Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation

Pascal Rapicault

To cite this version:

Pascal Rapicault. Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation. Modélisation et simulation. Université Nice Sophia Antipolis, 2002. Français. NNT : . tel-00505470

HAL Id: tel-00505470
https://theses.hal.science/tel-00505470v1

Submitted on 23 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

École doctorale « Sciences et Technologies de l'Information et de la Communication » de Nice - Sophia Antipolis
Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES ET TECHNIQUES

MODÈLES ET TECHNIQUES POUR SPÉCIFIER, DÉVELOPPER ET UTILISER UN FRAMEWORK : UNE APPROCHE PAR MÉTA-MODÉLISATION

---

présentée et soutenue publiquement par

Pascal RAPICAULT

le 25 Mai 2002 à l'E.S.S.I. *devant le jury composé de*

Président du Jury	Charles ANDRÉ	Université de Nice - Sophia Antipolis
Rapporteurs	Isabelle BORNE	Université de Bretagne-Sud
	Jean-Pierre BRIOT	Université de Paris 6
	Houari A. SAHRAOUI	Université de Montréal
Examinateur	Michel DAO	France Telecom R&D
Directeur de thèse	Jean-Paul RIGAULT	Université de Nice - Sophia Antipolis
Co-directeur de thèse	Mireille FORNARINO	Université de Nice - Sophia Antipolis

À mes parents,
À Estelle.
— Pascal

Table des matières

Conventions typographiques xv

Remerciements xvii

Introduction générale 1

1 Les frameworks 3
- 1.1 Définition . . . 3
- 1.2 Pourquoi utiliser un framework ? . . . 4
- 1.3 Types de frameworks et utilisation . . . 4
  - 1.3.1 Frameworks blancs . . . 5
  - 1.3.2 Frameworks noirs . . . 5
- 1.4 Frameworks et autres techniques de réutilisation . . . 5
  - 1.4.1 Composant . . . 6
  - 1.4.2 Schéma de conception . . . 6
  - 1.4.3 Bibliothèque . . . 7
- 1.5 Cycle de vie d'un framework . . . 7
- 1.6 Utilisation de frameworks . . . 8
  - 1.6.1 Personnalisation . . . 9
  - 1.6.2 Instanciation . . . 9
  - 1.6.3 Difficultés liées à la documentation . . . 10
- 1.7 Problématique de la thèse . . . 11
- 1.8 Plan de la thèse . . . 12

I Expression des dépendances structurelles 13

2 Techniques d'expression des dépendances structurelles 15
- 2.1 Exemple de framework : JUnit . . . 16
- 2.2 Critères de comparaison . . . 16
- 2.3 Représentations fonctionnelles . . . 17
  - 2.3.1 *Cookbook* et outils . . . 17
  - 2.3.2 *Active cookbooks* . . . 18
  - 2.3.3 *SEA-Preceptor* . . . 20
  - 2.3.4 Documentation de frameworks par des *patterns* . . . 20

2.3.5 Les *Hooks* . . . 22
2.4 Représentations structurelles . . . 22
2.4.1 Schémas de conception . . . 22
2.4.2 *Meta patterns* . . . 23
2.4.3 UML-F . . . 26
2.4.4 FRED et les *specialisation patterns* . . . 26
2.5 Techniques non dédiées aux frameworks . . . 29
2.5.1 *Law-governed regularities* . . . 29
2.5.2 *Logic meta programming* . . . 29
2.5.3 Outils dédiés aux schémas de conception . . . 29
2.5.4 *Design constraints* . . . 30
2.6 Récapitulatif des techniques d'expression des dépendances structurelles . 30

Problématique des dépendances structurelles **33**

3 Modèle d'expression des dépendances structurelles **35**
3.1 Présentation générale du modèle des dépendances structurelles . . . 36
3.1.1 Représentation flexible d'un framework . . . 37
3.1.2 Réification d'un framework et de ses dépendances . . . 38
3.2 Représentation d'un framework . . . 41
3.2.1 Éléments du modèle . . . 42
3.2.2 Réalisation, navigation et vérification . . . 46
3.2.3 Création d'un méta-framework . . . 51
3.3 Modèle Bean complet . . . 52
3.4 Réutilisation des dépendances . . . 53
3.4.1 ISL, le langage d'expression des règles de création . . . 53
3.4.2 Réutilisation des règles de vérification . . . 56
3.4.3 Réutilisation de dépendances . . . 57
3.5 Dépendances structurelles et la hiérarchie de modèles de l'OMG . . . 58
3.6 Conclusion sur notre modèle d'expression des dépendances structurelles . 63

4 Utilisation des dépendances structurelles **65**
4.1 Bibliothèque de dépendances . . . 65
4.1.1 Méthode / Classe . . . 66
4.1.2 Qualification . . . 66
4.1.3 Héritage . . . 68
4.1.4 Remarques sur la séparation des dépendances . . . 71
4.1.5 Structure . . . 73
4.1.6 Conclusion sur la bibliothèque des dépendances . . . 73
4.2 Représentation et utilisation de schémas de conception . . . 73
4.2.1 Abstract Factory . . . 74
4.2.2 Visitor . . . 77
4.2.3 Template Method . . . 79
4.2.4 Conclusion sur la représentation des schémas de conception à l'aide de dépendances . . . 79
4.3 Scénario d'utilisation du Visitor . . . 79

4.4 Intégration au cycle de vie 81
4.4.1 Conclusion sur l'intégration des dépendances structurelles dans le cycle de vie 85
4.5 Outils 85
4.5.1 Environnement de développement orienté framework 87
4.5.2 Implémentation du modèle 91
4.6 Conclusion sur l'utilisation de notre modèle d'expression des dépendances structurelles 92
4.7 Conclusion sur le modèle des dépendances structurelles et travaux futurs 92
4.7.1 Extension du modèle 93
4.7.2 Création d'outils 93
4.7.3 Génie logiciel 94

II Expression des dépendances comportementales 95

5 Techniques d'expression des dépendances comportementales 97
5.1 Exemple : la pompe à essence 98
5.2 Critères de comparaison 99
5.3 Diagrammes UMLs 99
5.3.1 Diagrammes de séquences 100
5.3.2 Diagrammes de collaboration 102
5.3.3 Diagrammes d'états-transitions 102
5.3.4 Diagrammes d'activités 104
5.4 Langage de description d'architectures (ADL) 105
5.4.1 Généralités 105
5.4.2 *Wright* 106
5.4.3 *Chemical Abstract Machines* 108
5.4.4 Approches connexes 108
5.5 Approches formelles 109
5.5.1 Expressions de chemins 109
5.5.2 Rumpe 110
5.5.3 *Regular types* 110
5.5.4 Yellin 110
5.6 Contrats 111
5.6.1 Contrats de Helm et al. 111
5.6.2 *Describing and Using Object Frameworks* 114
5.6.3 Contrats d'Eiffel 114
5.7 Schémas de conception 114
5.8 Récapitulatif 115

6 Présentation du modèle synchrone 117
6.1 Systèmes réactifs 117
6.2 Systèmes réactifs synchrones 118
6.3 Esterel, un langage synchrone 119
6.4 SyncCharts, une représentation graphique et formelle 120

