l'aide des extensions génératrices ainsi que à la spécification d'un générateur d'extensions génératrices. Comme mentionné dans le section 2.1.3, il existe deux façon de construire un générateur d'extensions génératrices : l'une indirecte, par auto-application (voir section 2.1.3.1), et l'autre directe, par l'écriture d'un générateur d'extensions génératrices (voir section 2.1.3.2). Cette dernière méthode est présentée dans cette section.

6.4.1 Les extensions génératrices

Sur la base des annotations de temps d'évaluation calculées dans la section précédente, nous abordons l'étude du processus de spécialisation de programmes. Comme mentionnée

dans la section 5.2.2, la notre est une approche génératrice, c'est-à-dire qu'on génère un générateur de programmes spécialisés (voir section 2.1.3) au lieu de produire directement le programme spécialisé selon les valeurs concrètes données (voir figure 2.3). Les deux approches ont comme point en commun l'utilisation des programmes annotés avec le temps de liaison pour effectuer la spécialisation correspondante. Dans ce contexte, les programmes peuvent se voir comme un ensemble de constructions annotables (voir section 5.3.3) avec leur annotation respective tel que :

$$et = \Gamma^{et}(c_a) \mid c_a \in C_A,\ et \in ET$$

Avant de spécifier le générateur d'extensions génératrices (voir section 6.4.1.1), nous abordons premièrement la différence entre la spécialisation à l'aide d'un évaluateur partiel hors ligne et l'approche génératrice. Prenons l'expression annotée binaire E_1 dont le temps d'évaluation est le suivant :

$$E_1 \ := \ \underset{\{\mathcal{D}\}}{\underset{\{\mathcal{S}\}}{\mathtt{x}} \ \mathtt{+} \ \underset{\{\mathcal{D}\}}{\mathtt{y}}}$$

Évaluateur partiel hors ligne. La spécialisation de l'expression E_1, à l'aide d'un évaluateur partial, donne comme résultat une expression qui représente, dans ce cas, l'addition d'une constante, la valeur concrète de la variable `x`, à la variable `y`. En reprenant la définition de l'évaluateur hors ligne de la section 2.1.2, il est possible d'exprimer la spécialisation de l'expression annotée E_1 comme suit :

$$[\![evalpar_{Exp}]\!](E_1, \Gamma_0^{cv}) = E_1' \tag{6.7}$$

L'environnement Γ_0^{cv} lie les variables statiques aux valeurs concrètes. Dans ce cas nous considérons, par exemple, $\mathtt{x=2} \in \Gamma_0^{cv}$. L'expression spécialisée E_1' est le résultat de l'évaluation de la partie statique et la résidualisation de la partie dynamique de l'expression originale. De même que les analyses décrites dans les sections précédentes, la spécialisation est effectuée de manière compositionnelle selon la structure de l'expression. Par exemple, la spécialisation de l'expression E_1 est le résultat de la spécialisation des sous-expressions correspondantes. En conséquence, la définition 6.7 peut être récrite sous la forme suivante :

$$[\![evalpar_{Exp}]\!](E_1, \Gamma_0^{cv}) = build_{Exp}([\![evalpar_{Exp}]\!](E_{11}, \Gamma_0^{cv}), [\![evalpar_{Exp}]\!](E_{12}, \Gamma_0^{cv})) \tag{6.8}$$

Dans la définition 6.8, les parties statiques et les parties dynamiques des sous-expressions sont reconstruites par le constructeur $build_{Exp}$. Le constructeur à appliquer dépendra du type de l'expression englobante (par exemple, $build_{op_b}$ pour les opérations binaires).

La sous-expression E_{11} étant une variable statique, sa spécialisation ne retourne que la valeur de la variable liée par l'environnement Γ_0^{cv} :

$$[\![evalpar_{Exp}]\!](E_{11}, \Gamma_0^{cv}) = (\Gamma_0^{cv}(\mathtt{x}), \bot) = (\mathtt{2}, \bot)$$

La sous-expression E_{12} étant une expression dynamique, sa spécialisation ne conduit à aucune évaluation, seulement à la résidualisation de la variable `y` :

$$[\![evalpar_{Exp}]\!](E_{12}, \Gamma_0^{cv}) = (\bot, \mathtt{y})$$

Finalement, la spécialisation résultant de l'application de l'évaluateur partiel $evalpar_{Exp}$ implique la résidualisation de l'expression statique `2` à cause du contexte dynamique influencé par la sous-expression E_{12}.

$$[\![evalpar_{Exp}]\!](E_1, \Gamma_0^{cv}) = (\bot, \mathtt{2} + \mathtt{y})$$

Approche génératrice. Dans l'approche génératrice, la spécialisation est produite lorsque une extension génératrice est exécutée. En général, une extension génératrice peut se voir comme une fonction avec comme arguments les variables statiques de la construction à spécialiser. Un fois les valeurs concrètes affectées aux arguments, l'extension génératrice génère une version spécialisée.

Rappelons que dans le contexte d'une approche génératrice la spécialisation est réalisée en deux étapes : en premier lieu, on génère le code de l'extension génératrice (voir définition 2.9), laquelle est exécuté, en second lieu, afin de générer la version spécialisée selon les valeur concrètes (voir définition 2.10). Lors de la première étape, la partie statique est résidualisée en un code source à évaluer lors de l'exécution de l'extension génératrice tandis que la partie dynamique est résidualisée en un code source à résidualiser lors de l'exécution de l'extension génératrice. Comme la partie à évaluer peut inclure des variables statiques, la liste des variables statiques doit être incluse dans l'information retournée par le générateur d'extensions génératrices. Cette liste sert ultérieurement à l'implémentation de la fonction qui représente l'extension génératrice. Dans le cas des expressions composées, l'ensemble des variables statiques correspond à l'union des variables statiques des sous-expressions. Lors de la deuxième étape, afin de générer la spécialisation correspondante, les constructions statiques sont évaluées et les constructions dynamiques sont résidualisées.

En reprenant la définition de $cogen_{Exp}$ (voir définition 2.9), l'extension génératrice de l'expression E_1 est exprimée comme suit :

$$[\![cogen_{Exp}]\!](E_1) = extGen_{E_1}$$

$$extGen_{E_1} = build_{Exp}([\![cogen_{Exp}]\!](E_{11}), [\![cogen_{Exp}]\!](E_{12})) \tag{6.9}$$

tel que

$$[\![cogen_{Exp}]\!](E_{11}) = (\mathtt{x}, \bot, \{\mathtt{x}\}) \text{ et } [\![cogen_{Exp}]\!](E_{12}) = (\bot, \mathtt{y}, \{\})$$

L'application de $cogen_{Exp}$ à l'argument E_{11} donne comme résultat simplement un triplet dont la partie statique est la construction elle-même, la partie dynamique est vide et la liste de variables statiques est l'ensemble singleton $\{\mathtt{x}\}$. Pour la sous-expression E_{12} le triplet résultant est formé d'une partie statique vide, de la variable `y` comme partie dynamique du triplet et une liste de variables statiques vide. Ici, de même que dans

la spécialisation effectuée en appliquant un évaluateur partiel, on utilise le constructeur $buildGen_{Exp}$ pour générer le triplet correspondant (voir définition 6.10).

$$[\![cogen_{Exp}]\!](E_1) = (\bot, \texttt{x+y}, \{\texttt{x}\}) \tag{6.10}$$

Notons que, dans la définition 6.10, la partie statique doit être résidualisées car il s'agit d'un contexte de spécialisation dynamique.

6.4.1.1 Le générateur d'extensions génératrices

Nous présentons maintenant la structure du générateur d'extensions génératrices pour les expressions binaires $cogen_{OB}$. Le générateur $cogen_{OB}$ se base sur l'application des générateurs correspondant aux sous-expressions :

Soit

$$E ::= E_1 \ op_b \ E_2$$

tel que

$$cogen_{Exp}(E_1) = (\mathcal{S}_{E_1}, \mathcal{D}_{E_1}, v_{E_1}),$$
$$cogen_{Exp}(E_2) = (\mathcal{S}_{E_2}, \mathcal{D}_{E_2}, v_{E_2}) \text{ et}$$
$$v_E = v_{E_1} \cup v_{E_2}$$

alors

$$cogen_{op_b}(E) = build_{Exp}(cogen_{Exp}(E_1), cogen_{Exp}(E_2))$$

Le générateur $cogen_{Exp}$ (voir définitions 6.4.1.1 et 6.4.1.1) représente un générateur d'extension génératrices générique dont la définition dépend de l'expression à spécialiser. Par application du générateur $cogen_{Exp}$ aux sous-expressions on obtient les triplets résultats où $\mathcal{S}_{E_i}$ représente la partie statique, $\mathcal{D}_{E_i}$ représente la partie dynamique et v_{E_i} la liste des variables statiques de l'expression annotée E_i.

$$cogen_{op_b}(E) = \begin{cases} (E, \bot, v_E) & \text{si } \Gamma^{et}(E) = \{\mathcal{S}\} \\ (\bot, build_{op_b}(build_{lift}(\mathcal{S}_{E_1}), \mathcal{D}_{E_2}), v_E) & \text{si } \Gamma^{et}(E_1) = \{\mathcal{S}\} \\ (\bot, build_{op_b}(\mathcal{D}_{E_1}, build_{lift}(\mathcal{S}_{E_2})), v_E) & \text{si } \Gamma^{et}(E_2) = \{\mathcal{S}\} \\ (\bot, build_{op_b}(\mathcal{D}_{E_1}, \mathcal{D}_{E_2}), v_E) & \text{sinon} \end{cases} \tag{6.11}$$

La définition 6.11 présente le générateur $cogen_{op_b}$, dont la structure est basée sur toutes les combinaisons possibles des annotations de temps d'évaluation des sous-expressions : le cas où les sous-expressions sont statiques (*i.e.*, $\Gamma^{et}(E) = \{\mathcal{S}\}$), les cas où une des sous-expressions est statique (*i.e.*, $\Gamma^{et}(E_1) = \{\mathcal{S}\}$ ou $\Gamma^{et}(E_2) = \{\mathcal{S}\}$) et finalement le cas où les deux sous-expressions sont dynamiques.

En premier lieu, l'application du générateur $cogen_{Exp}$ à une expression annotée $\{\mathcal{S}\}$ résulte en un triplet où l'expression à évaluer est l'expression elle-même alors que la partie résiduelle est vide.

En second lieu, dans les cas où l'expression est annotée $\{\mathcal{D}\}$, une des sous-expressions peut être annotée $\{\mathcal{S}\}$ (voir deuxième et troisième ligne de la définition 6.11). Cela implique le traitement d'une expression statique dans un contexte dynamique et par conséquent, cette expression doit être *liftée*. Rappelons que la fonction *lift* (voir section 2.1.2.1) appliquée à une expression statique implique, d'abord, l'évaluation de l'expression et, ensuite, la résidualisation de la valeur calculée. Autrement dit, tandis que la fonction *lift* est exécutée lors de l'exécution de l'extension génératrice, son argument est évalué lors de l'exécution du générateur. Pour cette raison, le fait que l'expression doive être liftée implique l'utilisation du constructeur $build_{lift}$ qui permet de générer l'appel à la fonction *lift* pour générer la résidualisation correspondante.

En dernier lieu, si les deux sous-expressions sont dynamiques alors le constructeur génère la partie à résidualiser en prenant les parties dynamiques des générateurs des sous-expressions.

Dans tous les cas, la liste de variables statiques est formée par l'union des variables statiques calculées par les générateurs des sous-expressions concernées.

Afin de donner une idée complète des relations existante entre les générateur des expressions et des sous-expressions, nous fournissons la définition du générateur associé aux variables $cogen_{var}$ qui est défini comme suit :

$$cogen_{var}(E) = \begin{cases} (E, \bot, \{E\}) & \text{, si } \Gamma^{et}(E) = \{\mathcal{S}\} \\ (\bot, build_{var}(E), \{\}) & \text{, si } \Gamma^{et}(E) = \{\mathcal{D}\} \end{cases} \tag{6.12}$$

6.4.1.2 Résidualisation

Le processus de spécialisation au moyen des extensions génératrices implique la résidualisation à deux étapes : une première étape concernant la partie statique, résidualisée par le générateur, et une deuxième étape concernant la partie dynamique, résidualisée par l'extension génératrice.

Dans cette section, nous étudions l'implémentation et l'application des générateurs et les générations d'extensions génératrices afin d'illustrer la résidualisation qui a lieu tout au long de la spécialisation. En termes d'implémentation, dans le contexte de cette thèse, la résidualisation implique concrètement la génération de code source, dans le langage d'implémentation des programmes analysés, sous la forme d'une chaîne de caractères.

Implémentation des générateurs d'extensions génératrices. Pendant la première étape, sur la base des triplets (voir définition 6.11), le générateur résidualise la partie statique pour générer le code à évaluer par l'extension génératrice. La partie dynamique, quant à elle, est aussi résidualisée mais, dans ce cas, afin d'être résidualisée ultérieurement par l'extension génératrice.

La figure 6.37 montre le pseudo-code des générateurs d'extension génératrices, appliqués aux constantes, les variables et les opérations binaires, ainsi que les constructeurs

```
cogenExp(E) = case e of
  c → cogenC(E);
  x → cogenVar(E);
  e1 OPB e2 → cogenOPB(E);

cogenC(E) =
  if (envET(E) = "S") → (E,"",{});
  if (envET(E) = "D") → error;

cogenVar(E) =
  if (envET(E) = "S") → (e,"",{E});
  if (envET(E) = "D") → (null,quote(toStringExp(E)),{});

cogenOPB(E) =                    // E := [E1,opb,E2]
  cogenExp1 := cogenExp(E.E1); // [S,D,v]
  cogenExp2 := cogenExp(E.E2); // [S,D,v]
  v := cogenExp1.v ∪ cogenExp2.v;
  if (envET(E) = "S")
     → (E,"",v);

  if (envET(E) = "S" and envET(E.E2) = "D")
     → (null,buildOPB(buildLift(cogenExp1.S),toStrOPB(E.opb),cogenExp2.D),v);

  if (envET(E.E1) = "D" and envET(E.E2) = "S")
     → (null,buildOPB(cogenExp1.D,toStrOPB(E.opb),buildLift(cogenExp2.S)),v);

  if (envET(E.E1) = "D" and envET(E.E2) = "D")
     → (null,buildOPB(cogenExp1.D,toStrOPB(E.opb),cogenExp2.D),v);

buildOPB(s1,sopb,s2)= → quote(s1^" "^quote("")^quote(sopb)^quote("")^" "^s2);

buildLift(E) = → "lift("^toStrExp(E)^")";

quote(s) = "\""^s^"\""

toStrExp(E) = case E of
   c → "c";
   x → "x";
   E1 OPB E2 → toStrExp(E1)^" "^toStrOPB(OPB)^" "^toStrExp(E2);

toStrOPB(opb) = case opb of
   + → "+";
   - → "-";
   / → "/";
   * → "*";
```

FIG. 6.37 – Générateur d'extensions génératrices pour les opérations binaires.

correspondants. Le générateur générique `cogenExp` appelle un générateur spécifique selon l'expression `E`, à savoir le générateur `cogenC` pour les constantes, le générateur `cogenVar` pour les variables et le générateur `cogenOPB` pour les expressions binaires. La fonction `envET` permet d'accéder à l'annotation de temps d'évaluation de l'expression. Les fonctions `toStringExp` et `toStringOPB`, appliquées aux expressions et opérateurs binaires, retournent la chaîne de caractères qui représente l'expression `E`. Dans le pseudo-code, les paramètres représentant les expressions sont préfixés par une lettre `E` tandis que ceux représentant les chaînes de caractères sont préfixés par une lettre `s`. La forme de la valeur de retour des générateurs est un triplet (*i.e.*, `(_,_,_)`) tandis que pour les autres fonctions c'est simplement une chaîne de caractères.

Les constantes sont traitées par le générateur `cogenC`. Il retourne l'expression qui représente la constante dans le cas d'une annotation statique et une erreur (`error`) si la constante est annotée dynamique.

En ce qui concerne les variables, le générateur utilisé est `cogenVar`. La chaîne de caractères représentant le code dans le langage de programmation est calculée par la fonction `toStringExp` selon la classe de l'expression. La fonction `quote` ajoute une couche de guillemets aux chaînes de caractères qui représentent la partie dynamique résultante. Cela permet à l'extension génératrice, comme nous le verrons ci-après, de résidualiser le code spécialisé.

Le générateur qui permet de produire l'extension génératrice d'une opération binaire est la fonction `cogenOPB`. Le résultat de l'application des générateurs correspondant aux sous-expressions est affecté à une variable dont les valeurs, correspondant, au triplet sont accédées à l'aide d'une notation pointée (par exemple, `cogenExp1.S` représente la partie statique du triplet `cogenExp1`). Ici, nous focaliserons l'explication sur les cas où une seule des expressions est annotée dynamique. Dans le deux cas possibles, la construction de la partie dynamique est faite à l'aide du constructeur `buildOPB` qui prend comme paramètres les chaînes de caractères représentant chacune des sous-expressions ainsi que celle de l'opérateur. Afin de reconstruire l'expression qui représente l'application de la fonction `lift`, on utilise le constructeur `buildLift` qui génère la chaîne de caractères correspondante à partir de l'expression statique passée comme paramètre. Rappelons que la fonction `lift`, comme nnous le verrons dans le paragraphe suivant, est exécutée lors de l'exécution de l'extension génératrice. Afin d'illustrer la manipulation des chaînes de caractères appliquons le générateur `cogenOPB` à l'expression e_1 définie et annotée auparavant.

Les triples retournés par le générateur `cogenVar`, suite à l'application aux sous-expressions `x` et `y`, sont les suivants :

```
cogenVar(x) = (x,"",{x})
cogenVar(y) = (null, ""y"",{})
```

Ainsi, le résultat final de l'application du générateur `cogenOPB` est un triplet dont la partie dynamique est calculée à l'aide du constructeur `buildOPB` comme suit :

```
buildOPB(buildLift(x),"+","y") = "lift(x)" ^ "^" ^ "+" ^ "^" ^ "y""
```

tel que

```
cogenOPB(E1) = (null,"lift(x)" ^ "^" ^ "+" ^ "^" ^ "y"",{x});
```

La fonction `quote` permet d'encapsuler dans la partie dynamique le code dynamique, résidualisée telle quelle, plus le code à lifter, évalué lors de l'exécution de l'extension génératrice.

Implémentation des extensions génératrices Lors de la deuxième étape, l'extension génératrice résidualise la partie dynamique à l'aide de l'évaluation du code produit dans la première étape. Concrètement, une extension génératrice est implémentée sous la forme d'une fonction prenant comme paramètres les variables statiques. Comme mentionné auparavant, l'exécution de cette fonction implique l'évaluation de la partie statique et la résidualisation de la partie dynamique où les fragments liftés sont aussi évalués. Par exemple, par application du générateur sur l'expression `E1`, on obtient la fonction `genExte1` définie comme suit :

```
genExte1(x) = (null, print("lift(x)" ^ "^" ^ "+" ^ "^" ^ "y""));
```

L'exécution de l'extension génératrice implique l'évaluation du code statique représenté par une expression et l'évaluation du résultat de l'impression de la chaîne de caractères qui représente la partie dynamique. Dans le dernier cas, l'expression à évaluer est :

```
lift(x) ^ "+" ^ "y"
```

où la fonction `lift` prend la valeur du paramètre `x` pour calculer finalement la chaîne de caractères correspondente. Notons que la fonction `lift` peut être implémentée simplement par l'appel de la fonction, dans le langage d'implémentation des extensions génératrices, qui permet traduire une valeur de type primitif vers une chaîne de caractères.

Application des extensions génératrices. Finalement, par application de l'extension génératrice on obtient le code spécialisé exprimé sous la forme d'une fonction où les paramètres représentent les variables dynamiques. Dans l'exemple de l'expression e_2, en affectant la valeur `2` à la variable `x`, le code résiduel est le suivant :

```
genExtOPBe1_x2(y) = → (2 + y);
```

6.4.2 Générateur d'extensions génératrices pour EFJ

Sur la base de l'explication présentée ci-dessus sur la définition d'un générateur d'extensions génératrices des expressions simples (par exemple, *cogenOPB* pour les opérations binaires), il est possible d'étendre cette idée pour un langage comme EFJ. La figure 6.38 présente le générateur d'extensions génératrices $cogen_{Prg}$ appliqué aux programmes EFJ dont les constructions sont annotés avec le temps d'évaluation stockés dans l'environnement Γ^{et}.

En suivant la structure des programmes EFJ, le générateur $cogen_{Prg}$ appelle les sous-générateurs associés aux constructions correspondantes. Pour illustrer un des générateurs,

$$
\begin{array}{lcl}
cogen_{Exp}(E) & = & \text{case } E \text{ of} \\
k & \Rightarrow & cogen_{k}(E) \\
\texttt{this} & \Rightarrow & cogen_{This}(E) \\
x & \Rightarrow & cogen_{Var}(E) \\
E_0.x & \Rightarrow & cogen_{Field}(E) \\
\texttt{new } c\texttt{(}\overline{E}\texttt{)} & \Rightarrow & cogen_{New}(E) \\
\texttt{(}c\texttt{)}E_1 & \Rightarrow & cogen_{Cast}(E) \\
E_1\ op_b\ E_2 & \Rightarrow & cogen_{op_b}(E) \\
op_u\ E_1 & \Rightarrow & cogen_{op_u}(E) \\
E_1\texttt{?}E_2\texttt{:}E_3 & \Rightarrow & cogen_{Cond}(E) \\
E_0.mt\texttt{(}d_1, \ldots, d_n\texttt{)} & \Rightarrow & cogen_{Invk}(E) \\
 & & \\
cogen_{Stmt}(S) & = & \text{case } S \text{ of} \\
\texttt{return } E & \Rightarrow & \text{do } cogen_{Exp}(E) \\
 & = & \text{case } \Gamma^{et}(E) \text{ of} \\
 & & \{\mathcal{S}\} \Rightarrow (build_{Return_{\mathcal{S}}}(\mathcal{S}_E), \bot, v_E) \\
 & & \{\mathcal{D}\} \Rightarrow (\bot, build_{Return_{\mathcal{D}}}(\mathcal{D}_E), v_E) \\
 & & \{\mathcal{S}, \mathcal{D}\} \Rightarrow (build_{Return_{\mathcal{S}}}(\mathcal{S}_E), build_{Return_{\mathcal{D}}}(\mathcal{D}_E), v_E) \\
 & & \\
cogen_{Mthd}(C, \_\ m\texttt{(}\overline{X\texttt{,}}\texttt{)\{}S\texttt{;\}}) & = & cogen_{Stmt}(S) \\
 & & \bigcup cogen_{Sign}(C, M) \bigcup cogen_{Body}(C, M) \\
 & & \\
cogen_{Const}(C, K) & = & (\bot, build_{Const}(K_C), v_{K_C}) \\
 & & \\
cogen_{Class}(\texttt{class } c \ldots \texttt{\{}\_\ K\ \overline{M}\texttt{\}}) & = & cogen_{Const}(C, K),\ constructor(C) = K_C \\
 & & \bigcup_{M\ \in\ methods(C)} cogen_{Mthd}(C, M) \\
 & & \\
cogen_{Prg}(\{C_{1_{ann}}, \ldots, C_{n_{ann}}, E_{main_{ann}}\}) & = & \bigcup_{C_{i_{ann}}} cogen_{Class}(C_{i_{ann}}) \\
 & & \bigcup C^{et}_{Exp}(\_, \_, E_{main_{ann}})
\end{array}
$$

FIG. 6.38 – Générateur d'extensions génératrices pour le langage EFJ.

prenons celui appliqué aux accès à un champs $cogen_{Field}$, montré dans la figure 6.39. Notons que, comme ce type de construction peut être du type objet, le générateur considère le cas où l'annotation de temps de liaison est $\{\mathcal{S}, \mathcal{D}\}$. Les autres générateurs sont définis de la même façon.

Dans le cas des méthodes, le générateur $cogen_{Mthd}$ utilise le générateur $cogen_{Sign}$ pour la génération de la signature et le générateur $cogen_{Body}$ pour la génération du corps de l'extension génératrice correspondante.

Comme il s'agit d'une approche de spécialisation comportementale, le générateur d'extensions génératrices appliqué sur la définition d'un constructeur, $cogen_{Const}$, génère simplement une définition identique à celle du constructeur. Pour la même raison, les définitions des champs incluses dans la définition de la classe font aussi partie du code à résidualiser et, en conséquence, sont traiteés comme des constructions dynamiques.

$$
\begin{array}{lcl}
cogen_{Field}(E ::= e_0.\mathtt{f}) & = & \text{do } cogen_{Exp}(e_0) \\
& = & \text{case } \Gamma^{et}(E) \text{ of} \\
& & \{\mathcal{S}\} \Rightarrow (E,\bot,v_{e_0}) \\
& & \{\mathcal{D}\} \Rightarrow (\bot,build_{Field_{\mathcal{D}}}(\mathcal{D}_{e_0},\mathtt{f}),v_{e_0}) \\
& & \{\mathcal{S},\mathcal{D}\} \Rightarrow (build_{Field_{\mathcal{S}}}(\mathcal{S}_{e_0},\mathtt{f}),build_{Field_{\mathcal{D}}}(\mathcal{D}_{e_0},\mathtt{f}),v_{e_0})
\end{array}
$$

FIG. 6.39 – Générateur et constructeur pour l'acces aux champs : $cogen_{Field}$.

6.5 Bilan

Dans ce chapitre nous avons étudié la spécialisation dans les langages à objet par la génération d'extensions génératrices. La génération des extensions génératrices est basée sur un programme annoté selon de temps de liaison calculé suite à une analyse à étapes. Cette analyse se caractérise par l'inclusion d'une analyse de type concret pour améliorer le précision, généralement affectée par le mécanisme d'héritage et de polymorphisme intrinsèque aux langages à objets.

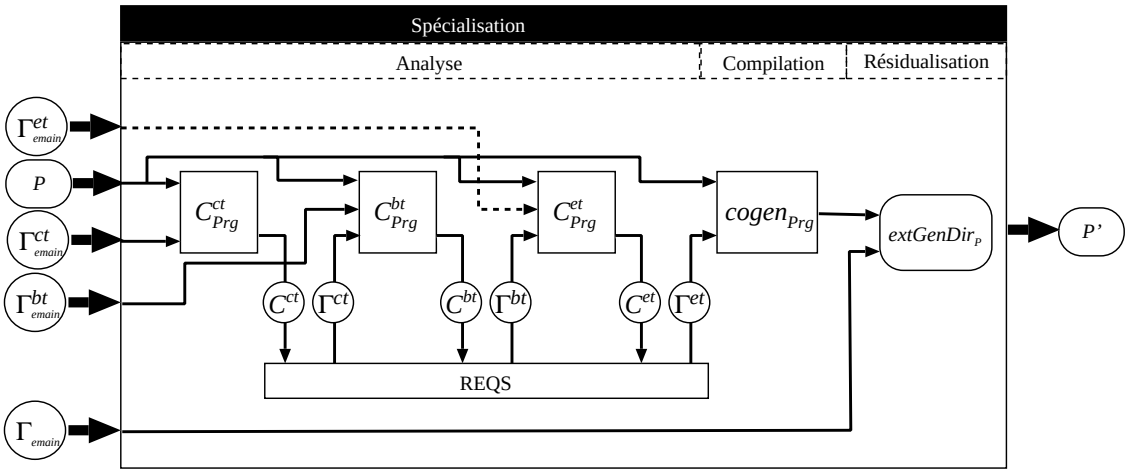


FIG. 6.40 – Analyse et spécialisation à l'aide d'extensions génératrices.

La figure 6.40 synthétise le processus de spécialisation proposé dans ce document. Nous avons défini et appliqué une approche à base de contraintes pour le calcul des annotations rèsultant résultant de chacun des étapes de l'analyse. Dans la figure on observe le caractère homogène de l'approche ainsi que les dépendances entre les annotations, par exemple, les annotations de temps de liaison Γ^{bt} dépendent des annotations de type concret Γ^{ct} pour la génération des contraintes correspondantes. La ligne pointillée associant les annotations initiales de temps d'évaluation $\Gamma^{et}_{E_{main}}$ à l'entrée du générateur de contraintes C^{et}_{Prg} indique que ces valeurs initiales ne sont parfois pas nécessaires.

Chapitre 7

Modèle minimal de composants

Sommaire

7.1 Caractéristiques du modèle **170**
7.1.1 Composants 170
7.1.2 Composition 172
7.1.3 Processus de développement 174
7.2 Instanciation de composants **176**
7.2.1 Instances partagées de composants 176
7.2.2 Instances de composants avec état 177
7.3 Description et implémentation de composants **180**
7.3.1 MoSCo-*CDL* 180
7.3.2 Conformité de l'implémentation par rapport à la définition des composants 181
7.4 Bilan **187**

Dans les modèles de composants étudiés dans le chapitre 3, nous observons l'existence de deux approches différentes, soit un langage de programmation étendu afin de supporter des constructions pour l'abstraction de composants, soit la définition d'applications à base de composants en utilisant deux langages : un langage pour la description architecturale et un langage pour la mise en œuvre de l'implémentation. Dans la première approche, comme evoqué dans le chapitre 3, la notion de composants en tant que boîtes noires ne s'avère pas très claire à cause de l'inclusion de l'implémentation comme partie de la description. C'est une des raisons principales pour laquelle nous avons choisi la deuxième approche. Le découpage dans la description des composants permettra, par exemple, d'étendre le langage de description de manière indépendante du langage d'implémentation. Dans ce chapitre, nous introduisons le modèle de composants *MoSCo* (*Model of Specializable Components*) qui permet le développement d'application à base de composant de manière similaire à celle des modèles étudiés dans la section 3. Dans le chapitre suivant, le modèle sera étendu afin de permettre l'adaptation des composants par rapport au contexte de réutilisation. Pour nous concentrer sur l'utilisation des techniques de spécialisation sur les applications à base de composants, le modèle mentionné conserve les caractéristiques minimales des modèles étudiés.

7.1 Caractéristiques du modèle

Dans le modèle *MoSCo* un composant est essentiellement considéré comme une boîte noire, c'est-à-dire que seul l'interface est visible au moment de la composition. Afin de se concentrer sur l'expression de composants, le modèle ne prend pas en compte la notion de conteneur ni des aspects comme, par exemple, la provision d'une couche supportant des services techniques tels que la persistance, les transactions, etc.

Dans cette section nous décrivons notre modèle sur la base des caractéristiques des modèles de composants décrites dans la section 3.3, à savoir la description et la composition de composants ainsi que les étapes du processus de développement.

7.1.1 Composants

En *MoSCo*, un composant est défini par une *interface*, qui spécifie les services fournis par le composant ainsi que les services requis, et l'*implémentation*, qui implémente les services fournis. Ici, un service a un seul point d'entrée représenté par la signature d'une fonction (méthode).

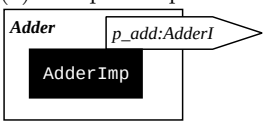
La figure 7.1(a) décrit graphiquement la structure du composant *Adder*, où les services sont représentés par des boîtes avec une forme triangulaire sur un des côtés. La boîte représentant les services requis *pénètre* les limites de l'interface tandis que celle représentant les services fournis *quitte* les limites de l'interfaces en suivant la direction marquée par le côté aiguë. Les services requis et fournis sont exposés dans l'interface du composant sous la forme d'un *port*. Un port est formé par une paire d'attributs : le *nom*, permettant d'identifier le port, et une *interface de port*, représentant un ensemble de services groupés selon une même sémantique. Dans la figure 7.1(a), on observe que le composant *Adder* fournit des services, à travers le port *p_add*, spécifiés dans la définition de l'interface de port *AdderI*.

Le producteur (ou le consommateur, comme expliqué dans la section 7.1.2) décrit l'interface d'un composant à l'aide du langage de description de composants MoSCo-*CDL* (voir section 7.3.1). Lors de la définition d'un composant, il suffit de fournir la description du composant proprement dit ainsi que la description associée aux interfaces des ports concernées. Les figures 7.1(b) et (c) présentent respectivement la description du composant *Adder* et celle de l'interface de port *AdderI*.

L'implémentation des composants, représentée par la boîte noire à l'intérieur de l'interface des composants, est réalisée en utilisant le langage de programmation EFJ (voir section 5.3). L'implémentation `AdderImp`, de la figure 7.1(d), représente le code mettant en œuvre les services fournis par le composant, dans le cas de *Adder* il s'agit des méthodes de l'interface `AdddderI` montrée dans la figure 7.1(e). Dans la section 7.3, nous expliquons plus en détail la relation entre le langage de description des composants MoSCo-*CDL* et le langage d'implémentation EFJ.

Un composant comme *Adder*, défini uniquement par la description et l'implémentation correspondante, est appelé *primitif*. En particulier, *Adder* est un composant primitif *sans dépendances* car il ne requiert aucun service. Par contre, dans le cas du composant *Multiplier* de la figure 7.1(f), il s'agit d'un composant primitif *avec dépendances* lesquelles sont

(a) Composant primitif sans dépendances *Adder*.



(b) Description du composant *Adder*.

fichier:../adder/adder.ldc
```
composant Adder {
 provides AdderI through p_add;
}
```

(c) Interface de port *AdderI*.

fichier:../adder/adderI.ldc
```
interface AdderI {
 int add(int, int);
}
```

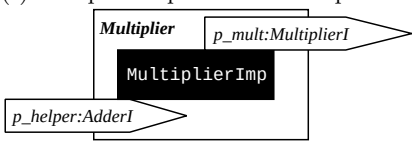
(d) Implémentation du composant.

fichier:../adder/AdderImp.java
```
class AdderImp implements AdderI {
 AdderImp(){
  super();
 }
 int add(int x, int y) {
  return x + y;
 }
}
```

(e) Implémentation de l'interface de port.

fichier:../adder/AdderI.java
```
interface AdderI {
 int add(int x, int y);
}
```

(f) Composant primitif avec dépendances *Multiplier*.



(g) Description du composant *Multiplier*.

fichier:../mult/multiplier.ldc
```
composant Multiplier {
 provides MultiplierI through p_mult;
 requires AdderI through p_helper;
}
```

(h) Interface de port *MultiplierI*.

fichier:../mult/multiplierI.ldc
```
interface MultiplierI {
 int multiply(int, int);
}
```

(i) Implémentation du composant.

fichier:../mult/MultiplierImp.java
```
class MultiplierImp implements MultiplierI {
 AdderI p_helper;
 MultiplierImp(AdderI p_helper){
  super();
  this.p_helper = p_helper;
 }

 int multiply(int x, int y) {
  return p_helper.add(x, multiply(x, y-1);
 }
}
```

(j) Implémentation de l'interface de port.

fichier:../mult/MultiplierI.java
```
interface MultiplierI {
 int multiply(int x, int y);
}
```

FIG. 7.1 – *MoSCo* : Définition des composants *Adder* et *Multiplier*.

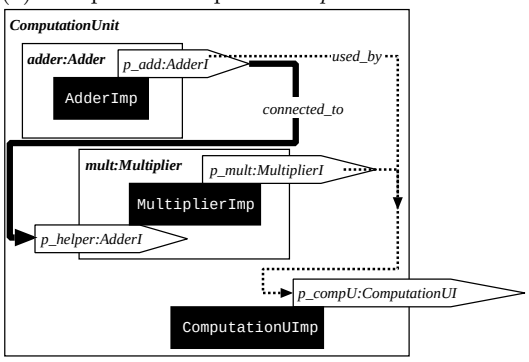
représentées par les services spécifiés par l'interface *AdderI* et requis, à leur tour, à travers le port *p_helper*. La définition du composant *Multiplier* inclut la description du composant ainsi que celles des interfaces de port *AdderI* (similaire à celle définie pour le composant *Adder* de la figure 7.1(c)), et *MultiplierI*, associée au port *p_mult*. La description du composant `Multiplier` et de l'interface `MultiplierI` sont montrées, respectivement, dans les figures 7.1(g) et (h).

La classe `MuliplierImp` de la figure 7.1(i) représente l'implémentation du composant. Celle-ci inclut le code implémentant les méthodes de l'interface des services fournis, c'est-à-dire les méthodes de l'interface `MultiplierI` de la figure 7.1(j). Observons que l'implémentation du service `multiply` se base sur le service `add` fourni à travers le port `p_helper`. Celui-ci est représenté dans l'implémentation du composant par une variable d'instance du même type que l'interface, c'est-à-dire `AdderI`, affectée au moment de l'instanciation du composant.

7.1.2 Composition

Dans le modèle *MoSCo*, l'assemblage de composants est effectué en connectant les services fournis d'un composant aux services requis d'un autre composant. Sur la base des composants comme ceux décrits dans la section précédente, le consommateur de composants peut définir de nouveaux composants, appelés *composants composés* (*composites*).

(a) Composant composé *ComputationUnit*.



(b) Description du composant *ComputationUnit*.

fichier:../compUnit/ComputationUnit.ldc

```
composant ComputationUnit {
 provides ComputationUI through p_compU;
 contains Adder adder, Multiplier mult;
 connects adder.p_add to mult.p_helper;
}
```

(c) Interface de port *ComputationUI*.

fichier:../compUnit/ComputationUI.ldc

```
interface ComputationUI {
 int add(int, int);
 int multiply(int, int);
 int subtract(int, int);
 int divide(int, int);
}
```

FIG. 7.2 – *MoSCo* : Définition du composant *ComputationUnit*.

Supposons, maintenant, la construction d'un composant qui fournit un ensemble de services pour effectuer des opérations arithmétiques simples comme la somme, la multiplication, la soustraction et la division de valeurs entières. La soustraction et la division peuvent être implémentées en appliquant l'addition d'une valeur négative, dans le premier cas, et en appliquant la soustraction de manière répétitive, dans le deuxième cas.

Une solution possible est la construction d'un nouveau composant, *ComputationUnitI* (voir figure 7.2(a)), en réutilisant les composants primitifs *Adder* et *Multiplier*.

La description des composants composés doit expliciter les références aux sous-composants ainsi que les connexions entre eux. Pour cette raison, dans la description du composant `ComputationUnit` de la figure 7.2(b), le producteur spécifie les sous-composants intervenants et leurs connexions en utilisant respectivement les clauses `contains` et `connects`. Dans ce cas, les services fournis par le composant *Adder* à travers le port *p_adder* sont utilisés pour satisfaire les services requis par le composant *Multiplier* à travers le port *p_helper*. La figure 7.2(c) présente la description de l'interface de port `ComputationUI`.

(a) Implémentation du composant.

fichier:../compUnit/ComputationUIImp.java

```
class ComputationUImp implements ComputationUI {
 Adder adder;
 Multiplier mult;
 Multiplier(Adder adder, Multiplier mult){
  super();
  this.adder = new Adder();
  this.mult = new Multiplier(this.adder);
 }
 int add(int x, int y) {
  return adder.p_add.add(x, y);
 }
 int multiply(int x, int y) {
  return mult.p_mult.multiply(x, y);
 }
 int subtract(int x, int y) { r
  return adder.p_add.add(x,-y);
 }
 int divide(int x, int y) {
  return (x > 0)
         ?adder.p_add.add(1, divide(sustract(x, y))
         :0;
 }
}
```

(b) Implémentation de l'interface de port.

fichier:../compUnit/ComputationUI.java

```
interface ComputationUI {
 int add(int x, int y);
 int multiply(int x, int y);
 int sustract(int x, int y);
 int divide(int x, int y);
}
```

FIG. 7.3 – *MoSCo* : Implémentation du composant *ComputationUnit*.

Finalement, la classe `ComputationUnitImp` présentée dans la figure 7.3(a) représente l'implémentation du composant *ComputationUnit*. Notons que, dans ce cas, il est possible d'utiliser les services fournis par tous les sous-composants, même s'ils sont connectés à d'autres services. C'est le cas de l'implémentation de la méthode `division` à l'aide de la méthode `add` fournie par l'implémentation du composant *Adder*, la classe `AdderImp`. Dans la figure 7.2(a), cette relation implicite entre les services fournis par les sous-composants et le composant englobant est signalée par la ligne pointillée. Tout comme les composants primitifs, les composants composés peuvent être aussi classifiés comme *sans* et *avec dépendances*. Le composant *ComputationUnit* est un exemple de composant composé sans dépendances.

```
 fichier:../adder/mosco-build.xml
<?xml version="1.0" encoding="UTF-8"?>
<mosco name="Adder" output="adder.jar" version="1.0">
  <component id="fr.emn.obasco.math.adder"
             label="Adder"
             file="../mosco/exemples/math/adder/Adder.cdl">
      <description/>
      <implementation>
          <class file="../mosco/exemples/math/adder/src/AdderImp.java" main="yes"/>
      </implementation>
  </component>
  <interface id="fr.emn.obasco.math.adderI"
             label="AdderI"
             file="../mosco/exemples/math/adder/AdderI.cdl">
      <description/>
      <implementation>
          <class file="../mosco/exemples/math/adder/src/AdderI.java" main="yes"/>
      </implementation>
  </interface>
</mosco>
```

FIG. 7.4 – *MoSCo* : Descripteur de développement du composant *Adder*

7.1.3 Processus de développement

Le processus de développement d'applications à base de composants dans le modèle décrit est conforme aux étapes mentionnées dans la section 3.3.3. En effet, les étapes concernées sont la *construction* et la *livraison* de composants effectuées par le producteur ainsi que l'*assemblage* et l'*exécution* de composants effectuées par le consommateur.

7.1.3.1 Construction

Lors de la construction des composants *MoSCo*, le producteur de composants doit fournir la description du composant ainsi que l'implémentation comme expliqué dans les sections précédentes. Une fois ces deux éléments spécifiés, le producteur établit leur liaison à l'aide d'un *descripteur de développement*. Ce descripteur, exprimé sous la forme d'un fichier XML (*mosco-build.xml*), lie les éléments de la description du composant (*i.e.*, fichiers *.cdl*) et ceux de l'implémentation (fichiers *.java*). La figure 7.4 présente le descripteur de développement associé au composant *Adder*. Notez que la description du composant est représentée dans le descripteur par les éléments `<component>` et `<interface>`. Pour chacun de ces éléments, le descripteur permet de spécifier le fichier contenant la description (attribut `file`) ainsi que l'implémentation (sous-élément `<implementation>`) en énumérant les classes EFJ correspondantes (sous-élément `<class>`).

7.1.3.2 Livraison

Le descripteur de développement est utilisé par l'environnement de développement (voir chapitre 9) pour packager la description du composant sous la forme d'un fichier *.jar*, dont le nom est indiqué par l'attribut `output` dans l'élément racine `<mosco>`.

```
 fichier:../compUnit/mosco-build.xml
<?xml version="1.0" encoding="UTF-8"?>
<mosco name="ComputationUnit" output="computationunit.jar" version="1.0">
  <component id="fr.emn.obasco.math.computationunit"
             label="ComputationUnit"
             file="../mosco/math/adder/ComputationUnit.cdl">
      <description/>
      <implementation>
          <class file="../mosco/math/adder/src/ComputationUnit.java" main="yes"/>
      </implementation>
  </component>
  <interface id="fr.emn.obasco.math.computationunitI"
             label="ComputationUnitI"
             file="../mosco/math/adder/ComputationUI.cdl">
      <description/>
      <implementation>
          <class file="../mosco/math/adder/src/ComputationUI.java" main="yes"/>
      </implementation>
  </interface>
  <contains>
      <subcomponent id="Adder" version="1.0" jar-file="../mosco/adder/adder.jar"/>
      <subcomponent id="Multiplier" version="1.0" jar-file="../mosco/adder/multiplier.jar"/>
    </contains>
</mosco>
```

FIG. 7.5 – *MoSCo* : Descripteur de développement du composant *ComputationUnit*

La figure 7.5 montre le descripteur du développement associé au composant *ComputationUnit*. Notons que, en plus des éléments mentionnés ci-dessus, le producteur, en tant que consommateur des composants, doit inclure dans le descripteur l'information sur les sous-composants *Adder* et *Multiplier* en référençant leur fichier *.jar* correspondant. Toute l'information utilisée lors de la construction n'est pas nécessaire lors de la réutilisation, (par exemple, l'information sur les sous-composants inclus dans la composition). Pour cette raison, le fichier *.jar* délivré n'inclut qu'une version réduite du descripteur initial, appelé ici *descripteur de déploiement* (*mosco.xml*). Ce nouveau descripteur contiendra l'information nécessaire pour la réutilisation, principalement les services fournis et requis. La figure 7.6 présente le descripteur de déploiement pour le composant *ComputationUnit*.