6.4.1 Exemple de SyncCharts . . . 120
6.4.2 Description de la notation graphique . . . 120

Problématique des dépendances comportementales 125

7 Points de vue comportementaux 127
7.1 Point de vue client . . . 128
7.1.1 But . . . 128
7.1.2 Construction . . . 129
7.1.3 Fonctionnalités . . . 129
7.2 Point de vue de composition . . . 129
7.2.1 But . . . 129
7.2.2 Construction . . . 130
7.2.3 Fonctionnalités . . . 130
7.3 L'encapsulation . . . 131
7.4 Exemple de l'organisation des points de vue . . . 132
7.5 SyncClass, une représentation des points de vues comportementaux . . . 134
7.5.1 Notation graphique . . . 135
7.5.2 Structuration d'un SyncClass, et contraintes d'utilisations . . . 136
7.5.3 Sémantique intuitive . . . 136
7.5.4 Intégration des SyncClass à UML . . . 137
7.6 Projection modèle objet / modèle synchrone . . . 139
7.6.1 Comparaison structurelle et comportementale des modèles . . . 139
7.6.2 Limites de la projection . . . 140
7.7 Transformation d'un syncClass en syncCharts . . . 141
7.7.1 Corps d'une méthode . . . 141
7.7.2 Envoi de message . . . 142
7.7.3 Séquence . . . 143
7.7.4 Structure de contrôle . . . 145
7.7.5 Appel de méthode levant des exceptions . . . 150
7.7.6 Appel à super . . . 150
7.7.7 Résultat d'un SyncClass transformé . . . 151
7.7.8 Réentrance . . . 152
7.7.9 Résumé des différences SyncCharts / SyncClass . . . 153
7.8 Conclusion sur les points de vue comportementaux . . . 153

8 Utilisation des points de vue comportementaux 155
8.1 Intégration des SyncClass dans le cycle de vie . . . 155
8.1.1 Analyse et conception . . . 156
8.1.2 Implémentation . . . 156
8.1.3 Test unitaire . . . 156
8.1.4 Test d'intégration . . . 157
8.1.5 *Packaging* du composant . . . 157
8.1.6 Réutilisation (validation, exploitation) . . . 157
8.2 Outils généraux . . . 157
8.2.1 Editeur / navigateur . . . 158

8.2.2 Finaliseur 158
8.2.3 Configurateur 161
8.2.4 Simulateur de SyncClass 161
8.3 Générateurs 166
8.3.1 Générateur de code / générateur de SyncClass 166
8.3.2 Générateur de tests 167
8.4 Vérificateurs statiques et dynamique 168
8.4.1 Vérificateur statique de l'adéquation d'un ensemble de composants 169
8.4.2 Vérificateur statique d'un code source par rapport à une configuration 173
8.4.3 Vérificateur statique de substituabilité comportementale 173
8.4.4 Vérificateur dynamique 175
8.5 Conclusion sur l'utilisation des syncClass 176
8.6 Conclusion sur les dépendances comportementales et travaux futurs . . . 176
8.6.1 Modèle de composant 177
8.6.2 Notation graphique 177
8.6.3 Outils 178
8.6.4 Applications 178

III Expression du modèle conceptuel des dépendances d'un framework 181

9 Mise en œuvre des dépendances dans BLOCKS 183
9.1 Présentation de BLOCKS 183
9.1.1 Fonctionnement de l'historique 184
9.1.2 Une extension de l'historique 186
9.2 Dépendances de BLOCKS 187
9.2.1 Extension de Snapshot 188
9.2.2 Extension de BackstrackableSnapshot 188
9.3 Conclusion sur la mise en œuvre des dépendances dans BLOCKS 189

Conclusion générale 191

Glossaire 195

A Présentation d'ISL 1

B Grammaire des parties actions 3

C Intégration des SyncClass à UML 5
C.1 Modèle UML des SyncClass 5
C.2 Modèle de la connexion SyncClass / Classifier 9

Bibliographie 12

Table des figures

1.1 Frameworks et schémas de conception. 7
1.2 Activités rencontrées lors du développement d'une application et d'un framework. 8

2.1 Diagramme de classes partiel de JUnit. 17
2.2 Exemple de recette livrée avec JUnit. 19
2.3 Exemple d'une organisation possible des *patterns* décrivant JUnit. 21
2.4 *Hooks* décrivant la création d'un test pour JUnit. 23
2.5 Les schémas de conception de JUnit, extrait de [GB]. 24
2.6 Représentation des schémas de conception de JUnit en UML. 24
2.7 Utilisation des *Meta patterns* pour représenter les points de variations de JUnit. 25
2.8 Représentation de JUnit en UML-F, recomposé à partir de [FPR00a]. . . . 27
2.9 Exemple d'un *specialisation pattern* pour JUnit. 28

3.1 Description d'un framework. 37
3.2 Méta-description, description et applications dérivées d'un framework. . 38
3.3 Modélisation à deux niveaux. 39
3.4 Représentation des niveaux pour JUnit. La partie grisée désigne le framework initial. 40
3.5 Diagramme de classes du modèle des méta-frameworks. 41
3.6 Modèle de dépendances et exemple des JavaBeans. 43
3.7 Méta-modélisation détaillée de `Bean` et `BeanInfo`. 45
3.8 Fonctionnement de la `Factory`. 46
3.9 Modèle détaillé des méta-frameworks (F2). 47
3.10 Réalisation d'un `conceptualElement` sans dépendance. 48
3.11 Réalisation d'un élément conceptuel et d'une dépendance associée. 49
3.12 La navigation dans un modèle, quelques opérations. 50
3.13 Exemple d'une règle de vérification. 51
3.14 Modèle des JavaBeans avec la méthode `getDescriptor`. 52
3.15 Règle vérifiant la cardinalité de la méthode `getDescriptor`. 53
3.16 Exemple de schéma d'interactions pour la dépendance Bean / BeanInfo. . 54
3.17 Instanciation du schéma d'interactions BeanInteraction (figure 3.16). . . . 54
3.18 Exemple de la fusion de deux interactions. 55

3.19 Exemple de la fusion d'une interaction avec une boucle. 56
3.20 Exemple d'un schéma de vérifications pour Bean / BeanInfo. 56
3.21 Instanciation du schéma de vérifications Bean / BeanInfo. 57
3.22 Dépendance complète entre Bean / BeanInfo. 57
3.23 Réutilisation d'une dépendance. 58
3.24 Modélisation en UML. 59
3.25 Rapprochement de F0 et ElementPhysique de M1. 60
3.26 Métaclasse et *template* d'UML. 61
3.27 F1, F0 et ElementPhysique dans M1. 61
3.28 F1 et F2 dans M2. 62
3.29 Rapprochement F3 / MOF. 63

4.1 Méta-Représentation du schéma de conception composite. 66
4.2 Dépendance connectant une méthode à sa classe. 67
4.3 Dépendance de qualification. 67
4.4 Dépendance d'héritage. 68
4.5 Dépendance d'héritage. 69
4.6 Dépendance de surcharge, partie aspiration. 70
4.7 Dépendance de surcharge, partie propagation. 71
4.8 Signature de la dépendance surcharge. 71
4.9 Résultat de la fusion d'héritage1, aspiration et propagation. 72
4.10 Dépendance de structure. 73
4.11 Modélisation des dépendances du schéma de conception Abstract Factory. 75
4.12 Dépendance existentielle. 75
4.13 Dépendance de cardinalité. 76
4.14 Méta-représentation du schéma Visitor. 78
4.15 Extension de surcharge avec élément conceptuel pour la méthode. 79
4.16 Méta-représentation du schéma template Method. 80
4.17 Étapes de l'utilisation du schéma Visitor. 82
4.18 Classe TestCase. 83
4.19 Possibilités de réification pour les méthodes de TestCase. 83
4.20 Template Method et sa réalisation dans JUnit. 84
4.21 Dépendance pour la gestion des résultats. 84
4.22 Méta-framework associé à JUnit. 86
4.23 Framework, méta-framework et rôles associés. 87
4.24 Perspective permettant la manipulation d'un méta-framework. 89
4.25 Perspective permettant la manipulation d'un framework. 90
4.26 Navigateur de dépendances. 91

5.1 Diagramme de classes de la pompe à essence. 98
5.2 Instanciation dans un diagramme de séquences. 100
5.3 Diagramme de séquences. 101
5.4 Diagramme de collaboration. 102
5.5 Diagramme d'états-transitions. Le nom des receveurs correspond au nom des relations dans les diagrammes de classes. 103
5.6 Représentation partielle de la pompe à essence avec l'ADL *Wright*. 107

5.7 Utilisation des collaborations à la Yellin pour la représentation de la relation entre la boîte de la pompe à essence et l'afficheur de message. . . . 112
5.8 Contrats entre la boîte de la pompe à essence et l'afficheur de volume, et entre la boîte et l'afficheur de message. 113

6.1 SyncCharts de l'exemple ABRO, extrait de [And96a]. 121
6.2 Code Esterel correspondant au SyncCharts ABRO (cf figure 6.1). 121
6.3 Transitions avec priorité dans un SyncCharts. 122
6.4 Exemples d'utilisation des SyncCharts. 123

7.1 Modélisation UML d'un système constitué de 3 classes. 128
7.2 Point de vue client de la pompe à essence. 132
7.3 Point de vue client du moteur. 133
7.4 Point de vue client de l'afficheur de volume. 133
7.5 Point de vue de composition du moteur sur l'afficheur de volume. 134
7.6 Point de vue de composition de la pompe sur l'afficheur de volume. 134
7.7 Exemples de SyncClass incorrects. 136
7.8 Exemples de SyncClass corrects. 137
7.9 Liaison SyncClass / UML. À l'exception de la classe SyncClass, la hiérarchie présentée est extraite d'UML. 138
7.10 Correspondance entre le concept de classe et de SyncClass. 141
7.11 SyncCharts résultant de la transformation d'un corps de méthode. 142
7.12 SyncCharts présentant le problème posé par la séquentialité sans signal de fin de méthode. 143
7.13 SyncChart représentant l'appel synchrone d'une méthode. 144
7.14 SyncChart représentant l'appel asynchrone d'une méthode. 144
7.15 Transformation d'une expression séquentielle en un SyncCharts. 145
7.16 SyncCharts résultant de la transformation d'une conditionnelle. 146
7.17 SyncCharts résultant de la transformation d'une itération. 147
7.18 Code java d'une exception. 148
7.19 SyncCharts résultant de la transformation d'une exception. 148
7.20 SyncCharts résultant de la transformation d'un throw. 149
7.21 SyncCharts montrant le marquage du contexte dans les appels de méthodes. 151
7.22 SyncClass montrant un cas de réentrance avec blocage. 152
7.23 SyncClass montrant un cas de réentrance sans blocage. 152
7.24 Etats puits dans un SyncCharts. 154

8.1 Les SyncClass dans le cycle de vie. 158
8.2 SyncClass et outils. Les outils dont le rectangle est blanc ne sont utilisés que par le développeur. 159
8.3 Éditeur de SyncClass. 160
8.4 Exemple de diagramme objets utilisé pour une configuration. 161
8.5 SyncCharts montrant le problème de communication de deux composants. 162
8.6 Connecteur généré par la configuration. 163
8.7 Algorithme de connexion des composants utilisé par le configurateur. . . . 164

8.8 Capture d'écran du simulateur. 165
8.9 Point de vue client du moteur et son code source généré. 166
8.10 Observateur généré pour la vérification statique. 169
8.11 Instrumentation d'un syncCharts pour la vérification statique. 170
8.12 Utilisation de XEVE pour la recherche d'une potentielle émission de KO. . 171
8.13 Fenêtre de résultat de XEVE indiquant que le signal KO n'est pas jamais émis. 172
8.14 Exemple d'une sous-classe (B) ou un appel de méthode a été déplacé. . . . 174
8.15 Vérification dynamique par méta-objet. 176

9.1 Diagramme de classes de la gestion d'historique dans Blocks. 184
9.2 Point de vue client de la classe Snapshot. 185
9.3 Point de vue client de la classe `HistoryDag`. 185
9.4 Extension de BLOCKS pour la gestion d'un historique navigable. 186
9.5 Point de vue client de `BacktrackableHistoryDag`. 187
9.6 Point de vue client de la classe `BacktrackableSnapshot`. 188
9.7 Dépendance pour la création de sous-classe de `Snapshot`, `Delta` et `Slot`. 189
9.8 Représentation des dépendances, avec gestion du cas pour `Backtrackable-Snapshot`. 190

C.1 Diagramme de classes des SyncClass. 6
C.2 Liaison SyncClass / UML. 10

Liste des tableaux

2.1 Comparatif des techniques d'expression des dépendances structurelles. . . 31

5.1 Comparatif des techniques de représentation des dépendances comportementales. 116

7.1 Tableau de synthèse sur la correspondance modèle objet / modèle synchrone. 140
7.2 Différents styles de conditionnels. 145
7.3 Représentations de boucles en SyncClass. 147

8.1 Intégration de SyncClass dans différents types de composants. 159
8.2 Outils utilisés par activité. 177

Conventions typographiques

Les conventions typographiques utilisées dans ce document sont les suivantes :
- framework* : désigne un mot qui a une entrée dans le glossaire ;
- `Descripteur` : désigne dans le texte des noms utilisés dans les figures ;
- *cookbook* : désigne les mots anglais non traduits ;
- « *To be or not to be* » : désigne une citation anglaise ;
- « A bon chat bon rat » : désigne une citation française.

Remerciements

Je voudrais tout d'abord remercier les membres du jury, Charles André, Isabelle Borne, Jean-Pierre Briot, Michel Dao et Houari Sahraoui pour m'avoir fait l'honneur d'évaluer mes travaux de recherche et les rapporteurs, Isabelle Borne, Jean-Pierre Briot et Houari Sahraoui pour leur remarques pertinentes.

Sincères remerciements à mes directeurs de thèse Mireille Fornarino et Jean-Paul Rigault, qui ont su me diriger et me faire bénéficier de leur grande expérience tout en me laissant une entière liberté.

Je suis reconnaissant envers Charles André pour avoir accepté de présider mon jury, pour m'avoir permis de détourner sa notation graphique et aussi pour m'avoir fourni de précieuses informations sur les modèles synchrones.

Merci à Sabine Moisan et Annie Ressouche pour avoir pris le temps de m'écouter lorsque mon modèle prenait forme, et de m'avoir donné l'opportunité de documenter BLOCKS.

Je tiens aussi à remercier tous les gens d'OTI et plus particulièrement John Wiegand pour sa générosité.

Je voudrais ensuite remercier :

- Anne-Marie Dery pour m'avoir écouté et conseillé dans les moments difficiles ;
- Fred pour ses méta-discussions et discussions sur le niveau méta ;
- Mes principaux partenaires de la joyeuse équipe du RU (aussi connue comme les joueurs de belote lourds de la Kfête) : Anne « arrêtez je vais être choquée », Bob et SaMain, Lionel « je suis super discret », Lolo, Mam « si si j'me suis remis à rédiger, mais j'ai encore changé de copine », Valéry « plus que 3 ans », Xav « ca y est j'ai terminé », Xoff « je suis un gars sérieux, mais j'hésite entre toutes ces femmes » ;
- Les Belettes pour les discussions techniques et les parties de jeux ;
- Luc Bourlier qui en remerciement de la mise en oeuvre de mes idées les plus farfelues a été exilé à Minneapolis.
- Nghi dont les cours de Kung Fu m'ont servi d'exutoire dans les chapitres difficiles de la rédaction ;
- et aussi Val, Dominique, Annabelle, David, Jérémy, Michel, Adeline, Pierre, Olivier, Freddy, Roselyne, Manu, Gérald, Cédric, Pierre-Charles, Nono...

Plus près de moi il y a Estelle dont l'amour, la patience, et l'assistance m'a apporté le réconfort et la motivation nécessaire à passer cette dernière année et pour qui un simple merci ne suffira jamais.

Et enfin, merci à ma famille, mes parents et Roger pour m'avoir soutenu tout au long de mes études et avoir accepté mon départ si loin de la verdoyante Sologne.

— Pascal

Introduction générale

De manière générale un « framework » est un squelette d'application qui peut être personnalisé ou instancié par un utilisateur afin de créer une application. Les frameworks orientés objets, constitués de (hiérarchie de) classes coopérant, en sont un exemple. Ces derniers ont suscité un grand engouement. Alors que la notion de classe commençait à montrer les limites d'une réutilisation à trop faible granularité, ces frameworks élevaient la granularité au niveau d'une architecture et leur caractère « préfabriqué » permettait d'envisager le développement très rapide d'applications complètes et performantes.