7.1.3.3 Assemblage

L'assemblage de composants s'effectue en connectant les services fournis d'un composant aux services requis d'un autre composant de la même façon qu'on a défini le com-

```
 fichier:../compUnit/mosco.xml
<?xml version="1.0" encoding="UTF-8"?>
<mosco name="ComputationUnit" version="1.0">
  <component id="fr.emn.obasco.mosco.math.computationunit" label="ComputationUnit">
    <description/>
    <implementation>
        <class file="fr.emn.obasco.mosco.math.computationunit.ComputationUnit.class"/>
    </implementation>
  </component>
  <interface id="fr.emn.obasco.mosco.math.computationuniti" port="p_compU" role="provides">
    <description/>
    <implementation>
        <class file="fr.obasco.mosco.math.computationunit.ComputationUnitI.class"/>
    </implementation>
  </interface>
</mosco>
```

FIG. 7.6 – *MoSCo* : Descripteur de déploiement du composant *ComputationUnit*

posant *ComputationUnit* (voir figure 7.2). Le résultat de l'assemblage peut être, soit un composant qui ne requiert aucun service, c'est-à-dire un composant sans dépendances (voir figure 7.1(a)), soit un composant qui requiert un ensemble de services, c'est-à-dire un composant avec dépendances (voir figure 7.1(f)).

7.1.3.4 Exécution

L'exécution d'un composant n'est possible que dans le cas où toutes les dépendances ont été satisfaites. C'est le cas des composants *Adder* et *ComputationUnit*, lesquels peuvent être exécutés simplement à partir de l'instanciation de la classe de composant correspondante. Dans la section 7.2, nous abordons l'instanciation des composants *MoSCo* de façon plus detaillée.

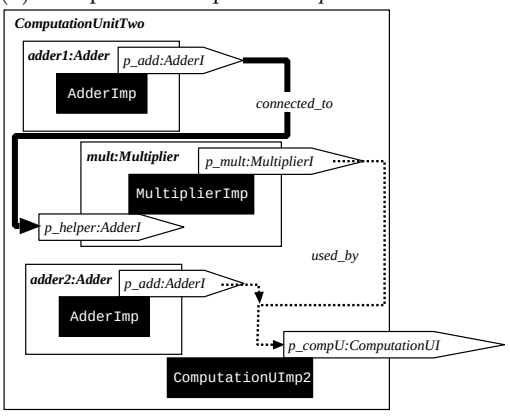
7.2 Instanciation de composants

7.2.1 Instances partagées de composants

Dans la description d'un composant résultant d'une composition, tel que le composant *ComputationUnit* (voir figure 7.3), on observe que la définition des sous-composants est réalisée en termes d'instances de composants. Dans l'exemple mentionné, la structure est formée par une instance du composant *Adder* et une instance du composant *Multiplier*, lesquelles sont référencées par l'identificateur *adder* et *mult* respectivement. Notez que, dans ce cas, le composant englobant et le sous-composant *Multiplier* partagent la même instance du composant *Adder*. Cependant, il est possible d'utiliser deux instances différentes du composant *Adder* comme celles utilisées dans la description du composant

ComputationUnitTwo de la figure 7.7(a). La figure 7.7(b) et (c) présentent la description et l'implémentation correspondantes.

(a) Composant composé *ComputationUnitTwo*.



(b) Description du composant.

fichier:../compUnit/ComputationUnit2.ldc

```
composant ComputationUnitTwo {
 provides ComputationUI through p_compU;
 contains Adder adder1, adder2;
 contains Multiplier mult;
 connects adder1.p_add to mult.p_helper;
}
```

(c) Implémentation du composant.

fichier:../compUnit/ComputationUnitImp2.java

```
class ComputationUnit2
      implements ComputationUnitI {
 Adder adder1, adder2;
 Multiplier mult;
 ComputationUnit2(Adder adder1,
                  Adder adder2,
                  Multiplier mult){
  super();
  this.adder1 = new Adder();
  this.mult = new Multiplier(this.adder);
  this.adder2 = new Adder();
 }
 int add(int x, int y) {
  return adder2.p_add.add(x, y);
 }
 ...
}
```

FIG. 7.7 – *CMname* : Composant `ComputationUnit` avec deux instances du composant `Adder`

7.2.2 Instances de composants avec état

Nous avons vu dans l'exemple de la section précédente que le partage d'instances de composants ne présente aucune restriction. Par contre, dans le cas des composants avec un état, exprimé par les attributs associés, le partage des instances est plus limité. Pour mieux illustrer cette possibilité, nous allons maintenant décrire le composant *Clock* (voir section 3.6) sous la forme d'une assemblage de composants *MoSCo*.

Reprenons l'implémentation ArchJava du composant *Clock* présentée dans la figure 3.30. La figure 7.8(a) montre la description du composant *Counter* qui fournit les services, spécifiés par l'interface de port *CounterI* (voir figure 7.8(b)), à travers le port `p_out`. Rappelons aussi que dans le langage EFJ, les objets sont des entités immutables par rapport à l'état, c'est-à-dire que les valeurs des champs sont affectées uniquement au

(a) Description du composant *Counter*.

fichier:../clock/counter.ldc
```
composant Counter {
 provides CounterI through p_out;
 int base;
 int value;
}
```

(b) Interface de port *CounterI*.

fichier:../clock/counterI.ldc
```
interface CounterI {
 int getValue();
 Object tick();
}
```

(c) Implémentation du composant.

fichier:../adder/CounterImp.java
```
class CounterImp implements CounterI {
 CounterImp(int base,int value){
  super();
  this.base = base; this.value = value;
 }
 int getValue() {
  return this.value;
 }
 Object tick(){
  return
   (((this.value + 1) > this.base)
   ? new CounterImp(this.base,0)
   : new CounterImp(this.base,this.value+1));
 }
}
```

(d) Description du composant *DCounter*.

fichier:../clock/dcounter.ldc
```
composant DCounter {
 provides CounterI through p_out;
 requires CounterI through p_in;
 int base;
 int value;
}
```

(e) Implémentation du composant.

fichier:../adder/DCounterImp.java
```
class DCounterImp implements CounterI {
 CounterI p_in;
 int base;
 int value;
 DCounterImp(CounterI p_in,int base,int value) {
  super();
  this.p_in = p_in;
  this.base = base; this.value = value;
 }
 int getValue() {
  return this.value;
 }
 Object tick(){
  return
   (((this.value + 1) > this.base)
   ? new DCounterImp(p_in.tick(),this.base,0)
   : new DCounterImp(p_in, this.base,
                              this.value+1));
 }
}
```

FIG. 7.8 – *MoSCo* : Définition des composants *Counter* et *DCounter*.

moment de l'instanciation. L'altération de l'état d'un objet est réalisée par la création d'un nouvel objet. Dans l'implémentation du service `tick` de la figure 3.30(a), on observe que l'état de l'objet concerné change. Pour cette raison, le service `tick` de l'interface de port `CounterI`, contrairement à l'implémentation originale, doit retourner une valeur de type `Objet` qui représentera les nouvelles valeurs du compteur comme le montre l'implémentation de la méthode `tick` de la classe `CounterImp` dans la figure 7.8(c).

En ce qui concerne le composant *DCounter*, la structure est similaire au composant *Counter* mais en requérant les services *CounterI* par le port *p_in* comme montré par la description `DCounter` de la figure 7.8(d). La méthode `tick` de la classe `DCounterImp`, l'implémentation du composant *DCounter* (voir figure 7.8(e)), crée également un nouvel objet chaque fois que les valeurs du compteur changent. Dans ce cas, lors d'un changement de valeur, on trouve que l'instance du composant qui fournit les services requis, c'est-à-dire

(a) Description du composant *Clock*.

fichier:../clock/clock.ldc

```
composant Clock {
 provides ClockI through p_clock;
 contains Counter hh;
          DCounter mm;
          DCounter ss;
 connects hh.p_out to mm.p_in;
          mm.p_out to ss.p_in;
}
```

(b) Interface de port *ClockI*.

fichier:../clock/clockI.ldc

```
interface ClockI {
 int getTime();
 Object tick();
}
```

(c) Implémentation du composant.

fichier:../adder/ClockImp.java

```
class ClockImp implements ClockI {
 Counter hh;
 DCounter mm,ss;
 ClockImp(Counter hh, DCounter mm, DCounter ss){
  super();
  this.hh = new Counter(hh.base,hh.value);
  this.mm = new DCounter(mm.base,mm.value,this.hh);
  this.ss = new DCounter(ss.base,ss.value,this.mm);
 }
 int getTime() {
  return (this.hh.getValue() * 3600) +
         (this.mm.getValue() * 60) +
         this.ss.getValue();
 }
 Object tick(){
  return new ClockImp(this.hh, this.mm,this.ss.tick());
 }
}
```

(d) Instanciation du composant.

```
new ClockImp(new Counter(0,24),
             new DCounter(0,60,null)
             new DCounter(0,60,null));
```

FIG. 7.9 – *MoSCo* : Description du composant *Clock*.

celle représentée par la variable `p_in`, change aussi.

Finalement, le composant *Clock* est construit à partir de la composition des composants décrits ci-dessus. Tout comme dans l'implémentation ArchJava, ici, la structure du composant *Clock* est formée par deux instances du composant *DCounter* et une instance du composant *Counter* comme montré dans la figure 7.9(a). La classe `ClockImp` de la figure 7.9(c) en tant qu'une implémentation conforme à la grammaire de EFJ, est aussi une implémentation possible pour le composant *Clock* (voir règles de conformité de la section 7.3.2). En effet, les paramètres du constructeur de cette classe permettent l'initialisation de tous les champs de la classe qui, dans ce cas, représentent des instances des sous-composants. L'instanciation de la classe du composant requiert donc la création des sous-composants associés à la structure, comme le montre la figure 7.9(d).

Par conséquent, l'utilisation d'un composant ainsi construit ne respecte pas l'approche de réutilisation de composants comme une boîte noire. Autrement dit, la structure d'un composant composé doit rester transparente pour les consommateurs. Cependant, comme montré par l'implémentation de la méthode `tick`, un constructeur ainsi défini permet la gestion de l'état des composants. En effet, la méthode `tick` de la classe `ClockImp` retourne un nouvel objet représentant une nouvelle instance du composant où les sous-composants sont aussi eux-mêmes représentés par de nouvelles instances.

Dans la section 11.2.1, nous expliquons comme le modèle *MoSCo* peut être étendu afin de conserver la tranparence de la structure des composants lors de l'instanciation.

7.3 Description et implémentation de composants

Dans cette section nous étudions en détail le langage de description et le langage d'implémentation utilisés pour la définition des composants.

Par les raisons citées dans la section 7.2.2, l'implémentation d'un composant avec état devient notamment plus compliquée. Celle-ci est une conséquence des limitations imposées par la définition du langage d'implémentation EFJ. Afin d'illustrer la spécialisation de composants d'une façon simple, en ce qui concerne les paramètres d'entrée et la valeur de retour des services des composants, ils sont restreints aux types primitifs `int` et `boolean`. Bien entendu, cette restriction est applicable seulement sur les méthodes de la classe du composant, c'est-à-dire qu'aucune restriction n'existe sur les méthodes des classes appartenant à l'implémentation du composant autre que la classe du composant.

En ce qui concerne l'état d'un composant, les attributs sont également restreints aux types primitifs. En principe, le passage de paramètres de type objet rend la composition de composants dépendante de la définition de tels objets. Par exemple, en utilisant un type qui n'est pas dans l'ensemble de bibliothèque communément disponibles dans l'environnement Java, la description d'un composant peut s'avérer incompatible pour des réutilisations ultérieures. Cette restriction facilite aussi la définition de scénarios de spécialisation comme nous l'expliquons dans le chapitre 8.

7.3.1 MoSCo-*CDL*

La figure 7.10 montre la grammaire du langage MoSCo-*CDL* qui a été utilisée dans les sections précédentes pour la description des composants. Ici, nous définissons la description d'une application à base de composants, dénotée par *Dc*, comme la paire formée par une description de composant, dénotée *Cp*, et celle des interfaces de ports, dénotée par $\overline{PI}$.

7.3.1.1 Description d'un composant

La description d'un composant contient des ports, *Po*, des attributs, *At* et, dans le cas d'un composant composé, une structure, *St*.

Un port est décrit par son rôle, `provides` ou `requires`, l'ensemble de services donné par l'identificateur de l'interface de port, *interfaceId* et l'identificateur du port proprement dit, *portId*.

Un composant *MoSCo* peut avoir un état associé. Il est donc possible de définir un ensemble d'attributs lesquels sont décrits simplement par un type et un identificateur. Du fait de notre modèle simple, les attributs ne peuvent être que de type primitif, à savoir `int` et `bool`.

Finalement, les sous-composants nécessaires à la description d'un composant composé sont spécifiés en utilisant la clause `contains` dont les ports sont connectés par des clauses `connects` en liant les port fournis aux (`to`) ports requis.

Dc	$\in$	$Description$	::=	$Cp\ \overline{PI_n}$
Cp	$\in$	$Component$	::=	`component` $cId\ \{\overline{Po_p}\ \overline{At_a}\ St\}$
Po	$\in$	$Port$	::=	$PC \mid RC$
PC	$\in$	$ProvidesClause$	::=	`provides` iId `through` pId `;`
RC	$\in$	$RequiresClause$	::=	`requires` iId `through` pId `;`
At	$\in$	$Attribute$	::=	$p\ aId$ `;`
p	$\in$	$PrimitiveType$	::=	`int`| `boolean`
St	$\in$	$Structure$	::=	$\overline{Ct_b}\ \overline{Cn_x}$
Ct	$\in$	$ContainsClause$	::=	`contains` $cId\ ciId$ `;`
Cn	$\in$	$ConnectsClause$	::=	`connects` CPR_{src} `to` CPR_{des} `;`
CPR	$\in$	$ComponentPortRef$	::=	$ciId.pId$
PI	$\in$	$PortInterface$	::=	`interface` $iId\ \{\overline{Sv_s}\}$
Sv	$\in$	$Service$	::=	$p\ sId($`(`$\overline{p\ x}$`)`$;$

FIG. 7.10 – *MoSCo*-CDL : Grammaire

7.3.1.2 Description des interfaces de ports

La description de l'interface d'un port prend la forme de la déclaration d'une interface dans le langage Java c'est-à-dire un ensemble de signatures de méthodes sans corps associé. Cependant, dans le langage d'implémentation EFJ (voir section 5.3), les interfaces ne font pas partie des constructions permises. En conséquence, elles sont remplacées par des classes, en respectant les restrictions stipulées par le langage EFJ, comme nous le verrons dans la section 7.3.2.2.

7.3.2 Conformité de l'implémentation par rapport à la définition des composants

Une approche découplant la description d'une application de son implémentation doit garantir la conformité entre les descriptions fournies et l'implémentation correspondante. Dans ce contexte, une application est définie par $(Dc, \overline{C}, D_D)$, où Dc est la description du composant et $\overline{C}$ est l'implémentation correspondante, constituée d'un ensemble de classes, lesquelles sont reliées à la description Dc par le descripteur de développement D_D.

Définition auxiliaires. Tout comme dans la définition du langage EFJ (voir figure 5.13), les définitions auxiliaires décrites dans la figure 7.11 sont utilisées pour l'extraction d'information des éléments des descriptions *Cp* et *PI*.

Dans le cas de la description d'un composant *Cp*, les éléments concernés sont les ports $\overline{Po}$, les attributs $\overline{At}$ et la structure $\overline{St}$. Pour extraire l'information des ports de la description d'un composant on utilise la fonction *ports*. Par contre, quand il est souhaitable

Ports :

$$\frac{Cp ::= \texttt{component}\ \_\ \{\overline{Po_p}\ \_\ \_\}\qquad \overline{Po_p} = \overline{PC_o} :: \overline{RC_r}}{ports(Cp) = \overline{Po_p}\qquad providedPorts(Cp) = \overline{PC_o}\qquad requiredPorts(Cp) = \overline{RC_r}}$$

$$\frac{Po ::= \_\ iId\ \texttt{through}\ pId;}{piId(Po) = iId\qquad portId(Po) = pId}$$

Attributs :

$$\frac{Cp ::= \texttt{component}\ \_\ \{\_\ \overline{At_a}\ \_\}}{attributes(Cp) = \overline{At_a}}\qquad \frac{At ::= p\ aId;}{atId(At) = aId\qquad atType(At) = p}$$

Structure :

$$\frac{Cp ::= \texttt{component}\ \_\ \{\_\ \_\ St\}\qquad St := \overline{Ct_b}\ \overline{Cn_x}}{subcomponents(Cp) = \overline{Ct_b}\qquad connections(Cp) = \overline{Cn_x}}$$

- Sous-composants :

$$\frac{Ct ::= \texttt{contains}\ cId\ ciId;}{componentType(Ct) = cId\qquad instanceId(Ct) = ciId}$$

- Connexions :

$$\frac{Cn ::= \texttt{connects}\ CPR_{src}\ \texttt{to}\ CPR_{des};}{refConnectedPort(Cn) = (CPR_{src}, CPR_{des})}$$

- Référence sur ports :

$$\frac{CPR ::= ciId.pId}{refCompInstanceId(CPR) = ciId\qquad refPortId(CPR) = iId}$$

Composant :

$$\frac{Cp ::= \texttt{component}\ ciId\ \{\_\ \_\ \_\}}{componentId(Cp) = cId}$$

Interfaces :

$$\frac{PI ::= \texttt{interface}\ iId\ (\overline{Sv_s})}{interfaceId(PI) = cId\qquad interfaceServices(PI) = \overline{Sv_s}}$$

$$\frac{Dc = Cp\ \overline{PI}\qquad ports(Cp) = \overline{Po_p}\qquad piId(Po) = interfaceId(PI)}{portInterface(Po) = PI}$$

FIG. 7.11 – MoSCo-*CDL* : Définitions auxiliaires.

d'obtenir les ports d'un rôle déterminé, fournis où requis, les fonctions utilisées sont respectivement *providedPorts* et *requiredPorts*. L'interface de port et l'identifiant spécifiés dans la déclaration d'un port donné s'obtiennent par les fonctions *piId* et *portId*. La fonction *attributes* permet d'obtenir l'ensemble des attributs associés au composant tandis que l'identificateur et le type d'un attribut déterminé sont calculés à partir des fonctions *atId* et *atType*. Afin de manipuler l'information sur la structure d'un composant, *St*, nous utilisons la fonction *subcomponents* qui retourne l'ensemble des clauses de sous-composants, *Ct*, et la fonction *connexions* qui retourne l'ensemble des clauses de connexions des ports, *Cn*. Dans le premier cas, les fonctions *componentType* et *instanceId* sont appliquées sur une clause *Ct* donnée afin d'obtenir l'identificateur et le nom d'instance correspondant. Dans le deuxième cas, les références sur le port des services fournis, CPR_{src}, et sur le port des service requis, CPR_{des}, sont retournées par la fonction *refConnectedPorts*. On

utilise les fonctions *refCompInstanceId* et *refPortId* quand l'identificateur de l'instance du composant et celui du port référencé, spécifiés dans leurs définitions, sont nécessaires.

Dans le cas des descriptions d'interfaces de ports *PI*, la fonction *interfaceId* retourne l'identificateur de l'interface de port et la fonction *interfaceServices* retourne l'ensemble des services déclarés dans l'interface de port *PI*. La fonction *portInterface* retourne la description d'une interface de port *PI* utilisée dans la définition d'un port *Po*.

7.3.2.1 Règles de Conformité

La figure 7.12 montre les règles permettant vérifier la conformité (*wf-rules*) de l'implémentation $\overline{C}$ par rapport à la description Dc dont les éléments sont liés par le descripteur de développement D_D.

Pour des raisons d'espace nous omettons la description des fonctions auxiliaires concernant l'information contenue dans les descripteurs de développement D_D, tel que $interfaceImpl$, $componentMainImpl$ et $subComponentImpl$. En général, ces fonctions servent à extraire les identifiant des classes qui implémentent les différents éléments de la description donnée (la classe qui implémente le composant ainsi que celles des sous-composants,...).

Ports :

$$\frac{Dc = Cp\ \overline{PI} \quad providedPorts(Cp) = \overline{PC_o} \quad \forall\ i\ \in [1,o]\ portInterface(PC_i) = PI \quad interfaceImpl(PI, D_D) = c \quad componentMainImpl(D_D) = d \quad c \in\ interfaces(d)}{(Dc, \overline{C}, D_D)\ \textit{wf-providedPorts}} \ [\ \textbf{P-PORT}^{CDL}]$$

$$\frac{Dc = Cp\ \overline{PI} \quad requiredPorts(Cp) = \overline{RC_r} \quad \forall\ i\ \in [1,r]\ portInterface(RC_i) = PI \quad interfaceImpl(PI, D_D) = c \quad componentMainImpl(D_D) = d \quad fields(d) = \overline{t\ f} \quad t = c \quad portId(RC_i) = f}{(Dc, \overline{C}, D_D)\ \textit{wf-requiredPorts}} \ [\ \textbf{R-PORT}^{CDL}]$$

Attributs :

$$\frac{attributes(Cp) = \overline{At_a} \quad componentMainImpl(D_D) = c \quad fields(c) = \overline{t\ f} \quad \forall\ i\ \in [1,a]\ atId(At_i) = f \quad atType(At_i) = t}{(Dc, \overline{C}, D_D)\ \textit{wf-attributs}} \ [\ \textbf{ATTR}^{CDL}]$$

Structure :

$$\frac{subcomponents(Cp) = \overline{Ct_t} \quad \forall\ i\ \in [1,b]\ subComponentImpl(Ct_i, D_D) = d \quad componentMainImpl(D_D) = c \quad fields(c) = \overline{t\ f} \quad t = d \quad instanceId(Ct_i) = f}{(Dc, \overline{C}, D_D)\ \textit{wf-contain}} \ [\ \textbf{CONT}^{CDL}]$$

$$\frac{\begin{array}{c} connections(Cp) = \overline{Cn_x} \quad \forall\ i\ \in [1,x]\ refConnectedPorts(Cn_i) = (CPR_{src_i}, CPR_{des_i}) \\ componentInstImpl(CPR_{des_i}, D_D) = c_{des_i} \quad componentInstImpl(CPR_{src_i}, D_D) = c_{src_i} \\ refCompInstanceId(CPR_{src_i}) = ciId_{src_i} \quad refCompInstanceId(CPR_{des_i}) = ciId_{des_i} \\ componentMainImpl(D_D) = c \quad fields(c) = \overline{t\ f} \quad \texttt{this}.f = \texttt{new}\ c_{des_i}\{\ldots \texttt{this}.f_k, \ldots\}\ \text{in}\ K_c \\ f = ciId_{des_i} \quad t = c_{des_i} \quad K_{c_{des_i}} = c_{des_i}(\overline{e\ g}) \quad g = ciId_{src_i} \quad e = c_{src_i} \end{array}}{(Dc, \overline{C}, D_D)\ \textit{wf-connection}} \ [\ \textbf{CONN}^{CDL}]$$

$$\frac{(Dc, \overline{C}, D_D)\ \textit{wf-contain} \quad (Dc, \overline{C}, D_D)\ \textit{wf-connection}}{(Dc, \overline{C}, D_D)\ \textit{wf-structure}} \ [\ \textbf{STRUC}^{CDL}]$$

FIG. 7.12 – MoSCo-*CDL* : Règles de conformité.

Ports. En ce qui concerne les ports on vérifie la conformité des ports fournis et celle des port requis. Dans le premier cas (règle **P-PORT**^{CDL}), on vérifie que pour tous les ports fournis, donnés par la fonction *portInterface*, sont implémentés par des interfaces incluses dans une classe d de l'implémentation fournie. Les interfaces associées à une classe sont calculées en appliquant la fonction *interfaces* avec la classe comme argument (voir section 7.3.2.2). Dans le deuxième cas (règle **R-PORT**^{CDL}), en plus de vérifier l'existence de l'implémentation correspondant à l'interface de port, la classe du composant doit inclure un champ du même type que la classe implémentant le port donné par la fonction *portId*. C'est le cas, par exemple, de la clause `requires AdderI through p_helper` incluse dans la description `Multiplier`, à laquelle correspond la définition du champ `p_helper` de type `AdderI` dans la classe du composant `MultiplierImp.java` (voir figure 7.1(i)). Dans les deux cas, le nom du fichier Java implémentant l'interface fait partie de l'information incluse dans le descripteur de développement D_D.

Attributs. Les attributs des composants sont représentés dans l'implémentation par des champs dont le nom et le type sont identiques aux nom et type données par les fonctions *atId* et *atType* dans les déclaration des attributs.

Structure. La conformité de la structure de la description *St*, exprimée par la règle **STRC**^{CDL}, se base sur celles des différents éléments, à savoir les clauses des sous-composants, *Ct*, et leurs connexions, *Cn*.

Une première règle (règle **CONT**^{CDL}) stipule que chacun des sous-composants est représenté par un champ dans la classe du composant. Ce champ a comme nom l'identificateur spécifié dans la définition de la clause, fourni par la fonction *instanceId* et comme type la classe d implémentant le sous-composant spécifié dans le descripteur de développement D_D. Cette classe est obtenue à travers la fonction *subComponentImpl* laquelle prend l'identificateur du type associé au composant dans la clause Ct_i, donné par la fonction *componentType*, ainsi que le descripteur D_D. Dans l'exemple du descripteur de développement du composant *ComponentUnit* montré dans la figure 7.5, la valeur à retourner est cherchée dans les descripteurs de déploiement inclus dans le fichier *.jar* des sous-composants correspondants. Par exemple, le composant *Adder* est implémenté par la classe `AdderImp` comme indiqué par la balise `<implementation>` du descripteur de déploiement de la figure 7.4.

Une deuxième règle (règle **CONN**^{CDL}) vérifie la connexion établie entre les composants qui a lieu lors de l'instanciation du composant. D'une part, la règle vérifie la connexion du type de l'instance affectée à chaque champ représentant un sous-composant. D'autre part, si un sous-composant requiert les services d'un autre sous-composant alors la référence au champ du composant fournisseur comme argument lors de l'instanciation du composant requérant. Par exemple, dans le constructeur de la classe `ComputationUnitImp` de la figure 7.3(a), le champ du sous-composant `mult` est affecté à une instance de la classe `Multiplier` avec la référence au composant *Adder* comme argument.

Composants. Finalement, la conformité d'une implémentation par rapport à une déscription donnée est exprimée par la règle **COMP**^{CDL} (voir figure 7.13).

Component :
$Dc = Cp\ PI$ D_D development descriptor $\overline{C}$ implementation
$(Dc, \overline{C}, D_D)$ *wf-providedPorts* $(Dc, \overline{C}, D_D)$ *wf-requiredPorts*
$(Dc, \overline{C}, D_D)$ *wf-attributs* $(Dc, \overline{C}, D_D)$ *wf-structure*
$\frac{}{(Dc, \overline{C}, D_D)\ \textit{wf-component}}$ [$\mathbf{COMP}^{CDL}$]

FIG. 7.13 – MoSCo-*CDL* : Règles de conformité (suite).

7.3.2.2 Interfaces

Selon la règle de conformité **P-PORT**^{CDL}, les interfaces de port fournies par le composant sont représentées par des interfaces implémentées par la classe du composant. Cependant, les interfaces ne sont pas des constructions acceptées par la grammaire de EFJ (section 5.3). En fait, l'idée sous-jacente dans l'utilisation des interfaces est d'assurer que tous les services fournis ont une implémentation associée. Les interfaces représentent les types associés à la classe du composant, cela permet d'utiliser une instance du composant où un sous-composant fournissant certains services inclus dans les interfaces concernées. Il est possible de définir des règles de conformité complémentaires afin de garantir que tous les services fournis sont implémentés par une méthode dans la classe du composant. Ainsi, la spécification des interfaces n'est plus nécessaire. Cependant, cette solution n'est pas adéquate quand le composant doit fournir des services à travers plusieurs ports.

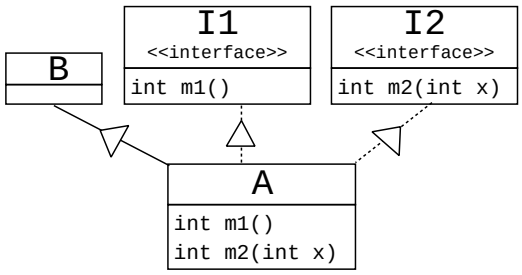
Pour cette raison, nous proposons, tout d'abord, la transformation des interfaces en classes et, ensuite, la liaison de ces classes par la relation d'héritage. La figure 7.14(a) et 7.14(b) présente le diagramme de classes et l'implémentation d'une classe de composant `A` qui étend la classe `B` et implémente les interfaces `I1`, `I2`.

Transformation d'une interface. Le résultat de la transformation d'une interface est une classe du même nom déclarant toutes les méthodes spécifiées par l'interface. La figue 7.14(c) montre les classes `I1` et `I2` représentant les interfaces du même nom dont le corps des méthodes est défini par un appel de la même méthode sur une instance de la même classe.

Transformation de la hiérarchie. En ce qui concerne la hiérarchie, les classes résultantes sont reliées entre elles et définissent une nouvelle hiérarchie où la racine est étendue par la classe de composants. Les classes représentant les interfaces sont chaînées par la relation d'héritage dont la racine est définie elle-même comme une sous-classe de la classe étendue par la classe de composant. Dans la figure 7.14(e), nous observons la hiérarchie résultante où la classe `B` est la racine et la classe de composant `A` est la feuille. En conséquence, toutes les interfaces ont disparues de l'implémentation et une instance du composant est de tous les types (interfaces) nécessaires. Dans l'exemple, la classe `A` est de type `I1` donc elle peut être utilisée où les services de l'interface `I1` sont requis. Le même raisonnement peut être appliqué pour les services de l'interface `I2` car la classe est aussi de type `I2`.

Cette technique a été utilisée dans l'implémentation du programme *Point2DStepping*

(a) Hiérarchie originale.



(b) Implémentation originale.

```
class A extends B implements I1,I2 {
 A(){
  super();
 }
 int m1(){...}

 int m2(int x){...}
}

interface I1 { int m1(); }

interface I2 { int m2(int x); }
```

(c) Classes résultantes.

```
class I1 extends I2{
 int m1(){ return this.m1();}
}

class I2 extends B {
 int m2(int x){ return this.m2(x);}
}
```

(d) Implémentation transformée.

```
 classe A extends I1 {
 A(){
  super();
 }
 int m1(){...}
 int m2(){...}
}
```

(e) Hiérarchie transformée.

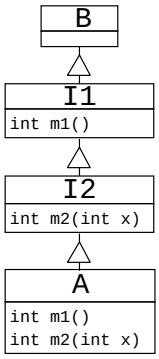


FIG. 7.14 – *MoSCo* : Transformation des interfaces.

(voir section 5.3.2) où les classes `Point2DX` et `StepperByX` sont définies de la manière expliquée ci-dessus. Notez qu'une classe ainsi définie devient une classe abstraite car elle ne peut pas être instanciée.

La figure 7.15 montre la définition de la fonction *interfaces* applicable sur les implémentations résultants des transformations décrites ci-dessus.

$$\frac{\texttt{class C extends D \{...\}} \qquad \texttt{C} \in \overline{\texttt{C}_m}}{interfaces(\texttt{C}, \overline{\texttt{C}_m}) = \texttt{D} \cup interfaces(\texttt{D}, \overline{\texttt{C}_m})}$$

FIG. 7.15 – MoSCo-*CDL* : Fonction auxiliaire *interfaces*.

7.4 Bilan

Ce chapitre a décrit un modèle minimal de composants, *MoSCo*, lequel se caractérise par la simplicité dans la définition des composant. Un composant dans le contexte de *MoSCo* est défini à partir d'une description architecturale en utilisant le langage de description de composants MoSCo-*CDL* à laquelle est associée une implémentation qui prend la forme d'un ensemble de classes EFJ. Nous avons abordé aussi les étapes du processus de développement ainsi que les limitations du langage d'implémentation choisi dans la gestion de l'état d'un composant. Finalement, nous avons explicité les règles de conformité qui permettent de s'assurer de la cohérence de la description d'un composant et de son implémentation.

Chapitre 8

Spécialisation de composants

Sommaire

8.1 Scénarios de spécialisation **189**
8.1.1 Description des scénarios 189
8.1.2 Définition des scénarios 193
8.2 Production de composants spécialisables **195**
8.2.1 Des composants aux générateurs de composants 196
8.2.2 Négociation entre les générateurs de composants 198
8.2.3 Spécialisation de composants 201
8.3 Bilan **203**

L'intérêt du développement à base de composants est, principalement, la possibilité de réutiliser du logiciel conçu sur la base d'un contexte générique sans oublier le fait que *maximiser la réutilisation minimise l'utilisation* [Szy02]. Cependant, la dose de généricité prise en compte lors de la construction des composants est la cause principale de l'inefficacité de l'application résultante, qui est obligée à conserver des fonctionnalités que ne seront jamais utilisée. Ce chapitre est consacré à la spécialisation de composants vis-à-vis de leur contexte d'utilisation en conservant la notion de boîte noire intrinsèque à l'approche de développement spécifique aux composants.

8.1 Scénarios de spécialisation

Sur la base des *scénarios de spécialisation* proposés par Le Meur *et al.* [LMLC04] (voir section 4.2) , nous étendons la définition du modèle de composant introduit dans la section 7 afin de pouvoir spécifier les opportunités de spécialisation des services fournis par les composants.

8.1.1 Description des scénarios

Considérons, par exemple, l'implémentation du composant *Multiplier* de la figure 7.1(i). On observe que le paramètre `x` du service `multiplier` est utilisé dans le test de l'expres-

sion conditionnelle ainsi que dans la boucle implémentant une solution récursive pour la multiplication. Donc, si la valeur de `x` est connue à la spécialisation alors il est possible d'évaluer partiellement tel service. Un raisonnement similaire peut être appliqué sur le paramètre `y`.

Dans ce contexte, il est recommandable que le producteur du composant expose telles opportunités de spécialisation afin de fournir une version plus efficace et adapté que la version originale du composant aux consommateurs éventuels. Le producteur décrit les opportunités de spécialisation sous la forme de *scénarios de spécialisation* en annotant les services du composant avec l'information de spécialisation selon les techniques de spécialisation mentionnée. Par exemple, le producteur peut définir un scénario en annotant le paramètres avec le temps de liaison, soit $\mathcal{S}$ pour ceux considérés statiques, soit $\mathcal{D}$ pour ceux considérés dynamiques. Un scénario peut être vu comme un ensemble de contraintes qui établissent de dépendances entre les paramètres d'entrée et de sorties des services de composants. Par rapport à l'exemple du composant *Multiplier*, il est pertinent spécifier le scénario $\mathcal{D}$ `multiply(`$\mathcal{D}$ `x,`$\mathcal{S}$ `y)` lequel indique qu'il existe une opportunité de spécialisation quand les paramètres du service `multiply` sont statique et dynamique, respectivement, tandis que la valeur de retour est dynamique.

Notez que les opportunités identifiées aurons toujours une forte dépendance de l'implémentation visée et, donc, ils ne peuvent pas être définie sans avoir tenu en compte, préalablement, l'implémentation correspondante. En plus, puisque les ports d'interfaces sont la seule information mise à disposition pour le consommateur au moment de l'assemblage, les scénarios sont uniquement basés sur ces ports. En effet, l'exposition d'information concernant à une implémentation déterminée attente contre la notion de boite noire laquelle nous cherchons conserver.

Dans le cadre de l'approche proposée ici, le producteur étend la description des composants avec les scénarios mentionnés auparavant. Le producteur regroupe tous les scénarios d'un service déterminé sous la forme d'une *description de spécialisation* (*i.e.*, similaire aux *modules de spécialisation* dans le contexte du travail de Le Meur *et al.* [LMLC04]). Car lors de la définition d'un composant le producteur ignore l'univers de contextes d'utilisation du composant, une solution possible est d'inclure la liste exhaustive de scénarios de spécialisation où toutes les combinaisons des annotations sur les paramètres sont considérées. Par exemple, la figure 8.1 présente la description de spécialisation `MultiplierS` lequel inclus tous les scénarios possibles définit sur le port d'interface `p_mult` de type `MultiplierI`. L'idée de base est de générer une version spécialisée du service pour chaque scénario. Notez que cette solution ne s'avère pas réaliste en vertu du temps dépensé dans la construction et la taille de l'implémentation résiduelle résultante.

Afin d'éviter ces inconvénients, le producteur peut réduire cette liste en considérant leur consistance et bénéfice vis-à-vis les opportunités de spécialisation concernées. Primairement, le scénario `(2)` de la figure 8.1(a) est considéré inconsistant car, de l'implémentation, on observe l'impossibilité du service `multiply` de retourner une valeur statique quand, au moins, un des paramètres d'entrée est dynamique. Un raisonnement similaire peut être applique sur les scénarios `(3)` et `(4)`. Deuxièmement, le scénario `(5)`, lui aussi, peut être éliminé mais dans ce cas parce qu'il s'avère redondent par rapport au scénario `(1)` car il serait possible de rendre dynamique (*i.e.*, *lifting*) la valeur de retour de ce dernier.

(a) Liste d'exhaustive.

```
specialization MultiplierS
  specializes MultiplierI {
  int multiply(int x, int y) {
   scenario S multiply(S x, S y); //(1)
   scenario S multiply(S x, D y); //(2)
   scenario S multiply(D x, S y); //(3)
   scenario S multiply(D x, D y); //(4)
   scenario D multiply(S x, S y); //(5)
   scenario D multiply(S x, D y); //(6)
   scenario D multiply(D x, S y); //(7)
   scenario D multiply(D x, D y); //(8)
  }
 }
```

(b) Liste réduite.

```
specialization MultiplierS
  specializes MultiplierI {
  int multiply(int x, int y) {
   scenario S multiply(S x, S y); //(1)

   scenario D multiply(S x, D y); //(6)

   scenario D multiply(D x, S y); //(7)
  }
 }
```

FIG. 8.1 – Description de spécialisation `MultiplierS`.

Troisièmement, car il n'apporte aucune opportunité de spécialisation le scénario `(8)` est aussi exclu de la description. Finalement, le producteur considère une description comme celle montrée dans la figure 8.1(b) où uniquement les scénarios `(1)`, `(6)` et `(8)` sont tenus en compte.

8.1.1.1 Hypothèses

En ignorant l'implémentation de service `add`, le producteur du composant *Multiplier* n'est pas au courant des opportunités de spécialisation associées, éventuellement, à tel service. Par contre, cette information pourrait être hautement profitable au moment de la définition des scénarios afin de maximiser la spécialisation dans l'implémentation du service `multiply`. Par exemple, dans le code de la classe `MultiplierImp` il est évident l'intérêt sur le temps de liaison du service `add` pour spécialiser la boucle, ici, implémentée par la méthode `loop_add`. En conséquence, même s'il n'est pas possible de raisonner en termes de opportunités de spécialisation concrètes sur les services requis, le producteur est toujours capable de faire d'hypothèses. Par exemple, le producteur peut définir un scénario $\mathcal{S}$ `multiply(`$\mathcal{S}$ `x,`$\mathcal{S}$ `y)` en présupposant l'existence du scénario $\mathcal{S}$ `add(`$\mathcal{S}$ `x,`$\mathcal{S}$ `y)`. Notez que cette hypothèse prend en compte uniquement la valeur de retour du service `add` car les paramètres d'entrée peuvent être calculés par la propagation des annotations correspondantes aux paramètres d'entrée du service `multiply`. Le problème d'imposer une hypothèse comme cela est qu'elle peut empêcher l'application d'un tel scénario si les paramètres d'entre du service fourni `multiply` deviennent dynamiques. Autrement dit, aucune type de spécialisation a lieu pour le scénario $\mathcal{S}$ `multiply(`$\mathcal{S}$ `x,`$\mathcal{S}$ `y)` si le scénario $\mathcal{S}$ `add(`$\mathcal{S}$ `x,`$\mathcal{S}$ `y)` n,'est pas disponible, même pas la spécialisation des constructions incluant les paramètres `x` et `y` du service `multiply`. Cependant, afin d'éviter ce problème, le producteur peut étendre l'ensemble de scénarios en ajoutant autres scénarios dont les hypothèses sont relaxées. La figure 8.2 montre les scénarios de spécialisation dont leur

```
specialization MultiplierS
  specializes MultiplierI {
  int multiply(int x, int y) {
   scenario S multiply(S x, S y) in AdderI assumes { S add(S x, S y)}; //(1)

   scenario D multiply(S x, S y) in AdderI assumes { D add(S x, S y)}; //(1')

   scenario D multiply(S x, D y) in AdderI assumes { D add(S x, D y)}; //(6)

   scenario D multiply(D x, S y) in AdderI assumes { D add(D x, S y)}; //(7)
  }
 }
```

FIG. 8.2 – Description de spécialisation `MultiplierS` avec d'hypothèses.

définition inclut l'hypothèse sur le service `add`. Le scénario `(1')` peut se voir comme une version relaxée du scénario `(1)`. En comparaison à `(1)`, la valeur de retour du service `mulitply` dans le scénario `(1')` est relaxée vers une valeur dynamique à cause de la valeur de retour du service `add`. Dans le cas des scénarios `(6)` et `(7)`, aucun bénéfice peut être obtenu sur l'hypothèse de une valeur de retour statique du service `add` et, comme un de deux paramètres est dynamique, l'appel au service `add` est résidualisé.

8.1.1.2 Découpage sur les composants

Parfois, il arrive que, lors de l'assemblage, le composant fournisse plus de services que les nécessaires. L'implémentation d'un composant peut utiliser partiellement l'ensemble de services fournis à travers une interface de port d'un port requis. Dans ce sa, les services qui ne sont pas utilisés (ou partiellement utilisés) peuvent être éliminés (découpé) de la composition résultante. Cette information peut être aussi exprimée sous la forme d'un scénario de spécialisation. Pour cette raison, nous introduisons l'annotation $\mathcal{K}$ (de *Kill*) pour indiquer que la construction n'est *ni statique ni dynamique*. Cette annotation permet de spécifier le critère de découpage en indiquant quelle partie de la valeur de retour d'un service n'est pas utilisée.

Rappelons nous que, dans le modèle *MoSCo*, la définition des services d'interfaces de port est fait uniquement en utilisant les types primitifs permis par le langage d'implémentation associé EFJ (*i.e.*, les paramètres et la valeur de retour de type `int` ou `boolean`). En conséquence, à l'état actuel du modèle, le découpage est possible uniquement par l'élimination d'un service de forme complète. Autrement dit, si la valeur de retour est annotée comme $\mathcal{K}$ et, en sachant que dans le contexte fonctionnel associé au langage de implémentation choisi (*i.e.*, absence d'effets de bord), la valeur de retour dépend impertinemment des paramètres d'entré alors la seule possibilité est d'annoter les paramètres d'entrée aussi comme $\mathcal{K}$.

Afin d'illustrer le découpage de manière partielle, c'est-à-dire à l'intérieur de

(a) Implémentation.

```
...
DivResult divide_o(int x, int y) {
 ...
 if (y != 0) {
  error = false;
  if (x >= 0) {
   if (y > 0) {
    quotient = 0;
    while (x >= y) {
     quotient = this.add(quotient, 1);
     x = this.subtract(x, y);
    }
    remainder = x;
   } else { ... }
  } else { ... }
 } else error = true;
 return new DivResult(quotient,
                      remainder,
                      error);
}
```

(b) Definition classe `DivResult`.

```
class DivResult {
 int quotient; int remainder, boolean error;
}
```

(c) Description de spécialisation `MultiplierS_o`.

```
specialization MultiplierS_o
  specializes ComputationUnitI {
  DivResult divide_o(int x, int y) {
   scenario (K quotient, D remainder, D error)
    divide(D x, D y);
  }
 }
```

FIG. 8.3 – Découpage du service `divide_o`.

l'implémentation d'un service, considérons l'implémentation du service `divide_o` du composant `ComputationUnit` de la figure 8.3(a). À cause du manque d'expressivité du langage EFJ, le service `divide_o` est implémenté en Java. La valeur de retour est un objet de type `DivResult` (voir figure 8.3(b)) qui encapsule le quotient de la division (`quotient`), le reste (`remainder`) et une valeur booléenne (`error`) qui indique si la division a eu lieu. Maintenant, considérons le contexte d'utilisation où uniquement le reste est utile. Dans ce cas, il est possible de spécifier un scénario de spécialisation où le quotient et la valeur d'erreur sont annotés $\mathcal{K}$ et le quotient est annoté $\mathcal{S}$ ou $\mathcal{D}$ (voir figure 8.3(c)). Cette dernière valeur de retour doit être annotée selon l'annotation des paramètres d'entrée (voir section 8.2.1.1). En conséquence, l'application du scénario `ComputationUnitS_o` résulte en la résidualisation de toutes les constructions intervenant dans le calcul du quotient. Notez que la description du scénario `ComputationUnitS_o` fait référence à la structure structure interne de l'objet retourné.