De nombreux exemples montrent que cet espoir a été effectivement réalisé, parfois. Malheureusement, les frameworks ont rapidement révélé leur principal inconvénient : leur difficulté d'utilisation qui réservait leur usage à des spécialistes, voire seulement à leur propres concepteurs. Cette difficulté est due à deux raisons complémentaires, au moins. La première est liée au caractère architectural des frameworks qui implique une collaboration étroite entre leurs constituants ; c'est la raison de leur puissance mais cela entraîne aussi un couplage et donc des dépendances entre ces éléments. La seconde est la conséquence des espoirs placés dans des frameworks qui a conduit leurs concepteurs à en augmenter la puissance, l'expressivité et donc la complexité.

Cette seconde raison ne disparaîtra sans doute pas de sitôt. La taille et la complexité des frameworks augmentent inexorablement, alors que les délais de livraison des applications sont de plus en plus courts. Il est donc indispensable de rechercher des solutions pour permettre aux utilisateurs de faire face à des frameworks inévitablement plus complexes.

Les concepteurs de frameworks ont bien entendu pris conscience de ce problème et ces dernières années ont vu apparaître des outils plus ou moins génériques d'assistance à l'utilisation. Toutefois, aucune solution vraiment générale, et donc acceptable consensuellement, n'a encore vu le jour. Le problème n'est certes pas simple. En particulier, une des difficultés majeures est de proposer une représentation des dépendances d'un framework qui soit adaptée au niveau d'abstraction de l'utilisateur et à ses compétences.

Si une telle représentation existe, elle doit proposer un modèle conceptuel des couplages internes d'un framework. Expliciter ce modèle doit permettre de fournir une documentation ou de concevoir des outils permettant véritablement d'aider l'utilisateur. Ce modèle comporte deux aspects complémentaires : un aspect structurel lié à l'architecture même du framework, et un aspect comportemental lié aux interactions dynamiques entre les éléments de cette architecture.

Ainsi dans cette thèse nous mettons en avant la modélisation de ces différentes formes de couplage et proposons des modèles et techniques permettant de les représenter. Il est souhaitable que l'explicitation de ce modèle n'augmente pas (trop) le travail du développeur de framework. Nous privilégions donc des techniques qui s'intègrent facilement dans le cycle de développement du framework et des représentations qui peuvent être obtenues à partir des pratiques actuelles du développement logiciel.

Ce manuscrit est organisé en quatre parties. La première détaille les causes des problèmes d'utilisation rencontrés dans les frameworks. La deuxième présente notre représentation des dépendances structurelles qui repose sur une méta-modélisation du framework. L'expression des dépendances comportementales est le sujet de la troisième partie ; elle s'appuie sur le modèle synchrone des systèmes réactifs. Enfin une quatrième partie, plus brève, montre l'utilisation conjointe des deux modèles proposés dans le cadre d'un exemple.

Chapitre 1

Les frameworks

Les frameworks*[1] constituent avec les composants*, les schémas de conception* (*design patterns*) et les bibliothèques* (*libraries*), des techniques de réutilisation du logiciel. Afin de mieux comprendre quels sont les buts de chacune de ces techniques, nous détaillons tout d'abord les frameworks qui font l'objet de cette thèse, puis les comparons aux autres.

« *A framework is the skeleton of an application that can be customized by an application developer* » [Joh97b, Joh97a].

1.1 Définition

Afin de définir un framework, nous préférons citer quelques définitions plutôt que d'en créer une nouvelle. Ainsi, voici une définition de Ralph Johnson [Joh97a] : « *a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their* instances interact ». Celle-ci peut être complétée de la définition suivante : « *a framework is the skeleton of an application that can be customized by an application developer* » [Joh93]. Ces définitions sont les plus utilisées, cependant certaines personnes omettent la dimension liée au domaine d'application, et d'autres vont jusqu'à affirmer qu'un framework est « *A design pattern implementation* » [Lor93] quand il ne s'agit pas d'une simple « classe abstraite » [Pre00]. Nous indiquons au lecteur que nous adhérons principalement aux définitions qui introduisent la notion d'application, et qu'ainsi nous n'adhérons, parmi les définitions précédemment citées, qu'à celles de Johnson.

En offrant modularité, extensibilité et un comportement minimal, un framework a pour but de faciliter la construction d'une application en fournissant un petit ensemble de composants (classes) connus et éprouvés à composer ou à étendre. Il va au-delà de la simple réutilisation de code en favorisant la réutilisation d'analyse et de conception. Nous retiendrons principalement de cet ensemble de classes son caractère adaptable et extensible, et la représentation partielle incomplète qu'il propose d'une application.

[1]Le terme framework sera gardé en anglais faute d'une traduction simple.

Un framework a donc à la fois un aspect structurel (une architecture de classes) et un aspect comportemental (les fonctionnalités fournies par ces classes).

Quelques-uns des frameworks les plus connus sont HotDraw [Bra95], ET++ [WGM88], CORBA [JMG97], SanFransisco [Boh98], JUnit [BG98], Eclipse [OTI01], AWT [Mic97a], JavaBeans [Mic97b], COM, X11, EJB [Mic]... Ils couvrent des domaines aussi variés que les interfaces graphiques au sens large (HotDraw, ET++, AWT, X11), la construction d'environnements de développement (Eclipse), la gestion d'entreprise (SanFransisco), les bus logiciels (DCOM, CORBA), les modèles de composants (COM, EJB, JavaBeans), etc.

1.2 Pourquoi utiliser un framework ?

Un framework est utilisé comme base à l'écriture d'une application, car il fournit des éléments à partir desquels il est plus facile de construire une application.

Les bénéfices liés à l'utilisation d'un framework sont [SBF96] :
- la cohérence entre les différents produits dérivés de celui-ci ;
- l'augmentation de la productivité [MN96] ;
- la réduction de la maintenance ;
- l'économie d'argent.

Paradoxalement, ces bénéfices ne deviennent réels que si le framework est utilisé sur une longue période de temps [FSJ99, p. 11],[SBF96]. En effet, l'apprentissage d'un framework est plus long que l'apprentissage d'une simple bibliothèque, et peut parfois être aussi long que de concevoir une application. Cela est dû à la complexité du framework qui ne résout pas un problème particulier, mais une généralisation de ce problème. Il est donc conseillé de réutiliser le même framework plusieurs fois. Par ailleurs, la construction d'un framework est une activité encore plus coûteuse[2]. Elle requiert une très grande expertise du domaine à modéliser de manière à fournir une flexibilité suffisante aux développeurs et la validation d'un framework n'est faite qu'une fois la mise en œuvre d'applications suffisamment différentes effectuée [Joh93].

1.3 Types de frameworks et utilisation

L'utilisation d'un framework implique l'extension, la paramétrisation, la composition, ou la définition de classes afin de réaliser une application. Nous appelons cette application, application dérivée*.

Bien qu'on puisse caractériser les frameworks par leur domaine d'application, leur taille, leur langage d'implémentation, le fait qu'ils possèdent ou non leur boucle de contrôle... nous avons décidé de les étudier par leur type d'utilisation et leurs mécanismes d'adaptation.

Ainsi, on identifie deux grands types de frameworks : les frameworks blancs et les frameworks noirs. La couleur ne représente pas une dichotomie stricte, mais décrit un style d'utilisation d'un framework qui n'est jamais ni complètement blanc, ni complètement noir.