8.1.2 Définition des scénarios

À la définition initiale d'un composant donnée par (Dc, P, D_D) (voir section 7.3.2) nous ajoutons la définition des scénarios de spécialisation dénotée ici par Ss. La figure 8.4 montre la grammaire associée aux scénarios de spécialisation qui étend le langage $MoSCo-CDL$ de la figure 7.10.

Une description de spécialisation SD permet aussi de spécifier le temps de liaison des attributs qui représentent l'état du composant à l'aide de la clause `where`. Dans le cas

$$
\begin{array}{llll}
SD & \in & ScenarioDescription & ::= \texttt{specialization}\ sdId \\
 & & & \texttt{specializes}\ cId\ \{\overline{Sc}\} \\
Sc & \in & ScenarioClause & ::= [\texttt{where}\ \{\overline{At_{ann}}\}]\ \texttt{do}\ \{\overline{Ss}\} \\
Ss & \in & SpecializationScenario & ::= \texttt{for}\ pId.sId\ \texttt{scenario}\ Sv_{ann}\ [\texttt{assumes}\ \{\overline{Sc}\}]; \\
At_{ann} & \in & AnnotatedAttribute & ::= BT\texttt{(}aId\texttt{);} \\
Sv_{ann} & \in & AnnotatedService & ::= BT\texttt{(}p\texttt{)}\ sId\texttt{(}\overline{BT\texttt{(}p\ x\texttt{)}}\texttt{);} \\
BT & \in & BindingTime & ::= \texttt{S} \mid \texttt{D} \mid \texttt{K}
\end{array}
$$

FIG. 8.4 – *MoSCo*-CDL : scénarios de spécialisation.

des hypothèses, les conditions sur le temps de liaison des services requis sont imposées par la clause `assumes` qui associe un ensemble de clauses de scénarios Sc. Cette définition récursive permet aussi de relier un état en termes de temps de liaison aux composants qui fournissent les services requis.

8.1.2.1 Règles de conformité

La figure 8.5 montre les règles de conformité qui permettent de vérifier si la description donnée par les scénarios de spécialisation est conforme à la description du composant associé.

Ports :

$$\frac{Dc = Cp\ \overline{PI_n} \qquad SD ::= \texttt{specialization}\ scId\ \texttt{specializes}\ cId\ \{\overline{Sc_d}\} \quad \forall\ i\ [1,d]\ Sc_i ::= \texttt{where}\ \{\ \_\ \}\ \texttt{do}\ \{\texttt{for}\ pId.sId\ \texttt{scenario}\ \_\ \} \quad Po\ \in\ providedPorts(Cp) \qquad portId(Po) = pId}{wf\text{-}sc\text{-}ports(Dc, SD)}\ [\ \textbf{SC-P-PORT}^{CDL}]$$

Attributs :

$$\frac{Dc = Cp\ \overline{PI_n} \qquad SD ::= \texttt{specialization}\ sdId\ \texttt{specializes}\ cId\ \{\overline{Sc_d}\} \quad \forall\ i\ [1,d]\ Sc_i ::= \texttt{where}\ \{\overline{At_{ann}}\}\ \texttt{do}\ \{\ \_\ \} \qquad At\ \in\ attributes(Cp) \qquad atId(At) = aId \quad At_{ann} ::= BT\texttt{(}aId\texttt{);}}{wf\text{-}sc\text{-}attributs(Dc, SD)}\ [\ \textbf{SC-ATTR}^{CDL}]$$

Hypothèses :

$$\frac{Dc = Cp\ \overline{PI_n} \qquad SD ::= \texttt{specialization}\ sdId\ \texttt{specializes}\ cId\ \texttt{do}\ \{\overline{Sc_d}\} \quad \forall\ i\ [1,d]\ Sc_i ::= \texttt{for}\ pId.sId\ \texttt{scenario}\ \_\ \texttt{assumes}\ \{\overline{Sas}\} \qquad Sas ::= \texttt{for}\ pId.sId\ \texttt{scenario}\ \_ \quad Po\ \in\ requiredPorts(Cp) \qquad portId(Po) = pId}{wf\text{-}sc\text{-}assumptions(Dc, SD)}\ [\ \textbf{SC-R-PORT}^{CDL}]$$

Scénarios :

$$\frac{Dc = Cp\ \overline{PI_n} \qquad SD\ \text{specialization scenarios} \quad wf\text{-}sc\text{-}ports(Dc, SD) \qquad wf\text{-}sc\text{-}attributs(Dc, SD) \quad wf\text{-}sc\text{-}assumptions(Dc, SD)}{wf\text{-}scenarios(Dc, SD)}\ [\ \textbf{SCEN}^{CDL}]$$

FIG. 8.5 – MoSCo-*CDL* : Règles de conformité sur les scénarios de spécialisation.

Les scénarios de spécialisation peuvent être définis uniquement sur les services associés aux port fournis par le composant, cette condition est vérifiée par la règle **SC-P-PORT**^{CDL}.

En ce qui concerne les attributs d'un composant, la règle **SC-ATTR**^{CDL} permet de vérifier si l'ensemble complet d'attributs a été inclus dans la liste correspondante.

Comme mentionné auparavant, les hypothèses sont faites uniquement sur les annotations des services requis. Cette conformité est exprimée par la règle **SC-R-PORT**^{CDL}.

Finalement, on dit que la description de spécialisation SD est conforme à la description du composant Dc si les conditions des trois règles décrites ci-dessus sont satisfaites (**SCEN**^{CDL}).

8.2 Production de composants spécialisables

Dans cette section nous étudions l'utilisation des scénarios de spécialisation, décrits ci-dessus, pour la génération des extensions génératrices de composants, appelées ici *générateurs de composants*.

La clé de l'approche proposée dans cette thèse, qui permet la spécialisation de composants dans le cadre du modèle de boîte noire, est la livraison des composants sous la forme d'un paquet qui encapsule la description, l'implémentation de composants ainsi que les scénarios de spécialisation correspondants.

Considérons, d'abord, le cas où tous les composants sont implémentés en utilisant le même langage d'implémentation et délivrés sous la forme de code source (ou même sous la forme de code binaire de haut niveau). Dans ce contexte, nous supposons que les consommateurs de composants ont accès au même analyseur et spécialiseur. En conséquence, l'assemblage résultant peut être analysé et spécialisé comme un tout, où l'analyse est guidée par les scénarios de spécialisation donnés. L'intérêt principal dans ce cas est que le consommateur réutilise le résultat de l'analyse effectuée lors de la production du composant pour la définition et la validation des scénarios de spécialisation associés.

Le cas simple décrit ci-dessus peut être affiné, et son hypothèse affaiblie, en délivrant des composants sous la forme d'une implémentation annotée en fonction de chacun des scénarios de spécialisation. Dans ce contexte, ni l'analyseur ni le spécialiseur spécifique ne sont plus nécessaire. Mais si cette approche simplifie l'utilisation de différents langages d'implémentation, son désavantage principal concerne la génération de composants spécialisables, dont la structure devient considérablement plus lourde, en termes de taille, que celle des composants originaux. Ensuite, les composants peuvent être empaquetés sous la forme d'un générateur de composants. En effet, le générateur de composants prend comme entrée les valeurs concrètes associées aux annotations statiques exprimées dans les scénarios de spécialisation et génère finalement les composants spécialisés. Un générateur de composants comprend, en même temps, le composant original, les opportunités de spécialisation ainsi que les mécanismes qui permettent la spécialisation proprement dite.

Au lieu d'assembler et configurer les composants *per se*, le consommateur de composants assemble, en fait, des générateurs de composants et leurs valeurs de configuration (*i.e.*, valeurs statiques) correspondantes (voir section 8.2.1). Une fois le contexte d'utilisation établi, les générateurs de composants collaborent dans la construction de l'application spécialisée. En effet, cette collaboration implique deux phases : la *négociation*, pendant laquelle les générateurs négocient sur la disponibilité des services requis (voir

section 8.2.2), et la spécialisation proprement dite (voir section 8.2.3), pendant laquelle la version spécialisée de l'application est finalement créée en fonction de la négociation effectuée.

8.2.1 Des composants aux générateurs de composants

Dans cette section nous montrons comment l'analyse de programmes et leur spécialisation à l'aide d'extension génératrices présentée dans le chapitre 6 est appliquée à l'analyse et la spécialisation des composants.

8.2.1.1 Analyse de composants

Pour générer les générateurs de composants il faut d'abord analyser l'implémentation du composant en fonction des annotations apportées par les scénarios de spécialisation. L'analyse est effectuée à l'aide de l'analyseur décrit dans le chapitre 6. Dans ce contexte, les annotations initiales propagées à travers l'implémentation analysée sont le type concret et le temps de liaison des paramètres. En l'absence de sous-typage entre les types primitifs, la seule information fournie est le temps de liaison des paramètres d'entrée et de la valeur de retour comme nous l'avons montré dans la description des scénarios de spécialisation dans la section 8.1.2.

En considérant la description des générateurs de contraintes, un scénario de spécialisation contient l'annotation initiale utilisée par le générateur de contraintes pour déclancher l'analyse. En effet, les annotations des services fournis constitue l'information apportée par l'expression initiale (e_{main}). De plus, le générateur a besoin de l'information donnée par les services requis dans le cas où le scénario inclut des hypothèses.

Rappelons que, contrairement au cas où l'implémentation est entièrement accessible, dans la cadre de composants *MoSCo* l'implémentation des services requis ne sont pas disponible lors de l'analyse. Pour cette raison, le traitement des appels des services requis doit être effectué autrement que pour les appels de méthodes qui appartiennent à l'implémentation de services fournis. Le temps de liaison des valeurs passées comme paramètres sont analysées de la même manière que dans le cas d'un appel d'une méthode appartenant à une classe du programme P. Toutefois, ces valeurs sont affectées par le temps de liaison associé aux paramètres dans la définition des hypothèses. Finalement, le temps de liaison de l'appel est donné aussi par l'annotation de la valeur de retour du service requis dans les clauses de l'hypothèse correspondante.

Vérification et complétion des scénarios Dans le contexte de l'analyse étudiée dans cette thèse (voir chapitre 6) l'analyseur peut être utilisé pour la vérification et la complétion de scénarios de spécialisation. Par exemple, dans le cas du composant `Adder`, un scénario retournant une valeur statique mais prenant, au moins, une valeur d'entrée dynamique est inpossible vis-à-vis l'implémentation choisie (voir figure 7.1(d)). En conséquence, un scénario de la forme `S add(S,D)` ou `S add(D,S)` ne sera jamais associé aux services du composant `Adder`.

Cependant, il est possible de compléter des scénarios, dont la définition a été donnée de manière partielle, à l'aide de l'analyseur. Par exemple, le scénario `? add(S,S)` permet d'indiquer que la valeur de retour doit être calculée par la propagation des valeurs associées aux paramètres. Notez que les valeurs `?` et `K` sont différentes car dans le premier cas il n'existe aucune contrainte sur la valeur résultante alors que dans le deuxième cas une valeur concrète pour la valeur de retour a été spécifiée. Par exemple, dans le cas du scénario `K add(S,S)`, l'analyseur détermine que ce scénario n'est pas correct car la propagation des valeurs statiques des paramètres associe finalement une valeur de retour statique.

8.2.1.2 Génération de générateurs de composants

La génération de composants, étant une spécialisation dans le sens classique, peut être réalisée tant par interprétation que par compilation. Dans le premier cas, un générateur générique peut être utilisé pour la génération des composants spécialisés en prenant l'arbre de syntaxe abstraite annoté selon l'analyse de temps d'évaluation et les valeurs concrètes de spécialisation. Dans le deuxième cas, on génère un générateur de composants spécifique construit à l'aide d'un générateur de générateurs de composants (*component generator generator - CGG*) qui prend aussi l'arbre de syntaxe abstraite annoté comme entrée. Dans ce cas, le générateur de composant peut être compilé pour être délivré sous la forme de code binaire. La littérature sur la spécialisation de programmes considère, en général, le code source comme le résultat de la spécialisation. Cependant, il est envisageable de produire du code binaire même de code natif si l'implémentation des composants doit être cachée aux consommateurs de composants.

Ici, nous considérons l'utilisation d'un CGG, mais cette approche est aussi applicable dans le cas du générateur générique. Pour illustrer à quoi ressemble le générateur de composants, nous prenons l'implémentation du composant `Multiplier` (voir figure 7.1(i)) et les scénarios de spécialisation correspondants (voir figure 8.1(b)). L'entrée du CGG est une version annotée des services pour chaque scénario de spécialisation. Le CGG crée une méthode, appelée *générateur de services* (*service generator*) pour chaque version annotée. Cette méthode est responsable de la génération d'une version spécialisée des services en fonction des valeurs concrètes passées comme arguments. Les générateurs de services sont groupés dans une classe qui représente l'implémentation du générateur du composant correspondant. Par exemple, la classe `GenMulitplier` de la figure 8.7 est l'implémentation du générateur de composant `Multiplier`. Lors de la génération du générateur d'un composant, un ensemble d'interfaces de ports, appelées *interfaces de ports de générateur*, est aussi généré. Une interface de port de générateur est produite pour chaque interface de port incluse dans la description du composant correspondant. La figure 8.6 montre les interfaces `GenAdderI` et `GenMultiplierI` résultant respectivement de la génération des générateurs des composants `Adder` et `Multiplier`. Les méthodes `gen_add` et `gen_multiply`, appelées *sélecteurs de générateur de services* (*service generator dispatching*), sont générées automatiquement par le CGG. La classe qui implémente le générateur d'un composant et, en conséquence l'interface de port de générateur, implémente les méthodes sélectrices afin de fournir la chaîne de caractères qui représente l'appel du service en utilisant la version spécialisée de la signature. Par exemple, dans l'hypothèse d'un scénario `S add(S,S)` et

(a) Interface de générateur de port `GenAdderI`.

```
interface GenAdderI {
 String gen_add(String aScenario, Hashtable staticValues, Hashtable dynamicParams);
}
```

(b) Interface de générateur de port `GenMultiplierI`.

```
interface GenMultiplierI {
 String gen_multiply(String aScenario, Hashtable staticValues, Hashtable dynamicParams);
}
```

FIG. 8.6 – Interfaces de ports de générateur `GenAdderI` et `GenMultiplierI`.

d'un ensemble de valeurs concrètes `x=2` et `y=1`, la méthode `gen_add` produit la chaîne `add_x2_y1()`.

Chaque générateur de composants s'appuie sur les générateurs de composants des services requis pour la génération des versions spécialisées des services demandés. Par exemple, une instance du générateur `GenMultiplier` utilise une instance du générateur `GenAdder`, `gen_p_helper`, comme le montre la figure 8.7, pour générer la version correspondante du service `add` défini dans l'interface `AdderI` (voir figure 7.1(e)).

8.2.2 Négociation entre les générateurs de composants

Notons que la disponibilité des services requis ne dépend pas seulement de la signature des services mais aussi des annotations des scénarios de spécialisation. En effet, un assemblage peut être rejeté quand le service requis est disponible mais le scénario de spécialisation demandé n'a pas été considéré lors de la production du composant. Dans ce cas, il est possible de satisfaire la demande en fournissant un scénario plus petit selon une relation d'ordre résultant de l'extension de la relation d'ordre établie entre les annotations de temps de liaison (c'est-à-dire $\sqsubseteq$, où $\mathcal{S} \sqsubseteq \mathcal{D}$). Cette relation sur les scénarios s'appuie sur une règle de covariance entre les valeurs de retour et de contravariance entre les paramètres des services. De même que pour une relation de sous-typage, nous définissons la relation de substitution entre les scénarios de spécialisation S_1 et S_2, tel que $\mathrm{S}_1 \sqsubseteq \mathrm{S}_2$, comme suit :

$$\text{Soient } S_1 = bt_1\ m\texttt{(}bt_{11} \ldots bt_{1n}\texttt{)} \text{ et } S_1 = bt_2\ \texttt{m(}bt_{21} \ldots bt_{2n}\texttt{)}$$

$$S_1 \sqsubseteq S_2 \text{ si } \forall\ i,\ bt_{1i} \sqsupseteq bt_{2i}, \text{ et } bt_1 \sqsubseteq bt_2 \tag{8.1}$$

Le remplacement d'un scénario requis, c'est-à-dire un scénario spécifié par la clause `assumes` (voir section figure 8.4), par un scénario équivalent (un *sous-scénario*) a comme but de maximiser la spécialisation résultante en choisissant le scénario le plus grand entre tous les scénarios plus petits que le scénario demandé (c'est-à-dire le scénario exposant plus arguments statiques que les scénarios équivalents restants).

```
class GenMultiplier extends Generator
 implements GenMultiplierI {
  GenAdderI gen_p_helper;
  String[] listOfScenarios = ...

  GenMultiplier(GenAdderI helper){
   this.gen_p_helper = helper;
  }

  String gen_multiply(String aScenario,
   Hashtable[] staticValues,
   Hashtable[] dynamicParams){
   ...
    // case aScenario do {
    // ...
    // "D multiply(D x, S y)":
    //     return "D_multiply_Dx_3y(x)";
    //     (in the case the static value
    //     of parameter y is 3)
    // }
    ...
  }

  String S_multiply_Sx_Sy(...) {
  ...
  }
  ...
```

```
  ...
  String D_multiply_Sx_Dy(
         Hashtable[] staticValues,
         Hashtable[] dynamicParams){
   String stream;
   int x = ... // bound to the value stored
                 // in staticValues
   stream  = "int D_multiply_" + x + "x_Dy";
   stream += "(" + ... unfolding of dynamicParams ...
          + ")";
   stream += "int output = 0";
   if (x >= 0) {
     for(int i=1; i<= x; i++)
      stream += "output = spc_p_helper."
             + gen_p_helper.gen_add(
                                "D add(S x, D y)",
                                staticValues,
                                dynamicParams);
   } else {
     stream += "if (y > 0) { ..."
   }
   stream += "return output;"
   return stream;
  }
 ...
}
```

FIG. 8.7 – `GenMultiplier` : Générateur du composant `Multiplier` (version simplifiée).

Dans ce contexte, le conflit établi entre un argument statique requis et un argument dynamique fourni ou entre une valeur de retour dynamique requise et une valeur de retour statique fournie est résolu par le passage (*lifting*) des valeurs statiques vers leur représentation dans le monde dynamique. Il est possible aussi de remplacer l'annotation $\mathcal{K}$ dans la valeur de retour par une valeur $\mathcal{S}$ ou $\mathcal{D}$.

Maintenant, illustrons la négociation décrite ci-dessus en utilisant les générateurs `GenAdder` et `GenMultiplier` des composants `Adder` et `Multiplier` (voir figure 8.8).

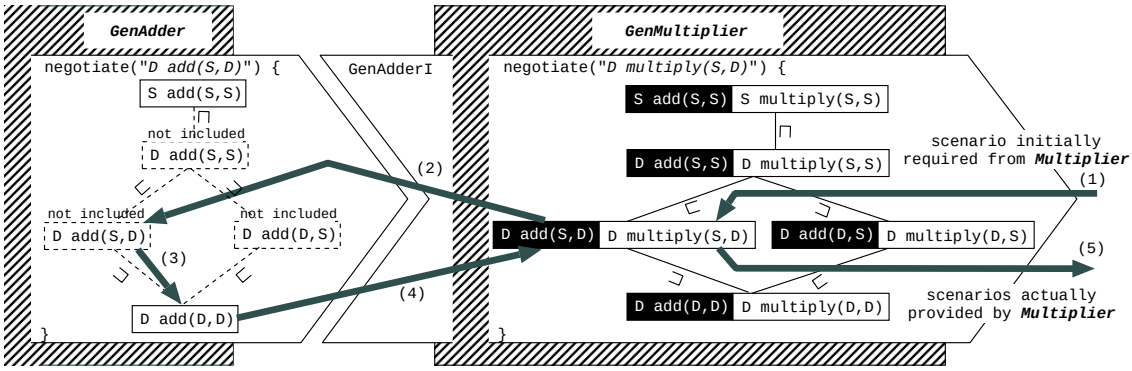


FIG. 8.8 – Négociation entre les générateurs de composants `GenAdder` et `GenMultiplier`.

Les générateurs de composants utilisent la méthode `negotiate` héritée de la classe `Generator`, qui implémente l'algorithme de substitution des scénarios expliqué ci-dessus (voir définition 8.1). Notez que les générateurs incluent chacun un scénario additionnel, `D add(D,D)` et `D multiply(D,D)`. Ce scénario, dit *identité* et inclus systématiquement lors de la génération du générateur de composant retournant, dans le pire cas, les composants génériques.

La négociation se déroule de la manière suivante : `(1)`, le composant `Multiplier` est sensé fournir le scénario `D multiply(S,D)` ; `(2)`, l'algorithme de sélection des scénarios équivalents vérifie si le scénario `D add(S,D)` est fourni par le composant `Adder` et comme aucun scénario du composant `Adder` ne satisfait ce scénario l'algorithme recherche le plus petit scénario équivalent ; `(3)`, le scénario `D add(D,D)` est choisi pour remplacer le scénario requis ; `(4)`, le temps de liaison d'argument `x` de l'appel du service `add` dans le service `multiply` est lifté ; `(5)`, le scénario requis initialement du composant `Multiplier` est finalement fourni.

8.2.2.1 Scénarios insubstituables

À cause de la relation d'ordre établie entre les scénarios, il n'existe pas de scénario de substitution pour le scénario `S add(S,S)`. En effet, un scénario plus petit est celui qui retourne une valeur statique et qui prend, au moins, un paramètre d'entrée dynamique, ce qui correspond à un scénario rejeté par l'analyseur (voir section 8.2.1.1). La question est donc de savoir quel est le résultat de la négociation entre les générateurs si le composant `Multiplier` est sensé fournir le scénario `S multiply(S,S)` alors que le scénario

(a) Implémentation.

```
class Power implements PowerI {
 int raise(int base, int exp) {
  int output = 0;
  if (exp == 0) return 1;
  if (exp == 1) return base;
  for (int i=1; i < exp, i++){
    output = output
             + multiplier.multiply(base,base);
  }
  return output;
 }
}
```

(b) Scénarios de spécialisation.

```
specialization PowerS specialize PowerI {
 int raise( int base, int exp) {
  scenario S raise (S base, S exp)
   in MultiplierI assumes S multiply(S x, S y);
...
  scenario S raise (D base, S exp)
   in MultiplierI assumes D multiply(D x, D y);
 }
}
```

FIG. 8.9 – Composant `Power`.