[2]Un framework est le plus souvent développé par généralisation à partir d'applications existantes

1.3.1 Frameworks blancs

Le terme framework blanc* désigne à la fois une forme de framework et sa technique de réutilisation. Ainsi l'utilisateur d'un framework blanc a accès au code source et doit généralement l'étudier avant de pouvoir l'étendre. L'utilisation d'un tel framework, principalement basée sur l'héritage, est très liée à des fonctionnalités du langage d'implémentation (héritage, liaison dynamique, polymorphisme), et se fait en ajoutant de nouvelles classes, sous-classes, surchargeant des méthodes... Bien qu'avec un tel framework toutes les modifications soient possibles, elles se produisent dans des endroits identifiés où l'adaptation a été prévue. On identifie ces endroits par les termes de « *hot-spots* » [Pre94] ou « *axis of variation* » [DMNS97]. Nous traduirons ces termes par point de variation*. Un point de variation identifie une classe ou un ensemble de classes à modifier afin de particulariser le framework. Ces points de variation sont décrits dans la documentation qui est associée au framework.

HotDraw et JUnit sont des représentants de cette catégorie de frameworks.

1.3.2 Frameworks noirs

Comme le terme framework blanc, le terme framework noir* identifie à la fois une catégorie de framework et une technique de réutilisation. Un framework noir est caractérisé par l'impossibilité d'accéder au code source, mais surtout par la technique d'utilisation qui lui est liée. Ainsi l'utilisation d'un framework noir se fait par la paramétrisation de classes et par l'assemblage de composants. Tout comme dans les frameworks blancs, ces modifications sont apportées aux points de variation.

COM est un représentant de cette catégorie de framework.

Cette distinction entre framework noir et blanc résulte de la dualité entre la facilité de la conception et la facilité d'utilisation. Un framework noir est plus difficile à développer [RJ96] mais plus simple à utiliser, alors qu'un framework blanc est plus simple a développer, mais plus complexe à utiliser. En effet, les difficultés liées à l'utilisation d'un framework noir sont principalement comportementales alors que pour un framework blanc elles sont principalement structurelles mais aussi comportementales.

Certains frameworks noirs sont livrés avec un ensemble de composants prêts à l'emploi. On parle alors de *component frameworks**. Tout comme les frameworks gris*, ils représentent un compromis entre la facilité d'utilisation et la facilité de conception en fournissant à la fois des zones « noires » et des zones « blanches » : « *Grey-box frameworks blend the two approaches, allowing customisation by means of object composition, and by means of subclassing* » [TAL95].

1.4 Frameworks et autres techniques de réutilisation

On voit très souvent associés les termes framework, schéma de conception et composant (par exemple Ralph Johnson écrivait « *framework = design patterns + components* » [Joh97b]). Cependant, rien n'empêche d'utiliser chacune de ces techniques indépendamment les unes des autres. C'est pourquoi nous comparons les avantages et

les inconvénients de chacune d'entre elles par rapport au framework. Nous ne comparerons pas les frameworks aux techniques de programmation générative [CE00] ni aux générateurs d'applications comme le fait parfois Johnson [Joh97b].

1.4.1 Composant

Le composant représente l'entité de réutilisation de code par excellence. Il est créé pour cela. Un composant est théoriquement simple à utiliser. Idéalement il suffit de le connecter à l'aide de glue* avec d'autres composants pour créer une application [McI68]. Malheureusement la paramétrisation nécessaire à leur souplesse d'utilisation gêne cet idéal.

Un framework peut être considéré comme un composant hautement paramétrable et d'une très grande réutilisabilité. Il promeut une classe d'application et donc son analyse, conception, et code. Le cumul de ces propriétés est à l'origine de la puissance du framework mais aussi de sa difficulté d'utilisation.

Une différence entre un composant et un framework se situe au niveau de l'apprentissage nécessaire à leur utilisation. Alors qu'il suffit d'appréhender chaque composant indépendamment (il suffit de comprendre la signification des méthodes et de leurs paramètres), l'apprentissage d'un framework ne peut se limiter à la compréhension classe par classe. Il faut tenir compte des relations structurelles et comportementales entre les différents éléments du framework.

En fait, les frameworks et les composants sont des technologies complémentaires. En effet, lorsqu'un framework évolue de blanc vers noir [RJ96], les classes qui ont été initialement développées pour l'étendre deviennent alors des composants, transformant alors le framework noir en un *component framework*. Cette évolution se produit lorsque dans le contexte de différentes applications on constate la même extension. Dans ce cas l'extension est figée sous forme d'un composant paramétrable et le framework s'obscurcit.

1.4.2 Schéma de conception

Un schéma de conception est la description nommée d'un problème de conception récurrent et de sa solution. Cette solution, bien que décrite précisément, ne doit pas obligatoirement être reproduite à l'identique laissant un degré d'adaptation au contexte. Un schéma de conception promeut la réutilisation de conception. Il facilite, grâce à son nommage, la communication d'informations liées à la conception. Le plus célèbre ensemble de schémas de conception est celui du livre du « Gang of Four » (GoF) [GHJV95].

Un schéma de conception est plus abstrait qu'un framework (il n'est pas lié à un domaine d'application) et plus petit. La conception d'un framework met généralement en œuvre plusieurs schémas de conception : « *Design patterns are the micro-architectural elements of frameworks* » [Joh97b]. La figure 1.1 illustre ce fait en montrant la possibilité pour un même élément de prendre part à plusieurs schémas de conception. Il est à noter que les schémas de conception du GoF ont été trouvés par l'analyse de frameworks.

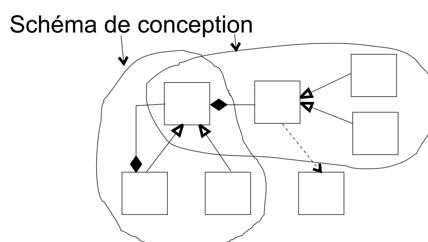


FIG. 1.1 – Frameworks et schémas de conception.

1.4.3 Bibliothèque

Les bibliothèques constituent la technique de réutilisation la plus courante actuellement. Quand elles sont écrites dans un langage à objet, elles sont composées d'un ensemble de classes leur valant souvent d'être confondues avec les frameworks [FSJ99, p. 58]. Elles promeuvent plus particulièrement la réutilisation de code et sont prêtes à l'emploi contrairement aux frameworks qui requièrent une adaptation.

Par ailleurs, une bibliothèque ne fournit jamais la boucle de contrôle d'une application alors que certains frameworks le font.

« *Frameworks are firmly in the middle of reuse techniques. They are more abstract and flexible (and harder to learn) than components, but more concrete and easier to reuse than a raw design (but less flexible and less likely to be applicable). They are most comparable to reuse techniques that reuse both design and code, such as application generators and templates* » [Joh97b].

1.5 Cycle de vie d'un framework

Que ce soit dans un cycle en V, cascade, spirale ou autre, développer une application (à partir ou non d'un framework) ou un framework requiert les mêmes activités : analyse, conception, tests unitaires, tests d'intégration, validation et packaging [JBR99]. Cependant si l'on n'y regarde de plus près la complexité, l'importance et la motivation de chacune de ses activités n'est pas la même selon les cas.

Ainsi la flexibilité requise par un framework ne peut être atteinte qu'au prix d'une analyse et conception cherchant explicitement cette flexibilité et réutilisabilité ; ce qui n'est pas le cas dans le développement d'une application.