`S add(S,S)` requis du composant `Adder` n'est pas disponible. En effet, l'absence d'un scénario équivalent implique de revenir *en arrière* dans la négociation et de chercher une alternative au scénario `S multiply(S,S)` parmi les scénarios considérés par le composant `Multiplier`. Ainsi, nous nous trouvons dans la même situation que celle décrite ci-dessus, c'est-à-dire qu'il n'existe pas de scénario plus petit que le scénario `S multiply(S,S)`. Dans ce cas, le générateur qui fait appel au générateur `GenMultiplier` doit chercher une alternative à ses besoins.

Un point important est l'organisation de la recherche des scénarios afin d'explorer le scénario le plus statique d'abord. Quand deux scénarios, ou plus, sont considérés les plus statiques par rapport à un scénario spécifié dans une hypothèse (c'est-à-dire déclaré dans une clause `assumes`), une heuristique possible, pour maximiser la spécialisation, consiste à prendre en compte le nombre de paramètres statiques, incluant la valeur de retour. Si le nombre est le même pour les scénarios trouvés alors la sélection est effectuée selon la position du premier paramètre statique. Par exemple, si l'espace de recherche inclut les scénarios `D add(S,D)` et `D add(D,S)`, le critère sélectionne le premier scénario.

8.2.3 Spécialisation de composants

Afin d'illustrer la spécialisation de composants qui, dans notre cadre, implique l'exécution des générateurs de composants correspondants, nous considérons la traduction de nombres exprimés dans le système de numération binaire vers leur équivalent dans le système de numération décimal. En fait, ce calcul peut être effectué par des additions successives de valeurs puissances de 2. Pour cette raison, nous définissons le composant `Power` dont la description et l'implémentation sont décrites dans la figure 8.9. On considère aussi la définition d'un scénario pour le service `raise` du composant `Power` où le paramètre `base` est annoté statique.

En premier lieu, le consommateur du composant `Power` choisit un scénario de

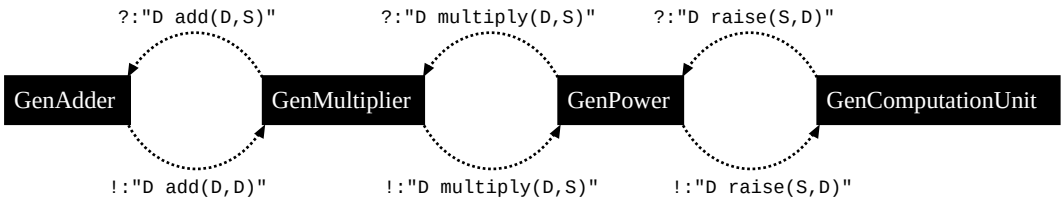


FIG. 8.10 – Négociation des scénarios pour la spécialisation de composants.

spécialisation et donne les valeurs concrètes des paramètres statiques. Pour le problème donné, le scénario est `D raise(S base, D exp)` tel que `base=2`. En second lieu, comme expliqué dans la section 8.2.2, les générateurs des composants concernés négocient afin de résoudre les problèms de disponibilité des scénarios. La figure 8.10 montre la négociation effectuée entre les générateurs `GenAdder`, `GenMultiplier` et `GenPower`. Notons qu'il n'est pas nécessaire de modifier la demande des scénarios requis car il existe toujours un scénario à fournir compatible avec celui demandé initialement. Finalement, la spécialisation des composants a lieu en générant une classe pour chaque composant selon les valeurs statiques données (voir figure 8.11).

```
class SpecAdder {
 int D_add_Dx_Dy(int x, int y) {
  return x + y;
 }
}

class SpecPower {
 SpecMultiplier specMultiplier;
 ...
 int D_raise_2base_Dexp(int exp) {
  int output = 0;
  if (exp == 0) return 1;
  if (exp == 1) return 2;
  for(int i = 1; i <= exp, i++)
   output = output
   + specMultiplier.D_multiply_2x_Dy(2);
  return output;
 }
}
```

```
class SpecMultiplier {
 SpecAdder specAdder;
 ...
 int D_multiply_2x_Dy(int y) {
  int output = 0;
  output = specAdder.D_add_Dx_Dy(output, y);
  output = specAdder.D_add_Dx_Dy(output, y);
  return output;
 }
}

class SpecComputationUnit {
 SpecAdder specAdder;
 SpecMultiplier specMultiplier;
 SpecPowed specPower;
 ...
 int D_raise_2base_Dexp(int exp) {
  return D_raise_2base_Dexp(exp);
 }
 ...
}
```

FIG. 8.11 – Spécilisation des composants `Adder`, `Multiplier` et `Power`.

8.3 Bilan

Dans ce chapitre, nous avons décrit le processus de spécialisation de composants. Dans ce contexte, les applications sont construites par l'assemblage de générateurs de composants. Un générateur, conçu comme une extension génératrice d'un composant, est créé à partir de l'implémentation du composant et des scénarios de spécialisation donnés par le constructeur du composant. Ces générateurs génèrent une version spécialisée du composant en fonction du contexte d'utilisation donné par la configuration associée à la application.

Chapitre 9

MoSCo*suite*

Sommaire

9.1 Architecture **205**
9.1.1 *ComProM* (*Component Producer Module*) 205
9.1.2 *ComCoM* (*Component Consumer Module*) 207
9.1.3 Dépôt de composants 207
9.2 Implémentation **208**
9.2.1 Conception 208
9.2.2 Points d'extension 210
9.3 Bilan **212**

Dans ce chapitre nous détaillons l'architecture et les fonctionnalités de MoSCo*suite*, un environnement de développement d'applications à base de composants conforme au modéle décrits dans le chapitre 7 et prenant en compte la définition des scénarios de spécialisation du chapitre 8.

9.1 Architecture

L'ensemble des outils fournis pour le développement d'applications à base de composants *MoSCo* forme l'environnement appelé MoSCo*suite* dont l'architecture est donnée figure 9.1. Cette architecture est composée de quatre éléments : le module *ComProM* (*Component Producer Module*) , le module *ComCoM* (*Component Consumer Module*), l'interface de l'utilisateur *GUI* (*Graphique User Interface*) et le dépôt de composants *CoRe* (*Component Repository*).

9.1.1 *ComProM* (*Component Producer Module*)

Le module *ComProM* fournit toutes les fonctionnalités concernant la production de composants, à savoir vérification de la cohérence de la définition des composants, analyse de l'implémentation par rapport aux scénarios de spécialisation, génération des générateurs de composants et déploiement des composants. Chacune de ces fonctionnalités est implémenteé par différents sous-modules comme le montre la figure 9.2.

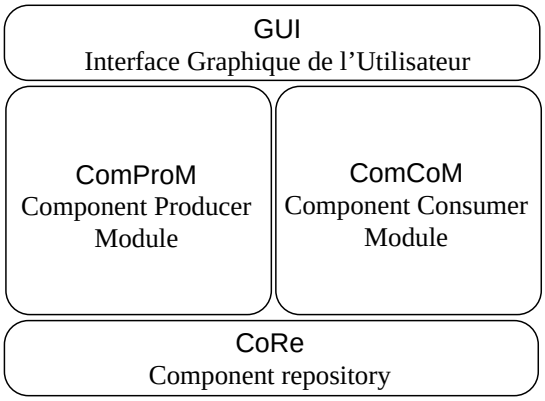


FIG. 9.1 – Architecture de MoSCo*suite*.

Le sous-module *checker* sert à vérifier si la description donnée pour un composant est conforme à son implémentation. Concrètement, l'implémentation de ce sous-module se base sur les règles de conformité présentées dans la section 7.12. En ce qui concerne la vérification de l'implémentation, étant donné un ensemble de classes Java, celle-ci est faite à l'aide des bibliothèques fournies pour la plate-forme d'implémentation utilisée.

Une fois vérifiée la description d'un composant, il est possible d'analyser son implémentation par rapport aux scénarios de spécialisation donnés. L'analyse est effectuée par le sous-module *analyzer*. Celui-ci génère les contraintes associées à chaque analyse (voir sections 6.1, 6.2 et 6.3). Ces contraintes sont traitées ensuite par le solveur REQS afin d'annoter le programme analysé. Le sous-module *compGen* prend l'implémentation

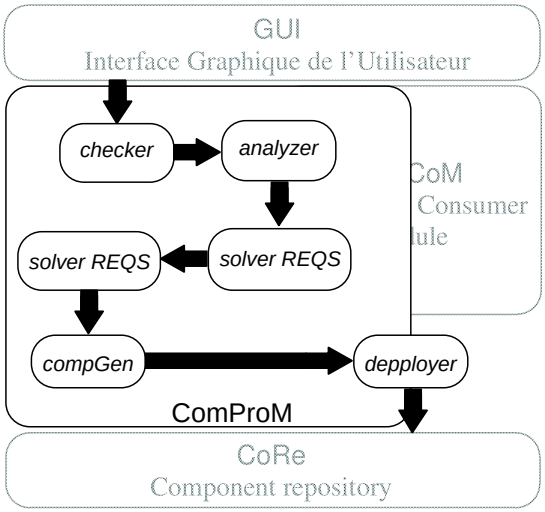


FIG. 9.2 – Module $ComProM$.

avec le résultat de l'analyse de temps d'évaluation et génère les extensions génératrices correspondantes.

Finalement, les extensions génératrices sont packagées sous la forme de *générateurs de composants* et les définitions initiales des composants sont remplacées par des descriptions de déploiement (voir section 7.1.3).

9.1.2 *ComCoM* (*Component Consumer Module*)

Le module *ComCoM*, quant à lui fournit les fonctionnalités liées à la consommation des composants, à savoir la sélection et l'assemblage de composants stockés dans le depôt de composants. Rappelons que dans le modèle *MoSCo* les composants sont en réalité des extensions génératrices produisant de composants spécialisés par rapport à des valeurs concrètes. Par conséquent, ce module permet aussi de produire les composants spécialisés. La figure 9.3 montre les sous-modules qui forment l'architecture du module *ComCoM*.

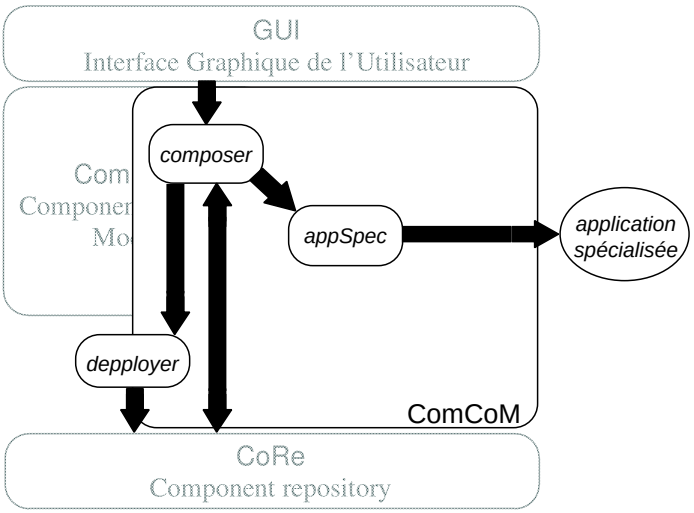


FIG. 9.3 – Module *ComCoM*.

Le sous-module *composer* permet aux consommateurs d'effectuer l'assemblage de composants. Une fois l'assemblage défini, le consommateur peut déployer le composant résultant afin de le rendre disponible pour des réutilisations ultérieures à l'aide du sous-module *depployer* présenté dans la section précédente. Dans le cas où le composant spécialisé doit être généré, le sous-module *appSpec* crée les instances des générateurs de composants correspondants selon le contexte donné (valeurs concrètes).

9.1.3 Dépôt de composants

Les paquets contenants les composants (générateurs), primitifs ou structurés, sont stockés et organisés dans un dépôt de composants (voir figure 9.4). Les sites de stockage sont gérés par le sous-module *tracker* à l'aide d'un registre d'emplacements (répertoires) associé à chaque installation de l'environnement. L'organisation est assurée par le sous-module *indexer* qui, en accédant à l'information disponible dans le descripteur, indexe et classifie les composants trouvés dans les sites donnés. Ce sous-module garde et rend aussi

disponible l'information sur les interfaces de ports, ports requis et ports fournis, définies par les composants stockés. Cela permet aux producteurs de reprendre la définition des interfaces existantes afin de diminuer l'utilisation d'*adaptateurs* (voir section 11.2.2).

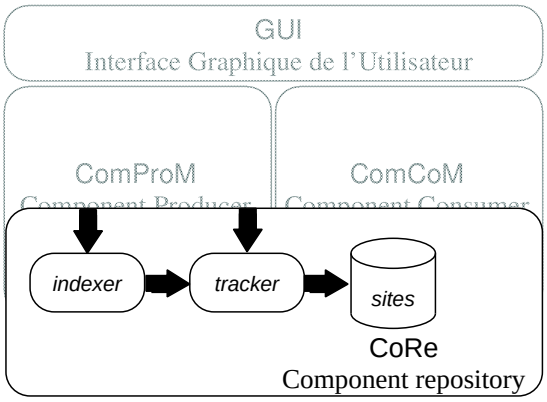


FIG. 9.4 – Module *CoRe*.

9.2 Implémentation

L'environnement MoSCo*suite* a été entièrement développé en Java sous la forme d'une infrastructure constitué d'un ensemble de plugins étendant la plate-forme Eclipse [Ecl]. En effet, chacun des modules constituant l'architecture décrite ci-dessus est implémenté par un plugin différent.

9.2.1 Conception

Tout au long du développement de l'infrastructure de MoSCo*suite* nous avons été confronté à des modifications structurelles du modèle d'analyse et de spécialisation, principalement en ce qui concerne la génération de contraintes.

Pour faire face à cette problématique, MoSCo*suite* a été développé sur la base d'une infrastructure permettant une définition flexible de l'analyse de code source Java. Dans ce contexte, et en cohérence avec notre définition des analyses (et du spécialiseur), l'analyse d'un programme est composée de différentes sous-analyses. Concrètement, l'infrastructure développée permet d'ajouter des générateurs de contraintes de manière indépendante (voir section 9.2.2).

9.2.1.1 Infrastructure

L'implémentation de MoSCo*suite* est formée par un ensemble de plugins Eclipse. La figure 9.5 montre le socle de l'infrastructure (composants dénotés par une boîte grise) et le composant qui contient l'implémentation de l'analyse de type concret (`mosco.ct.analyser`, voir section 9.2.2.1). Dans les sections suivantes nous abordons la méthode utilisée pour l'extension du socle pour la définition de différents analyseurs et générateurs de code.

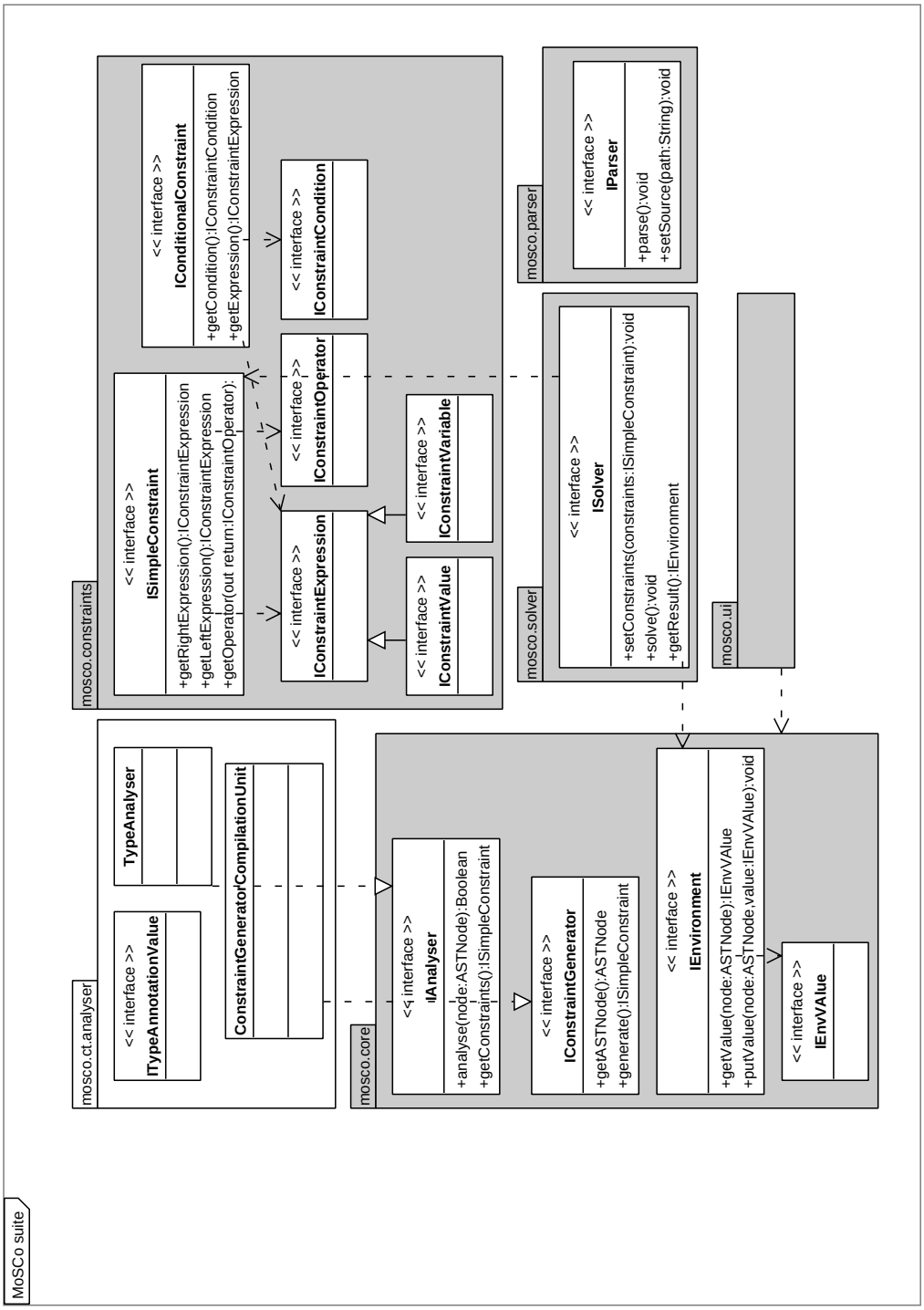


FIG. 9.5 – Infrastructure de MoSCo*suite*.

9.2.2 Points d'extension

Eclipse est une plate-forme dans laquelle toutes les fonctionnalités sont entièrement fournies par des plugins. Eclipse peut être étendu en rajoutant de nouveaux plugins qui réutilisent des fonctionnalités fournies par les plugins initialement disponibles. Dans cette plate-forme, le mécanisme pour la définition des fonctionnalités réutilisables (et extensibles) d'un plugin s'appel *point d'extension* (*extension point*). Concrètement, un plugin peut fournir de nouvelles fonctionnalités à la plateforme Eclipse tout en ajoutant des nouvelles ressources pour tout point d'extension.

Les plugins formant le socle de MoSCo*suite* fournissent des points d'extensions pour la définitions des analyseurs, des solveurs de contraintes ainsi que des générateurs de code. Dans cette section nous expliquons l'utilisation des points d'extensions pour la définition de l'évaluateur partiel décrit dans le chapitre 6.

9.2.2.1 Définition d'un analyseur

Comme l'illustre la figure 9.5, un analyseur est basé sur un visiteur qui parcourt la structure du programme (*i.e.*, AST) et des générateurs, qui dans ce cas sont des générateurs de contraintes, pour chacun des types de constructions d'un programme.

La figure 9.6 montre l'utilisation du point d'extension `analyser` pour la définition de l'analyseur de type concret.

L'attribut `ASTNodeClass` corresponde au nom de la classe du nœud de l'AST donné par le package *org.eclipse.jdt.core.dom* qui est disponible dans la plate-forme Eclipse et utilisé dans l'implémentation du visiteur. Dans la figure 9.6, le générateur de contraintes `CGMethodDeclaration` (*i.e.*, C^{ct}_{Mthd} décrit dans la section 6.1.2.2), associé à l'analyseur `ct(analyseur)`, permet la génération des contraintes correspondantes (voir figure 9.7) quand un nœud de la classe *org.eclipse.jdt.core.dom.MethodDeclaration* a été trouvé (visité) dans le programme analysé.

Ce mécanisme est utilisé aussi pour la définition d'un analyseur qui permet de vérifier que le programme analysé est conforme à une grammaire donnée (*i.e.*, un programme EFJ).

9.2.2.2 Solveurs de contraintes

L'infrastructure permet la définition d'un adaptateur qui permet l'utilisation des solveurs de contraintes. Comme mentionné dans les chapitres précédentes, nous utilisons le solveur REQS pour la résolution des systèmes de contraintes décrits dans le chapitre 6.

Le point d'extension `solver` permet, au travers de l'attribut `class`, de définir un adaptateur spécifique au solveur de contraintes utilisé

La figure 9.8 présente la définition de l'adaptateur qui permet l'utilisation de REQS.