Au-delà de ce point, la principale différence se trouve lors des tests d'intégration (« *Are we doing the system right ?* ») et de la validation (« *Are we doing the right system ?* »). Alors que dans le contexte du développement d'une application ces activités consistent à vérifier le fonctionnement des classes écrites et à vérifier l'adéquation de l'application avec les besoins du client, cela est beaucoup plus complexe dans le cas du framework. En effet, les besoins ne sont pas aussi clairement établis et les tests d'intégration et la validation doivent donc permettre de s'assurer que le framework offre bien la flexibilité nécessaire au développement d'applications. Le caractère incomplet du framework rend ces activités d'autant plus délicates et longues que des applications

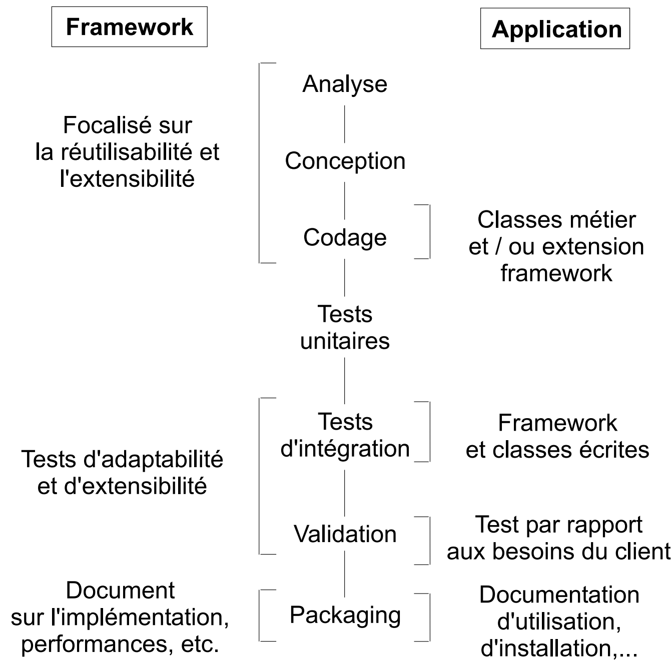


FIG. 1.2 – Activités rencontrées lors du développement d'une application et d'un framework.

jouets doivent être mises au point pour les besoins spécifiques de ces tests. Il apparaît donc qu'afin d'avoir des tests crédibles, beaucoup de temps doit être passé et de nombreuses applications utilisant différents aspects du framework écrites. En fait, on constate que ces activités ne sont réellement pertinentes que dans le cadre d'une application dérivée, mais qu'elles sont cependant indispensables afin de vérifier le bon fonctionnement du framework.

L'autre différence notable entre le développement d'une application et d'un framework, qui est une conséquence du public visé, se trouve dans le *packaging* (la mise en forme prête à l'emploi). Dans le cas d'une application la documentation doit expliquer comment l'utiliser, accéder à une fonctionnalité ou montrer des captures d'écran, alors que pour un framework il est nécessaire de fournir le domaine visé, les possibilités d'extension, les performances, des exemples d'utilisation, etc. Ces informations sont d'autant plus primordiales qu'elles permettront à l'utilisateur de tirer avantage du framework.

1.6 Utilisation de frameworks

Nous avons brièvement vu les frameworks et leurs types d'utilisation. Nous analysons maintenant les problèmes posés par l'utilisation des frameworks qui constituent le cœur du travail présenté dans ce document.

1.6.1 Personnalisation

De par les nombreuses possibilités qu'offre un framework, le terme utilisation prend plusieurs significations qui sont parfois confuses. Ainsi, dans un framework blanc le terme utilisation a souvent la signification d'extension (création de nouvelles classes, adaptation de la structure du framework...). C'est ce que nous appelons personnalisation*. Les problèmes rencontrés lors de la personnalisation d'un framework sont essentiellement dus aux dépendances inhérentes à la structure du framework, aux éléments constituant l'application voire à la combinaison des deux « *Class dependencies occur when different hot spots aren't independent of each other, but represent the same axis of variability* » [Rüp99]. Bien qu'ayant un aspect comportemental, ces dépendances sont principalement structurelles, et leur respect est primordial pour assurer le bon fonctionnement du framework. En particulier leur non-respect peut entrainer des erreurs à la compilation ou à l'exécution. Cependant ces dépendances ne font pas l'objet d'une attention particulière lors de la conception du framework. Elles sont considérées comme faisant partie de l'implémentation, et ne sont donc prises en compte que tardivement lors de la création de documentation comme les *cookbooks* (sous-section 2.3.1). « *Framework instantiation usually is more complex than simply assigning concrete classes to roles : variation points might have interdependencies, might be optional, and so on.* » [FPR00b] [3].

Parfois la modification aux points de variation ne suffit plus pour étendre le framework. Bien que cela ne soit pas conseillé pour des aspects de portabilité, l'utilisateur a recours à la modification de classes centrales au framework. Pour cela il peut soit modifier le code source du framework, ce qui pose des problèmes liés à la traçabilité des modifications, soit créer de nouvelles sous-classes, ce qui pose des problèmes liés à l'instanciation et au référencement de cette nouvelle sous-classe. Ces problèmes et la solution adoptée, mettant en œuvre le mécanisme d'interclassement [RN01], ne seront pas développés dans le contexte de ce document.

1.6.2 Instanciation

La seconde signification du terme utilisation décrit l'instanciation* d'un framework pour la réalisation d'une application. Ceci consiste en la création et la connexion d'instances soit des classes du framework soit des classes obtenues par personnalisation. Ainsi, pour un framework blanc elle se produit après la personnalisation, alors que pour un framework noir elle consiste plus généralement à sélectionner les composants intéressants et à les connecter. La difficulté de cette utilisation réside dans la compréhension des dépendances comportementales existant entre les instances à l'exécution, et le protocole d'utilisation* de chacune de ces instances. En effet, de par sa complexité, un framework implique inévitablement des dépendances comportementales entre les éléments qui le composent. Ainsi certaines classes doivent être initialisées avant d'être appelées, ou les appels de méthodes de plusieurs composants doivent être enchevêtrés dans le temps. Les erreurs générées par la mauvaise utilisation du protocole du framework sont d'autant plus difficiles et fastidieuses à corriger que, pour les reproduire, il faut parfois rejouer de longues séquences de tests.

[3] On notera que l'auteur utilise instanciation avec le sens de personnalisation.

Nous n'évoquons pas les problèmes liés à l'utilisation de plusieurs frameworks dans la réalisation d'une seule application. En effet, cela accroît considérablement les problèmes d'instanciation, et de surcroît, pose un problème lié à la fusion des boucles de contrôle des frameworks utilisés [MB97].

La citation suivante synthétise les problèmes précédents et constate leur enchevêtrement : « *customizing is typically done by subclassing an existing class of the framework and overriding a small number of methods. Sometimes, however, the framework insists that the customization preserve* a protocol of collaboration *between several subclasses, so customization requires the parallel development of these subclasses and certain of their methods* » [BD99, p. 495].

1.6.3 Difficultés liées à la documentation

L'une des causes des problèmes rencontrés lors de la personnalisation et de l'instanciation est la mauvaise documentation des frameworks. Cela est dû à la perte des informations de conception. On compte parmi ces informations les schémas de conception utilisés, le comportement des composants, les dépendances entre tous les éléments, la localisation des points d'extension, etc. Cette mauvaise documentation a pour cause :

- l'inadéquation entre l'information à représenter et les supports disponibles rendant délicat l'accès à l'information à travers des outils d'aide à l'utilisation ;
- la mauvaise intégration de cette documentation, sa forme inadéquate comme par exemple des diagrammes de classes livrés sur papier ;
- le coût de production de la documentation, mais aussi le coût engendré par la conservation de la synchronisation entre cette documentation et le code.

Cette documentation est d'autant plus dure à produire qu'elle doit satisfaire des utilisateurs aux préoccupations différentes [BD99, p. 496] :

- le développeur d'application. Les débutants (utilisateurs qui ne connaissent pas le framework) ont pour objectif d'apprendre au plus vite comment étendre et paramétrer le framework pour écrire leur application. Ils ne cherchent pas immédiatement à comprendre le fonctionnement interne en détail, mais plutôt à savoir comment utiliser les fonctionnalités du framework. Ils ont donc principalement besoin de connaître les points de variation. Les experts (ou utilisateurs familiers) ont quant à eux moins besoin de consulter la documentation qui leur est d'un apport restreint.