9.2.2.3 Générateur de code source

En ce qui concerne la génération du code, il est possible de spécifier un générateur en utilisant la même structure que les analyseurs (voir section 9.2.2.1). Le programme

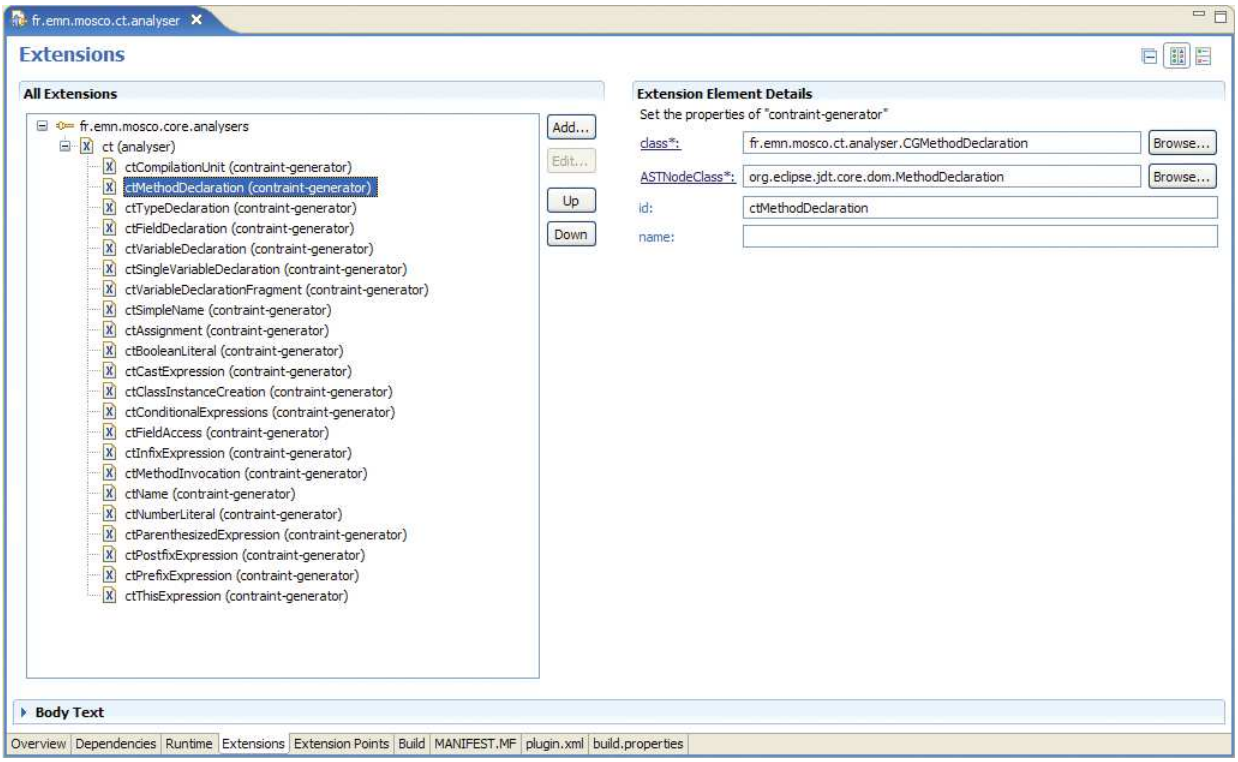


FIG. 9.6 – Analyseur de type concret dans MoSCo*suite*.

```
package fr.emn.mosco.ct.analyser.impl;

import java.util.List;

import org.eclipse.jdt.core.dom.MethodDeclaration;

import fr.emn.mosco.constraints.api.ISimpleConstraint;
import fr.emn.mosco.core.api.ConstraintGeneratorSupport;
import fr.emn.mosco.core.api.IConstraintGenerator;

public class CGMethodDeclaration extends ConstraintGeneratorSupport
                                    implements IConstraintGenerator {

    public List<ISimpleConstraint> generate() {
        MethodDeclaration m = (MethodDeclaration)this.getASTNode();
        List<ISimpleConstraint> consts = ...

        consts.add(new SimpleConstraint(
                new CTVariable(m),
                new InclusionOperator(),
                new CTVariable(m.getBody())));

        return consts;
   }
}
```

FIG. 9.7 – Générateur de contraintes `CGMethodDeclaration`.

annoté est parcouru à l'aide d'un visiteur qui génère le code en utilisant le générateur correspondant au type de construction visité.

9.3 Bilan

Dans cette section nous avons décrit l'architecture de l'outil qui permet la spécialisation de composants. Un prototype a été partiellement développé sous la plate-forme Eclipse. En particulier, ce prototype se base sur le concept de point d'extension fourni par la plate-forme. Cela permet d'ajouter des fonctionnalités de manière incrémentale. Ainsi, il n'est, par exemple, pas nécessaire d'avoir la définition complète d'un analyseur pour pouvoir tester la génération de contraintes. Cette caractéristique permet aussi d'inclure des modifications dans la définition des différents analyseurs sans affecter des autres parties du système.

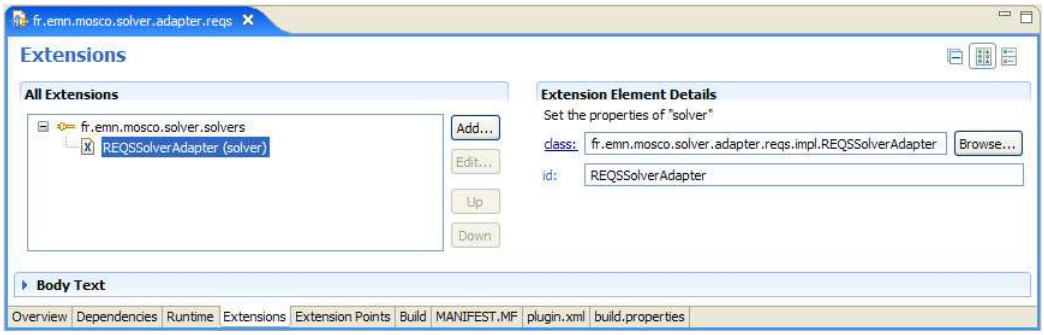


FIG. 9.8 – Adaptateur du solveur REQS dans MoSCo*suite*.

Troisième partie

Conclusions

Chapitre 10

Conclusions

Sommaire

10.1 Analyse et spécialisation dans les langages à objets **217**
10.2 Modèle minimal de composants *MoSCo* **218**
10.3 Spécialisation de composants **218**

Cette thèse propose une solution à la généricité des applications à base de composants, généricité intrinsèque aux définitions des composants. Notre approche se base sur la définition d'un évaluateur partiel dans le cadre des langages à objets et la définition d'un modèle minimal de composants. Finalement, la solution proposée s'appuie sur la théorie des extensions génératrices pour la génération de générateurs de composants qui génèrent les composants spécialisés selon un contexte d'utilisation donné.

10.1 Analyse et spécialisation dans les langages à objets

Nous avons défini un évaluateur partiel hors ligne pour des programmes écrits dans le langage à objet EFJ. Pour des raisons de précision nous avons découpé la phase d'analyse en trois étapes. Premièrement, l'analyse de type concret qui permet de calculer une approximation des types concrets associés aux constructions du programme. Cela améliore la précision de l'analyse car il est possible de réduire l'imprécision sur le flot de contrôle imposé par la combinaison du mécanisme de polymorphisme, qui implique la liaison tardive du type concret du receveur, et l'héritage, deux caractéristiques incontournables dans les langages à objets. Deuxièmement, sur la base de l'information collectée dans l'étape précédente nous calculons ensuite le temps de liaison des constructions du programme. Cette information est utilisée lors de la troisième étape pour déterminer le temps d'évaluation des constructions du programme, qui permet de savoir quelles sont les constructions qui doivent réellement être résidualisées parmi les constructions qui peuvent être spécialisées. Toutes les analyses sont définies de manière systématique sous la forme d'analyses à base de contraintes. En fait, toute la définition a comme point de départ les règles de type déclaré du langage EFJ, lesquelles sont réécrites pour exprimer la relation des annotations générées dans chaque étape. Ce travail traite de manière complète le

processus d'analyse de programmes à la base de la spécialisation de programmes.

Sur la base des annotations obtenues à l'aide des analyses décrites ci-dessus, nous avons défini un générateur d'extensions génératrices. À partir d'un programme annoté, ce générateur génère un programme dont les paramètres représentent les valeurs statiques. Ainsi, étant donné un ensemble de valeurs statiques, le résultat de l'exécution de ce programme génère un programme résiduel dont les paramètres représentent seulement les valeurs dynamiques. Cette technique permet de coder les actions concernant la spécialisation et le programme cible de ces actions dans une même entité, c'est-à-dire le générateur. Nous avons appliqué cette approche à la spécialisation de composants pouvant encapsuler la définition du composant original et ses versions spécialisées dans une même entité.

10.2 Modèle minimal de composants *MoSCo*

Le modèle de composants *MoSCo* a été conçu pour illustrer le processus de spécialisation de composants proposé dans la présente thèse. Ce modèle a comme but la construction d'applications à base de composants de manière simple par assemblage de composants. Les composants sont définis par une interface et l'implémentation correspondante fournis par le producteur des composants tandis que le consommateur n'a accès qu'à l'information représentée par l'interface lors de l'assemblage des composants. Le but de ce modèle est aussi de rendre explicite le mécanisme de composition des composants, où la composition est établie par la mise en relation des services requis d'un composant avec les services fournis d'autres composants.

10.3 Spécialisation de composants

Nous avons abordé ce travail en nous interrogant sur l'impact de l'excès de généricité des composants sur les applications résultant de l'assemblage de tels composants. Il n'est pas nécessaire de faire une étude sophistiquée sur les désavantages concernant les aspects de performance, usage de mémoire,... de ces applications par rapport à celles où cette généricité a été réduite en adaptant les composants à un contexte d'utilisation concret.

Pour cette raison, nous avons étendu le modèle de composant *MoSCo* afin de fournir un mécanisme pour spécifier les opportunités de spécialisation tenue en compte par les producteurs de composants. En plus de la description originale associée à un composant, à savoir l'interface et l'implémentation, les composants peuvent aussi inclure des scénarios de spécialisation pour les services fournis. Au moyen de l'analyseur et le générateur d'extensions génératrices présentés dans ce document, la description des composants sont réécrits sous la forme de générateurs de composants. En termes généraux, le générateur d'un composant peut être vu comme un ensemble de générateurs de services pour chaque scénario de spécialisation. Dans ce cas, l'assemblage de composants pour le développement d'applications est effectué par l'assemblage de générateurs de composants, lesquels produisent les composants spécialisés en fonction des valeurs concrètes des contextes d'utilisation.

Ainsi, nous avons démontré qu'il est possible de profiter des avantages du développement d'applications à base de composants génériques et d'adapter ces dernières selon un contexte donné. Le point distinctif de notre approche par rapport à d'autres approches existantes pour l'adaptation de composants est la possibilité d'adaptation de l'implémentation des composants. Cette adaptation est effectuée tout en respectant le modèle de réutilisation boîte noire. Autrement dit, ici la séparation des rôles entre de producteur et de consommateur est aussi respectée, même si le consommateur obtient des composants dont l'implémentation se voit modifiée, sans accéder directement à cette implémentation.

Finalement, afin de tester les idées présentées tout au long de ce document, nous avons développé l'environnement MoSCo*suite* pour la construction d'applications à base de composants du modèle *MoSCo*.

Chapitre 11

Perspectives

Sommaire

11.1 Analyse et spécialisation dans les langages à objets **221**
11.1.1 Preuve formelle de la correction de la phase d'analyse 221
11.1.2 Langage cible de la spécialisation 221
11.1.3 Résolution intégrée de contraintes 223
11.1.4 Sensibilité des analyses 223
11.2 Modèle minimal de composants *MoSCo* **223**
11.2.1 MoSCo-*CDL* 223
11.2.2 Adaptateurs de composants 224
11.3 Spécialisation de composants **225**
11.3.1 Quantification de la spécialisation des services 225
11.3.2 Fusion de composants 225
11.3.3 Applications réalistes 225

Les perspectives dégagées par notre travail concernent les contributions mentionnées dans le chapitre 10.

11.1 Analyse et spécialisation dans les langages à objets

11.1.1 Preuve formelle de la correction de la phase d'analyse

Bien que nos analyses aient été construites de manière raisonnée et soient décrites de manière formelle, leur correction reste à prouver formellement. En l'absence d'une telle preuve, il reste difficile de se convaincre conceptuellement de la correction de la spécialisation. Ce besoin se renforcerait encore dans le cas d'utilisation d'analyses plus sophistiquées.

11.1.2 Langage cible de la spécialisation

Affectations. Parmi les limitations qui font du langage EFJ un cadre idéal pour la formalisation et l'étude des propriétés des langages à objets, celle qui est la plus importante

est l'absence des affectations dans les constructions valides. En pratique, l'inclusion des affectations dans le cadre de notre travail impliquerait des modifications dans l'analyse proposée. En suivant l'approche classique dans le traitement des affectations dans les langages impératifs, il est possible de prendre en compte cette caractéristique en ajoutant une nouvelle étape d'analyse, dite *analyse d'alias*. Étant donné un emplacement de mémoire, l'analyse d'alias permet de calculer les différentes références sur cet emplacement. Notons que l'inclusion des affectations implique aussi l'inclusion de la notion de bloc.

Toutefois, la conséquence la plus importante par rapport aux affectations est que les objets ne sont plus considérés comme des structures invariables. Cela implique qu'un nouveau niveau de précision des annotations devrait être défini car, dans ce cadre, une même instance peut voir changer son annotation en fonction du point du programme analysé. Dans ce contexte, une approche monovariante pourrait consister à continuer d'associer une seule annotation pour toutes les instances créées.

Membres de la superclasse. Entre autres, la référence aux membres de la superclasse, `super`, permet de profiter des avantages de la relation d'héritage en ce qui concerne la réutilisation de la structure et du comportement des classes étendues. Concernant l'analyse de programme, l'inclusion de la construction `super` demanderait de représenter la propagation des annotations vers les membres correspondants.

Dans un langage statiquement typé, comme Java [GJSB00], la référence `super` est liée statiquement à la classe correspondante, c'est-à-dire à celle déclarée dans la définition de la classe (*i.e.*, dans la clause `extends`). La superclasse directe d'une classe n'est pas forcement la classe dans laquelle le membre référencé est défini. Autrement dit, l'expression `super` permet de réutiliser une méthode qui se trouve dans l'espace de recherche donné par la hiérarchie des superclasses. Dans le cadre de notre approche, on considère que l'annotation sur l'expression `super` correspond à la superclasse qui contient la définition du membre référencé. Cela permet d'unifier les annotations sur le membre *racine* (*root definition*). Pour cela, il serait nécessaire d'identifier la superclasse de la définition. La figure 11.1 montre la définition de la fonction *mRootDef* qui retourne la classe dans laquelle la méthode a été initialement définie.

$$\frac{superClass(c) = d \qquad t\ m(\overline{t\ \_})\{\ldots\} \in methods(d) \qquad e\ m(\overline{e\ \_})\{\ldots\} \in methods(c) \qquad t = e \qquad \overline{t} = \overline{e}}{overridden(m, c)}$$

$$\frac{\texttt{class}\ d\ \{\_\ \_\ M_{d_1} \ldots M_{d_n}\} \qquad \texttt{class}\ c\ \texttt{extends}\ d\ \{\_\ \_\ M_{c_1} \ldots M_{c_n}\} \qquad \exists\ M_{d_i} = t\ m(\overline{t\ \_})\{\ldots\} \qquad \neg overridden(m, c)}{mRootDef(m, c) = d}$$

Fig. 11.1 – EFJ : Définition initiale des méthodes.

Rappelons, cependant, qu'il est toujours possible d'implémenter une transformation du programme, afin d'éliminer la relation d'héritage, avant d'effectuer l'analyse.

11.1.3 Résolution intégrée de contraintes

Même si les étapes de l'analyse de programmes (i.e., type concret, temps de liaison et temps d'évaluation) sont enchaînées, le calcul s'effectue de manière séparée. En effet, comme le montre la figure 6.40, les contraintes de temps de liaison, par exemple, sont générées une fois que la solution pour les contraintes de type concret a été rendue. Cependant, la figure n'exprime pas le fait que les solutions données par les solveurs de contraintes sont stockées dans des structures de données gérées et accédées par le générateur de contraintes. Pour éviter ce type de changement de contexte, une possibilité envisageable peut être la génération et la résolution des contraintes de manière intégrée. En effet, on pourrait combiner la génération de contraintes de type, qui dans le cas actuel prend en compte le type déclaré des constructions, avec la génération de contraintes de temps de liaison. Par exemple, si on prend en compte l'annotation de temps de liaison d'une expression conditionnelle, concrètement de celle de la condition, le type concret de l'expression peut varier si la condition est annotée comme statique ou dynamique. Dans le cas statique, on sait que seule une des branches sera vraiment exécutée. En l'absence de valeurs concrètes pour résoudre la condition, une possibilité est d'introduire un nouvel opérateur qui permette d'exprimer cette disjonction (exclusive), c'est-à-dire d'associer le type concret d'une branche en fonction du temps de liaison du test. Pour le moment, dans le contexte du solveur REQS, cette solution n'est pas possible.

11.1.4 Sensibilité des analyses

Pour une analyse de programme donnée, il y a souvent une tension entre précision de l'analyse d'une part et sensibilité et simplicité de cette analyse. Le choix de la simplicité, et donc d'analyses insensibles au contexte (notamment lors des appels de méthode), nous a permis d'aborder l'ensemble du processus de spécialisation. L'ajout d'une analyse de type concret, particulière aux langages à objets, est intuitivement une amélioration de la sensibilité dans le cadre d'une utilisation pratique. Toutefois, les gains apportés par cette contribution restent à vérifier dans un cadre plus réaliste que nos simples illustrations.

Cependant, il est tout à fait envisageable que la sensibilité de l'analyse puisse être améliorée tout en restant raisonnablement efficace grâce à la modularité. La modularité pourrait limiter les effets de sur-spécialisation en focalisant l'analyse sur les parties pertinentes du programme.

11.2 Modèle minimal de composants *MoSCo*

11.2.1 MoSCo-*CDL*

Délégation par ports. Il serait utile de considérer l'extension du langage de description de composants *MoSCo* pour exprimer la délégation de services fournis à l'implémentation de services requis. En pratique, il serait possible de connecter les ports fournis des sous-composants aux services fournis du composant englobant et les services requis du composant englobant aux services requis des sous-composants. Dans les deux cas, les ports du

composant englobant seraient préfixés par le mot clé `This`. Cette modification implique aussi la définition des règles de conformité correspondantes.

Initialisation de composants structurés Nous avons vu que, à cause des limitations imposées par le langage d'implémentations choisi où tous les champs d'une classe doivent être initialisés lors de l'instanciation de la classe (voir section 7.2), l'architecture d'un composant structuré devient visible aux consommateurs du composant. Pour cette raison, nous considérons la possibilité d'instancier un composant structuré à l'aide d'une méthode spécifique pour l'initialisation de tout ce qui concerne le contexte d'exécution du composant, à l'exception de l'état. Cette méthode d'initialisation (*bootstrapping*) pourrait être définie de la manière suivante :

```
class ComposantA {
 Aa1 aa1;
 ...
 Aan ann;
 ComposantB b;
 ComposantA(Aa1 aa1,..., Aan aan, Ab1 ab1,...,Abm abm) {
  this.aa1 = aa1;
  ...
  this.aan = aan;
  init(Ab1,...,Abm abm);
 }
 void init(Ab1 ab1,...,Abm abm) {
  b = new B(ab1,...,abm);
 }
 ...
}
```

Dans l'exemple, le composant `ComposantA` dont l'état est représenté par les attributs `Aa1,...,Aan`, inclut le sous-composant `ComposantB`, lequel a aussi un état représenté par les attributs `Ab1,...,Abm`. La méthode `init` prend comme paramètres les attributs des sous-composants pour l'instanciation correspondante.

11.2.2 Adaptateurs de composants

L'assemblage de composants est effectué par la liaison de ports associés à la même interface de port, c'est-à-dire à l'aide d'une sorte de typage nominal. Actuellement, l'environnement de développement MoSCo*suite* oblige les producteurs d'un composant (avec dépendances) de tenir compte des interfaces de port associées aux port fournis des composants disponibles dans les dépôts de composants au moment de spécifier les interfaces requises d'un composant. Cela implique, entre autres, une limitation importante de l'ordre de création des composants. En effet, la création d'un composant est limitée aux interfaces existantes et donc, étant donné un assemblage déterminé, il faut créer les composants selon

leur relation de dépendance. Dans ce contexte, nous pourrions envisager inclure un type de composant *adaptateur* qui permette d'adapter deux définitions d'interfaces de port. Un adapteur peut être utilisé, en principe, pour résoudre les conflits de noms entre les interfaces, non seulement au niveau de l'interface, mais aussi au niveau des services.

11.3 Spécialisation de composants

11.3.1 Quantification de la spécialisation des services

Le langage de modules proposé par Le Meur *et al.* [LMLC04] a été repris dans notre approche pour la représentation des scénarios de spécialisation appliqués tout au long de la spécialisation de composants. L'approche de Le Meur, utilisée dans le contexte de programmes C, ne considère pas la notion d'hypothèses à cause de l'absence d'un mécanisme de substitution. L'approche proposée dans cette thèse aborde la substitution de scénarios, mais de manière partielle. L'algorithme utilisé pour la recherche du scénario le plus adapté par rapport un scénario demandé, mais absent, se base sur les annotations des services fournis. Concrètement, cette approche se base sur le nombre et l'ordre des paramètres statiques des scénarios qui définit une relation d'ordre entre les scénarios. Mais cette relation d'ordre est grossière, par exemple un scénario est considéré plus petit qu'un autre si le nombre de paramètres statiques est supérieur. Il serait donc utile d'affiner cette relation d'ordre en considérant par exemple un système de quantification sur les scénarios qui puisse donner une mesure de la spécialisation résultante en fonction du nombre et du type de constructions affectées par la propagation des annotations statiques, du nombre d'hypothèses, ...

11.3.2 Fusion de composants

Le résultat de la spécialisation d'une application à base de composants conserve l'architecture originale, c'est-à-dire que chacun des composants est remplacé par le composant spécialisé correspondant où même les connections entre les ports sont conservées. Il serait intéressant de considérer la possibilité de fusionner, de manière partielle ou complète, les composants spécialisés. Il serait possible de guider la spécialisation des composants intervenant dans un assemblage en décidant quels composants (implémentation) peuvent être fusionnés, par exemple, en résidualisant le code de services requis dans la classe des services fournis (appelants). Dans le sens inverse, il s'agirait de contraindre la spécialisation à un sous-ensemble des composants d'un assemblage donné. Ainsi, un composant conservant sa définition originale (au moins l'interface) pourrait être la cible d'une reconfiguration dynamique.

11.3.3 Applications réalistes

Une notion limitée d'évaluation partielle est utilisée dans la configuration de composants dans le cadre du modèle Koala [vOvdLKM00, vO02]. Koala, un modèle de composants dédié à la définition de lignes de produits logicielles (*software product lines*)

pour la génération de composants dans l'industrie électronique. En prenant en compte les perspectives d'évolution de notre modèle mentionnées ci-dessus (par exemple, expressivité du langage d'implémentation), il pourrait être possible d'appliquer nos propositions au développement d'applications aussi réalistes. En effet, les scénarios de spécialisation peuvent se voir comme la définition d'une famille de composants à partir desquels un membre (composant spécialisé) peut être généré.

Bibliographie

[ACN02a] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In Magnusson [Mag02], pages 334–367.

[ACN02b] J. Aldrich, C. Chambers, and D. Notkin. ArchJava : Connecting software architecture to implementation. In ICSE2002 [ICS02], pages 187–197.

[ADS91] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, analysis, and verification, TAV4*, pages 60–73, Victoria, Canada, October 1991. ACM Press.

[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6) :589–616, 1993.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(1) :79–111, 1999.

[And93] L.O. Andersen. Binding-time analysis and the taming of C pointers. In PEPM93 [PEP93], pages 47–58.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[And95] P.H. Andersen. Partial evaluation applied to ray tracing. DIKU Research Report 95/2, Computer Science Department, University of Copenhagen, 1995.