  Les fonctionnalités requises pour écrire l'application sont indépendantes de la connaissance du framework. Ainsi, on appellera utilisateur basique, l'utilisateur dont les besoins sont couverts par les points de variation et utilisateur avancé un utilisateur dont les besoins requièrent une connaissance précise du fonctionnement du framework. On notera que la même nature de connaissance est nécessaire au mainteneur ;
- l'utilisateur en charge de la maintenance du framework. Son objectif étant de maintenir le framework, il a besoin d'en connaître l'architecture globale, son fonctionnement interne, et la flexibilité qui était requise initialement lors de sa création. Ainsi il doit connaître en priorité le fonctionnement des points de variation, puis le rôle de chacune des classes sans oublier leurs interactions ;

– le développeur d'un autre framework. Son objectif est de trouver de l'inspiration pour implémenter un nouveau framework (quel que soit son domaine), ainsi il sera particulièrement intéressé par les points de variation et la flexibilité qu'ils fournissent ;
– le vérificateur. Son objectif est de s'assurer de la rigueur de son système. Pour cela, il utilise généralement des méthodes formelles dans le but de clairement spécifier les limites d'une utilisation (protocole), de l'extension, et de la paramétrisation d'une classe. Un tel souci de vérification est par exemple présent dans le framework BLOCKS [MRR01].

« *The key question in framework documentation is how to produce adequate information dealing with a specific specialization problem and how to present this information to the application developer.* » [HHK+01]

« *Framework documentation : accurate and comprehensible documentation is crucial to the success of large-scale frameworks. However, documenting frameworks is a costly activity and contemporary tools often focus on low level, method oriented documentation, which fails to capture the strategic roles and collaborations among framework components* » [Fay99]

1.7 Problématique de la thèse

Malgré les différences de but des utilisateurs du framework, nous pensons qu'ils ont tous la nécessité d'en respecter les dépendances, encore faut-il leur donner accès à ces informations ou à des outils permettant l'accès à ces dépendances. Ces dépendances, étant connues du développeur, nous pensons qu'il est souhaitable que cette information dérive des modèles manipulés dans les phases de développement du framework, car ceci contribuera à réduire l'effort nécessaire à la mise en place d'une documentation.

Cette thèse donne donc des modèles pour représenter cette information de dépendances afin de résoudre les problèmes liés à l'instanciation et la personnalisation des frameworks. Plus particulièrement, elle présente des techniques d'expression du modèle conceptuel des dépendances d'un framework.

Dans notre étude nous avons séparé ce modèle conceptuel de dépendances en deux aspects complémentaires. Le premier est un aspect structurel, qui est principalement sollicité lors de la personnalisation. Le second est un aspect comportemental sollicité par l'instanciation. Le premier est lié à l'architecture du framework et au respect de l'utilisation qui doit être faite de celle-ci, alors que le second est lié au comportement produit par cette architecture et donc aux interactions entre les composants du framework.

Ces dépendances sont étudiées de manière à en fournir une expression claire et opérationnalisable. De surcroît, cette expression sera intégrée au cycle de vie du framework favorisant l'utilisation des modèles de données propres aux développeurs dans la création de cette base d'informations, et évitant ainsi une surcharge de travail au développeur.

1.8 Plan de la thèse

Ce document est constitué de trois grandes parties dont les deux premières décrivent les modèles d'expression des dépendances d'un framework et la dernière un exemple d'utilisation commun de ces deux modèles.

La première partie présente le modèle d'expression des dépendances structurelles. Après avoir inventorié puis synthétisé les techniques existantes pour exprimer de telles dépendances (chapitre 2), notre solution est décrite au chapitre 3. Cette solution, qui fait d'une dépendance structurelle une entité de premier ordre, met l'accent sur la flexibilité offerte pour capturer les dépendances. Afin d'exemplifier ce dernier point, le chapitre 4 présente une bibliothèque des dépendances structurelles qui est alors utilisée pour montrer l'intégration de ce concept dans le développement d'un framework. Enfin, l'esquisse d'un outil mettant en œuvre les dépendances est faite avant de conclure sur les travaux présentés dans cette partie.

La seconde partie présente le modèle d'expression des dépendances comportementales. Cette partie est organisée de la même manière que la première. Ainsi elle débute par un état de l'art des techniques d'expression comportementales (chapitre 5), présente le modèle synchone au chapitre 6 puis introduit au chapitre 7 notre notation graphique basée sur ce modèle. Le chapitre 8 présente l'intégration de cette notation au cycle de vie du framework et les différentes vérifications formelles que nous avons mises en œuvre dans un environnement d'aide à la spécification, conception et développement de composant. Enfin une conclusion clôture les travaux présentés dans cette partie.

La troisième partie, plus succincte, réunit ces deux propositions autour d'un exemple et montre ainsi la complémentarité des approches.

Enfin, une conclusion dresse un bilan des deux modèles et ouvre une voie pour des travaux futurs les réunissant.

Première partie

Expression des dépendances structurelles

Chapitre 2

Techniques d'expression des dépendances structurelles

L'objectif de cette section est de répertorier, dans le contexte des frameworks, les techniques d'expression des dépendances structurelles*. Bien qu'à notre connaissance le terme de dépendance structurelle n'ait jamais été utilisé, le problème est identifié depuis longtemps et se retrouve dans le terme de *design constraints* chez Klarlund et al. [KKS96], ou de *Law governed regularities* chez Minsky [Min96]. Quel que soit le signifiant, le sens de ces mots signifie que la modification d'un élément d'une application a des répercussions sur d'autres parties de l'application, et cela sur un aspect purement structurel. Ainsi, une *design constraint* est par exemple utilisée pour indiquer que toute sous-classe d'une classe persistante doit être persistante. Nous avons délibérément évité tous les termes restrictifs comme contrainte, conception ou implémentation, pour qualifier nos dépendances. Bien qu'il puisse exprimer un processus de propagation, dans le monde objet et particulièrement d'UML avec OCL, le mot contrainte a plus un sens de vérification et d'expression de règles de validité alors que nos dépendances se veulent à la fois actives (on aide dynamiquement l'utilisateur) et passives (on vérifie à posteriori ce qu'a fait l'utilisateur). Nous n'avons pas voulu nous restreindre à la conception ou l'implémentation puisque nous pensons que ces dépendances existent à la fois dans la conception et l'implémentation. De plus un framework promeut à la fois la réutilisation de l'analyse, de la conception et du code.

Ces dépendances sont fréquentes dans les frameworks et constituent une des difficultés d'utilisation des frameworks blancs. En effet, l'extension de ces frameworks se fait par ajout de méthodes, de classes, de sous-classes... opérations qui, si elles ne respectent pas les règles de validité inhérentes à la structure du framework, entraînent des erreurs à la compilation ou à l'exécution (pour reprendre l'exemple précédent, si la classe dont on hérite n'est pas persistente la sauvegarde indiquera une erreur).

Cette validité, qui fait la difficulté d'utilisation des frameworks, constitue la majeure partie des documentations de ceux-ci, et cela quel que soit le type d'utilisateur. En effet, que l'on soit utilisateur du framework, et que l'on ait besoin d'une documentation liée aux fonctionnalités du framework, ou que l'on soit en charge de la mainte-

nance et que l'on ait besoin de connaître le fonctionnement des points de variation, ce sont toujours les dépendances structurelles que l'on cherche à exprimer ou à connaître. « *Understanding the implementation dependencies between abstract classes and their subclasses is central to the successful use of application frameworks* » [HHG90]. La seule différence existant entre tous les utilisateurs d'un framework est leur niveau d'expertise générale et leur connaissance du framework, et donc les informations auxquelles on peut leur donner accès.

Ainsi dans cet état de l'art nous étudierons à la fois les systèmes de documentation qui présentent les dépendances structurelles sous un aspect fonctionnel[1] qui sont généralement destinées aux utilisateurs d'un framework (développeurs d'applications), ainsi que sous un aspect purement structurel qui peut convenir à la fois aux développeurs et aux mainteneurs du framework. Chacune des techniques présentées sera conclue par ses points positifs et négatifs.

Avant de présenter les techniques nous introduisons un des deux exemples de ce manuscrit : le framework JUnit[2] [BG98]. Cet exemple sera utilisé comme base dans la comparaison des différentes techniques d'expression des dépendances structurelles, ainsi que dans la présentation de notre travail.