[Arc] ArchJava. `http ://archjava.fluid.cs.cmu.edu/software/`.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In *ECOOP 2003 - Object-Oriented Programming, 16th European Conference*, LNCS, Darmstadt, Germany, July 2003. Springer-Verlag.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bal69] R. Balzer. EXDAMS - extandable debugging and monitoring system. In *Proceedings of the Spring Joint Computer Conference*, volume 34, pages 567–586. AFIPS Press, 1969.

[Ban] The bandera project. `http ://bandera.projects.cis.ksu.edu/`.

[Bat98] D. Batory. Product-line architecture. In *4th Conference Smalltalk und Java in Industrie und Ausbildung*, Erfurt, Germany, October 1998. Invited talk.

[BB88] M.A. Bulyonkov and G.J. Barzdin. Mixed computation as a tool for extracting compilation phases. In *Methods of Compilation and Program Construction. All-Union Conference, Novosibirsk*, pages 21–23, 1988. (In Russian).

[BC85] J.-F. Bergeretti and B. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1) :37–61, 1985.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of ECOOP-OOPSLA*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[BCC+95] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *TAPOS - Theory and Practice of Object Sytems*, 1(3) :221–242, 1995.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2 edition, 2003.

[Ber90] A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, Florida, USA, June 1994. ACM Press.

[BH93a] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging - AADEBUG'93*, pages 206–222, London, UK, 1993. Springer-Verlag.

[BH93b] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, Charleston, South Carolina, USA, January 1993. ACM Press.

[BHOS76] L. Beckman, A. Haraldson, O. Oskarson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7(4) :319–357, 1976.

[Bin98] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11) :583–594, 1998. Special issue on program slicing.

[BJ03] F. Besson and T. Jensen. Modular class analysis with datalog. In *Static Analysis, 10th International Symposium, SAS 2003*, volume 2694 of *LNCS*, pages 19–36, San Diego, CA, USA, June 2003. Springer-Verlag.

[BN02] G. Bobeff and J. Noyé. On the interaction of partial evaluation and inheritance. In *Proceedings of the First Inheritance Workshop at ECOOP 2002*, pages 16–22, Malaga, Spain, 2002.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Denmark, 1990.

[Bon91] A. Bondorf. *Similix Manual, System Version 4.0*, 1991.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Transactions on Software Engineering*, 11(5) :89–94, September 1994.

[Bul93] M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In PEPM93 [PEP93], pages 59–65.

[BW94] L. Birkedal and M. Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposium, PLILP'94*, volume 844 of *LNCS*, pages 198–214, Madrid, Spain, September 1994. Springer.

[BW96] Alan W. Brown and Kurt C. Wallnau. Engineering of component based systems. In *Component-Based Software Engineering*, pages 7–15. IEEE Computer Society, 1996.

[BW97] M. Büchi and W. Weck. A plea for grey-box components. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the First Workshop on the Foundations of Component-Based Systems*, pages 39–49, Zurich, Switzerland, September 1997. TUCS Technical Report 122.

[BW98] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Transactions on Software Engineering*, 15(5) :37–46, September 1998.

[Car02] R. Cardone. Language and compiler support for mixin programming. Master's thesis, University of Texas, Austin, USA, May 2002.

[Cas97] G. Castagna. *Object-Oriented Programming : A Unified Foundation.* Springer-Birkhauser, 1997.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *7th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, July 1995. ACM Press.

[CD90] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. Jones, editor, *ESOP'90 - Third European Symposium on Programming*, volume 432 of *LNCS*, Copenhagen, Denmark, May 1990. Springer-Verlag.

[CF94] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4) :1097–1113, 1994.

[CH01] B. Councill and G.T. Heineman. Definition of a software component and its elements. In Heineman and Councill [HC01], pages 5–19.

[Cha93] C. Chambers. Predicate classes. In O. Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming - 7th European Conference*, volume 707 of *LNCS*, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Che05] D. Chefrour. Developing component based adaptive applications in mobile environments. In *Proceedings of the 2005 ACM symposium on Applied computing, SAC'05*, pages 1146–1150, Santa Fe, New Mexico, 2005. ACM Press.

[CHL+98] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and E.N. Volanschi. Tempo : Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3), September 1998.

[CHN+96] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.-N. Volanschi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [DGT96], pages 54–72.

[CJ01] B. Chang and J. Jo. Granularity of constraint-based analysis for java. In *3rd ACM Conference on Principles and Practice of Declarative Programming*, pages 94–102, Firenze, Italy, September 2001. ACM Press.

[CMN91] J. Choi, P. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4) :491–530, October 1991.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL'96 : The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg, FL, USA, January 1996. ACM Press.

[CND+04] P. Cointe, J. Noyé, R. Douence, T. Ledoux, J-M. Menaud, G. Muller, and M. Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. *RSTI L'Objet, colloque en l'honneur de Jean-François Perrot*, 10(4), 2004.

[Cod] CodeSurfer. `http ://www.grammatech.com/products/codesurfer/`.

[Com] ComponentJ Model. `http ://www-ctp.di.fct.unl.pt/ jcs/ComponentJ/`.

[Con88] C. Consel. New insights into partial evaluation : The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming*, volume 300 of *LNCS*, pages 236–246, Nancy, France, May 1988. Springer-Verlag.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, Nice, France, 1990. ACM Press.

[Con93] C. Consel. A tour of Schism : A partial evaluation system for higher-order applicative languages. In PEPM93 [PEP93], pages 66–77.

[Cor] OMG, CORBA. `http ://www.omg.org/corba/`.

[CORS98] R. Cherinka, C. Overstreet, J. Ricci, and M. Schrank. Maintaining a COTS component-based solution using traditional static analysis techniques. *LNCS*, 1543 :165–166, 1998.

[Cou02] *The Fractal Composition Framework*, June 2002. Version 0.9.

[CSC00] J. Costa Seco and L. Caires. A basic model of typed components. In E. Bertino, editor, *Proceedings of the European Conference on Object-oriented Programming (ECOOP 2000)*, number 1850 in LNCS, pages 108–128, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[CSC02] J. Costa Seco and L. Caires. *ComponentJ in a nutshell*. Computer Science Department, Universidade Nova de Lisboa, 2002.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4) :471–522, December 1985.

[DDDCG00] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle : Dynamic object re-classification. In J. Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, number 2072 in LNCS, pages 130–149, Budapest, Hungary, June 2000. Springer-Verlag.

[DeM03] L.G. DeMichiel. *Enterprise JavaBeans^{TM} Specification*. SUN Microsystems, November 2003. Version 2.1, Final Release.

[DGS92] E. Duesterwald, R. Gupta, and M. Soffa. Rigorous data flow testing throug output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92*, pages 131–145, California, USA, 1992.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation, International Seminar, Dagstuhl Castle*, volume 1110 of *LNCS*. Springer-Verlag, February 1996.

[DHH97] D. Dussart, R. Heldal, and J. Hughes. Module-sensitive program specialization. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 206–214, Las Vega, NV, USA, May 1997. ACM SIGPLAN Notices, 32(5).

[DHM95] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications : Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *Proceedings of the Second International Symposium on Static Analysis, SAS'95*, volume 983 of *LNCS*, pages 118–135, Glasgow, UK, September 1995. Springer-Verlag.

[Ecl] Eclipse. `http ://www.eclipse.org`.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Programming Series. Microsoft Press, Redmond, WA, 1998.

[EFB01] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming : Introduction. *Communications of the ACM*, 44(10) :29–32, October 2001.

[EJ04] M. Éluard and T. Jense. Validation du contrôle d'accès dans des cartes à puce multiapplications. *Technique et science informatique*, 23(3) :323–357, 2004.

[Ers77] A.P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2) :38–41, 1977.

[Ers78] A.P. Ershov. On the essence of compilation. *Formal description of Programming Concepts*, 5(2) :21–39, 1978.

[FF98] M. Flatt and M. Felleisen. Units : Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, May 1998. ACM SIGPLAN Notices, 33(5).

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3) :319–349, 1987.

[FS02] Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correction by construction. In *On the Move to Meaningful Internet Systems - Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 995–1012, Irvine, California, USA, May 2002. Springer-Verlag.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5) :45–50, 1971.

[Gen03] General Dynamics Decision Systems. *Openwings Specification*, January 2003. Version 1.0, Final.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJ89] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[GJSB00] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition edition, 2000.

[GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8) :751–761, 1991.

[Goa82] C. Goad. Automatic construction of special purpose programs. In D.W. Loveland, editor, *6th Conference on Automated Deduction*, volume 138 of *LNCS*, pages 194–208. Springer-Verlag, New York, USA, 1982.

[Gro02] Object Management Group. CORBA components. Adopted Specification formal/02-06-65, OMG, June 2002. Version 3.0.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, December 1993. World Scientific Publishing Company.

[Gur94] C. A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Goedel*. PhD thesis, Department of Computer Science, University of Bristol, January 1994.

[Haa02] K. Haase. *Java^{TM} Message Service API Tutorial*. Sun Microsystems, November 2002. Version 1.3.

[Ham97] G. Hamilton. *JavaBeans^{TM}*. Sun Microsystems, July 1997. Version 1.01.

[HC01] G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering – Putting the Pieces Together*. Addison-Wesley, 2001.

[HDC88] J. Hwang, M. Du, and C. Chou. Finding program slices for recursive procedures. In *Proceedings of the Twelfth International Computer Software and Applications Conference, COMPSAC'88*, pages 220–227, October 1988.

[Hei92a] N. Heintze. Practical aspects of set based analysis. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 765–779, Washington D.C., USA, November 1992. MIT Press.

[Hei92b] N. Heintze. *Set-Based Program Analysis*. PhD thesis, School of Computer Science, Carneige Mellon University, 1992.

[Hei94] N. Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 306–317, Orlando, Florida, United States, June 1994. ACM Press.

[HJ94] N. Heintze and J. Jaffar. Set constraint and set-based analysis. In Alan Borning, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer, 1994.

[HM94] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Programming Languages and Systems - ESOP'94 - Fifth European Symposium on Programming*, volume 788 of *LNCS*, pages 287–301, Edinburgh, UK, April 1994. Springer-Verlag.

[HN00] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages : Flow, context and return sensitivity. *Journal of Theoretical Computer Science*, 248(1-2) :3–27, October 2000.

[HNC97] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *LNCS*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[Hor90] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90*

Conference on Programming Language Design and Implementation, pages 234–245, White Plains, New York, USA, June 1990. ACM Press. ACM SIGPLAN Notices, 25(6).

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3) :345–387, 1989.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1) :26–60, January 1990.

[Hug91] J. Hughes, editor. *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, Cambridge, Massachusetts, USA, August 1991. Springer-Verlag.

[ICS02] *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002*, Orlando, FL, USA, May 2002. ACM Press.

[Ind] The Indus Project. `http ://indus.projects.cis.ksu.edu/`.

[IPW01] A. Igarashi, B.C. Pierce, and P.C. Wadler. Featherweight Java : A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3) :396–450, May 2001.

[Jac90] H.F. Jacobsen. Speeding up the back-propagation algorithm by partial evaluation. Student Project 90-10-13, DIKU, University of Copenhagen, Denmark. (In Danish), October 1990.

[Jav] Java Layers. `http ://www.cs.utexas.edu/users/richcar/`.

[JBo] JBoss Application Server. `http ://www.jboss.org/products/jbossas`.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall, 1993.

[JL96] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In Danvy et al. [DGT96], pages 238–262.

[JOn] Java OpeN Application Server. `http ://jonas.objectweb.org/`.

[JPR99] T. Jensen, F. Ployette, and O. Ridoux. *L'Outil de résolution de système d'équations récursive REQS*. Lande project, IRISA, Rennes, 1999.

[JPR02] T. Jensen, F. Ployette, and O. Ridoux. Iteration schemes for fixed point computation. In *Proceedings of 4th Int workshop on Fixed Points in Computer Science (FICS'02)*, Copenhagen, Denmark, July 2002.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation : The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 124–140, Dijon, France, 1985. Springer-Verlag.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix : A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1) :9–50, 1989.

[Kah84] K.M. Kahn. Partial evaluation, programming methodology, and artificial intelligence. *The AI Magazine*, 5(1) :53–57, 1984.

[Kam93] M Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linköping University, 1993.

[Kav] The Kaveri Tool. `http ://indus.projects.cis.ksu.edu/projects/`.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL '81 : Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, New York, NY, USA, 1981. ACM Press.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finnland, June 1997. Springer-Verlag.

[Kom81] H. J. Komorowski. *A specification of an Abstract Prolog Machine and its application to partial evaluation*. PhD thesis, Linköping Studies in Science and Technology Dissertations, Linköping University, June 1981.

[Kom82] H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language : a theory and implementation in the case of prolog. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 255–267, New York, NY, USA, 1982. ACM Press.

[KS04] R. M. Kolonay and M. Sobolewski. Grid interactive service-oriented programming environment. In *The 11th ISPE International Conference on Concurrent Engineering : Research and Applications*, pages 97–102, China, 2004. Taylor & Francis.

[Lar00] M. Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. PhD thesis, Institutionen för informationsteknologi, Uppsala universitet, December 2000.

[Lau91] J. Launchbury. A strongly-typed self-applicable partial evaluator. In Hughes [Hug91], pages 145–164.

[LMCE02] A. Le Meur, C. Consel, and B. Escrig. An environment for building customizable software components. In *IFIP/ACM Working Conference - Component Deployment*, pages 1–14, Berlin, Germany, June 2002. Springer-Verlag.

[LMLC04] A. Le Meur, J. Lawall, and C Consel. Specialization scenarios : A pragmatic approach to declaring program specialization. *Higher-Order and Symbolic Computation*, 17 :49–92, 2004.

[Löw01] J. Löwy. *COM and .NET component services*. O'Reilly, September 2001.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11(3-4) :217–242, October 1991.

[LV95] D. C. Luckham and James V. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, 1995.

[Lyl84] J. R. Lyle. *Evaluating variations on program slicing for debugging (dataflow, ada)*. PhD thesis, University of Maryland, 1984.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[Mag02] Boris Magnusson, editor. *ECOOP 2002 - Object-Oriented Programming,17th European Conference*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer-Verlag.

[MB93] T. Mogensen and A. Bondorf. Logimix : A self-applicable partial evaluator for Prolog. In K.-K. Lau and T. Clement, editors, *LOPSTR 92. Workshops in Computing*, pages 214–227. Springer-Verlag, January 1993.

[McI68] M.D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 94–115, New Haven, CT, USA, June 1991. ACM Press. ACM SIGPLAN Notices, 26(9).

[Mey94] B. Meyer. *Reusable Software : The Base Object-Oriented Component Libraries*. Prentice-Hall, 1994.

[MFH01] S. McDirmid, M. Flatt, and W.C. Hsieh. Jiazzi : New-age components for old-fashioned Java. In *OOPSLA'01, Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2001.

[MHD94] K. Malmkjær, N. Heintze, and O. Danvy. ML partial evaluation using set-based analysis. In *ACM SIGPLAN Workshop on ML and Its Applications*, pages 112–119, Orlando, Florida, USA, June 1994.

[MMV+98] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[Mog86] T. Mogensen. The application of partial evaluation to ray-tracing. Master's thesis, Computer Science Department, University of Copenhagen, Denmark, 1986.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Mog89] T. Mogensen. Separating binding times in language specifications. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 12–25, London, UK, September 1989. Addison-Wesley.

[MP02] R. Marvie and M. Pellegrini. Modèles de composants, un état de l'art. *Numéro spécial de la revue l'Objet*, 8(3), 2002.

[MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, April 1995.

[MS97] T. Mogensen and P. Sestoft. Partial evaluation. *Encyclopedia of Computer Science and Technology*, 37 :247–279, 1997.

[MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class. In E. Jul, editor, *ECOOP'98 - Object-Oriented Programming - 12th European Conference*, volume 1445 of *LNCS*, pages 355–382, Brussels, Belgium, July 1998.

[MSP+00] M. Morisio, C. B. Seaman, A. T. Parra, V. R. Basili, S. E. Kraft, and S. E. Condon. Investigating and improving a COTS-based software development. In *Proceedings of the 22th International Conference on Software Engineering*, pages 32–41, Limerick Ireland, 2000. ACM Press.

[MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[MVM97] G. Muller, E.-N. Volanski, and R. Marlet. Scaling up partial evaluation for optimizing the SUN commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–126, Amsterdam, The Netherlands, June 1997. ACM Press. ACM SIGPLAN Notices, 32(12).

[MWP+01] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2) :217–251, 2001.

[NEK94] J. Q. Ning, A. Engberts, and W. V. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5) :50–57, 1994.

[NHCL96] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specilization : Implementation and experimental study. Technical report, IRISA, Rennes, France, November 1996.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects - Survival Guide.* John Wiley and Sons Ltd., 1996.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Software Development Environments (SDE)*, pages 177–184, 1984.

[OPS92] N. Oxhøj, J. Palsberg, and M.I. Schwartzbach. Making type inference practical. In *ECOOP'92 - Object-Oriented Programming - 6th European Conference*, volume 615 of *LNCS*, pages 329–349. Springer-Verlag, 1992.

[Par72] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, December 1972.

[PD04] J. Privat and Roland Ducornau. Intégration d'optimisations globales en compilation séparée des langages à objets. In *LMO 2004 - Langages et modèles à objets*, pages 61–74, Lille, France, March 2004. Hermès.

[PDN86] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4) :307–334, November 1986.

[PEP93] *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation - PEPM'93*, Copenhagen, Denmark, June 1993. ACM Press.

[Pie02] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PS94] Jens P. and M. I. Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., 1994.

[PW92] Dewayne E. P. and Alexander L. W. Foundations for the study of software architecture. *ACM SIGPLAN Notices*, 17(4) :40–52, October 1992.

[RB89] T. Reps and T. Bricker. Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management, , ACM Software Engineering Notes 17*, pages 46–55, Princeton, NJ, USA, October 1989. ACM Press.

[RFS+00] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit : Component composition for systems software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation OSDI 2000*, pages 347–360, San Diego, USA, October 2000. ACM Press.

[Riv87] F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Écoles des Mines de Nantes, 1987.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 1 edition, 1999.

[RN01] P. Rapicault and A. Napoli. Evolution d'une hiérarchie de classes par interclassement. In R. Godin and I. Borne, editors, *LMO 2001 - Langages et modèles à objets*, pages 215–230, Le Croisic, France, January 2001. Hermès. L'Objet, 7(1-2).

[RS02] R. Rinat and S. Smith. Modular internet programming with cells. In Magnusson [Mag02], pages 257–280.

[RT96] T. Reps and T. Turnidge. Program specialization via program slicing. In Danvy et al. [DGT96], pages 409–429.

[Ruf93] E. Ruf. *Topics in online partial evaluation*. PhD thesis, Stanford University, February 1993.

[Sah91] D. Sahlin. *An Automatic Partial Evaluator for Full Prolog.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1991.

[Sam97] Johannes Sametinger. *Software engineering with reusable components.* Springer-Verlag, New York, NY, USA, 1997.

[SB99] Y. Smaragdakis and D. Batory. Implementing layered design with mixin layers. In R. Guerraoui, editor, *ECOOP'99 - Object-Oriented Programming - 13th European Conference*, volume 1648 of *LNCS*, pages 550–570, Lisbon, Portugal, June 1999. Springer-Verlag.

[Sch99] U.P Schultz. Black-box program specialization. In J. Bosch, C. Szyperski, and W. Weck, editors, *Fourth International Workshop on Component-Oriented Programming*, Lisbon, Portugal, June 1999. In conjunction with ECOOP 1999.

[Sch00] U.P. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation.* PhD thesis, Université de Rennes I, December 2000.

[Sco76] Dana S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3) :522–587, 1976.

[SDK+95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *ieeetse*, 21(4) :314–335, 1995.

[Ses00] R. Sessions. *COM+ and the battle for the Middle Tier.* Wiley, 2000.

[Sie00] J. Siegel. *CORBA 3 Fundamentals and Programming.* John Wiley and Sons Ltd., January 2000.

[SLC03] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Transactions on Programming Languages and Systems*, 25(4) :452–499, 2003.

[Sør94] M.H. Sørensen. Turchin's supercompiler revisited. Master's thesis, Computer Science Department, University of Copenhagen, 1994. DIKU Research Report 94/9.

[Sre01] V. C. Sreedhar. ACOEL on CORAL : A COmponent Requirement and Abstraction Language. In *Proceedings of the SAVCBS 2001 Specification and Verification of Component-Based Systems Workshop at OOPSLA 2001*, pages 125–131, Orlando, FL, USA, October 2001. ACM Press.

[Sre02] V. Sreedhar. Mixin'up components. In ICSE2002 [ICS02], pages 198–207.

[Sta76] J.F. Stay. Hipo and integrated program design. *IBM Systems Journal*, 15(2) :143–154, 1976.

[SW99] A. Schürr and A. Winter. UML, the future standard software architecture description language ? In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications for Businesses and Systems*, volume 523, chapter 14, pages 193–206. Kluwer Academic, 1999.

[Szy02] C. Szyperski. *Component Software.* Addison-Wesley, 2002. 2nd edition.

[Tic79] W. F. Tichy. Software development control based on module interconnection. In *ICSE '79 : Proceedings of the 4th international conference on Software engineering*, pages 29–41, Piscataway, NJ, USA, 1979. IEEE Press.

[Tur77] V.F. Turchin, editor. *Basic Refal and Its Implementation on Computers.* Moscow : GOSSTROI SSSR, TsNIPIASS, 1977. (In Russian).

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3) :292–325, July 1986.

[Unr] The Unravel Project. `http ://www.itl.nist.gov/div897/sqg/unravel/`.

[vHB88] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36 :401–412, 1988.

[vO02] R. van Ommering. Building product populations with software components. In OM [ICS02], pages 255–265.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, 2000.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In Hughes [Hug91], pages 165–191.

[Web] BEA WebLogic Server. `http ://www.weblogic.com/`.

[Wei84] M. Weiser. Program slicing. *ACM Transactions on Programming Languages and Systems*, 10(4) :352–357, July 1984.

[Wis] Wisconsin program slicing tool. `http ://www.cs.wisc.edu/wpis/`.

[WL86] M. Weiser and J. R. Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[Zha98] J. Zhao. Applying slicing technique to software architectures. In *Fourth International Conference on Engineering Complex Computer Systems*, pages 87–99, Monterey, California, USA, 1998. IEEE Computer Society Press.