2.1 Exemple de framework : JUnit

JUnit est un framework de gestion de tests unitaires pour Java. Il permet à un programmeur de définir des tests et des suites de tests, et lui offre la possibilité de créer des rapports personnalisés. Ce framework et le concept de test unitaire ont été particulièrement diffusés depuis l'apparition de l'eXtreme Programming [Bec99] et ont été déclinés en de nombreux autres langages : C++, Perl, Smalltalk, etc.

JUnit est particulièrement intéressant car c'est un exemple de framework « idéal » : la structure en est décrite par un ensemble de schémas de conception, et un *cookbook*, expliquant comment l'utiliser, est fourni. De plus, la distribution contient un document rarement fourni avec les frameworks détaillant la conception de celui-ci, il s'agit du *JUnit cook's tour* [GB].

Les principaux points de variation de ce framework blanc sont les tests (avec la classe `TestCase` (le schéma de conception Template Method est utilisé pour initialiser le test), la gestion des résultats (`TestResult`), et les suites de tests (`TestSuite`) qui sont implémentées avec le schéma de conception Composite. La figure 2.1 montrent une partie du diagramme de classes de JUnit.

2.2 Critères de comparaison

L'étude des techniques de représentation des dépendances structurelles est effectuée dans l'objectif d'en extraire un modèle unique, modèle servant alors de base à tous les besoins en expression de dépendances structurelles.

Ainsi dans notre étude nous aborderons à la fois notations et outils, en soulignant les utilisateurs à qui ils sont destinés. Comme nous l'avons vu précédemment les do-

[1] On ne montre directement les dépendances, mais les fonctionnalités mises en œuvre par ces dépendances.
[2] http ://www.junit.org.

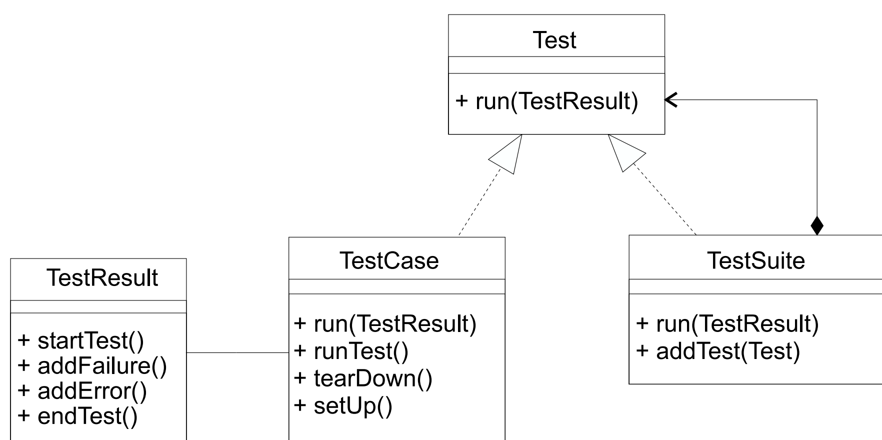


FIG. 2.1 – Diagramme de classes partiel de JUnit.

cumentations des frameworks constituant souvent un fardeau supplémentaire pour le développeur, nous regarderons le degré d'intégration, dans le cycle de vie du framework, des techniques proposées.

Quand bien même la notation serait intégrée au cycle de vie, celle-ci doit être suffisamment modulaire pour permettre une représentation partielle des dépendances du framework. De plus cette modélisation ne doit pas être liée au langage d'implémentation du framework.

Les dépendances étant difficiles à assurer, nous cherchons des outils permettant au choix la génération automatique d'éléments ou la vérification d'un modèle.

2.3 Représentations fonctionnelles

Les outils et notations de cette section ne présentent pas directement les dépendances structurelles, mais les approches qui, en présentant comment ajouter une fonctionnalité, assurent le respect des dépendances structurelles.

2.3.1 *Cookbook* et outils

Le *cookbook* (livre de recettes) a été historiquement le premier moyen de documenter un framework. Il est apparu avec les premières implémentations de Smalltalk, et servait à indiquer à un utilisateur débutant comment il pouvait, par exemple, créer des interfaces graphiques, ou utiliser le modèle MVC. Le *cookbook* offre une approche des dépendances structurelles d'un point de vue purement fonctionnel. Ainsi l'utilisateur cherche la fonctionnalité qu'il veut mettre en œuvre, et se laisse guider par les étapes décrites dans une recette du livre.

Un *cookbook* est spécifique à un framework. Il contient une description textuelle partielle de l'architecture d'un framework, et une manière de l'utiliser pour implémen-

ter une fonctionnalité donnée. Généralement un *cookbook* est fait de recettes et chacune d'elles détaille la mise en œuvre d'une unique fonctionnalité. Ces recettes sont le plus souvent triées selon les types de fonctionnalité qu'elles offrent et donc les parties du framework qu'elles modifient. Une recette est faite d'étapes que l'utilisateur doit suivre afin d'arriver au résultat escompté. Elle contient aussi des informations sur des recettes complémentaires (*related*), ainsi que du code.

La figure 2.2 montre une des recettes livrées avec JUnit. On y retrouve les différentes étapes, ainsi que des exemples de code.

On peut regretter que les *cookbooks* ne puissent pas être exhaustifs dans la description qu'ils font des extensions possibles, et que leur orientation fonctionnelle laisse pour compte les utilisateurs avancés ou les développeurs qui cherchent généralement des informations sur le fonctionnement interne du framework. De plus l'absence de formalisme dans l'expression des étapes rend l'assistance active ou passive difficile.

+ : Description précise d'une utilisation, facilement compréhensible et applicable ; **−** : Étapes difficilement outillables en l'état ; visent principalement les utilisateurs débutants.

2.3.2 *Active cookbooks*

Le concept d'*Active Cookbook* [SSP95, PPSS95] étend l'idée des *cookbooks* en les représentant sous forme d'hypertexte, et en proposant un ensemble d'outils nécessaires à l'adaptation des frameworks. L'outil a pour finalité le guidage des utilisateurs dans l'adaptation du framework : « *Active cookbooks guide programmers and end users through typical framework adaptation steps* » [PPSS95].

Les *active cookbooks* reposent sur un système à base de règles qui sont de deux types :

1. informel, décrivant les interactions entre les composants et le but de certaines méthodes. Selon les auteurs, grâce à cette technique, on arrive à guider complètement l'utilisateur d'une recette ;
2. structurel, décrivant (très paradoxalement) les relations comportementales entre les composants « *structural relations capture the interaction between components* » [PPSS95]. Le but de ces règles est de pourvoir aux besoins d'extensions non-prévues, en fournissant à l'utilisateur une description des interactions entre les composants.

L'article [PPSS95] montre des captures d'écran de l'outil qui permet la sélection d'une recette et présente celle-ci sous forme d'hypertexte. Lorsque l'utilisateur utilise la recette, des outils spécifiques à l'adaptation sont invoqués. Ces outils collectent les informations nécessaires à leur fonctionnement et génèrent le code C++ correspondant à l'extension spécifiée.

Cependant, les informations dont les outils ont besoin ne sont pas clairement explicitées. De plus, on ne sait ni dans quelle mesure la définition de nouveaux outils est nécessaire, ni quelle est la quantité de travail pour créer ses propres règles, ni si l'outil est facilement adaptable à d'autres frameworks.

Simple Test

Case How do you write testing code ?

The simplest way is as an expression in a debugger. You can change debug expressions without recompiling, and you can wait to decide what to write until you have seen the running objects. You can also write test expressions as statements which print to the standard output stream. Both styles of tests are limited because they require human judgment to analyze their results. Also, they don't compose nicely- you can only execute one debug expression at a time and a program with too many print statements causes the dreaded "Scroll Blindness".
JUnit tests do not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do :

1. Create an instance of TestCase :
2. Override the method runTest()
3. When you want to check a value, call assert() and pass a boolean that is true if the test succeeds

For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, write :

```
public void testSimpleAdd() {
   Money m12CHF= new Money(12, "CHF");
   Money m14CHF= new Money(14, "CHF");
   Money expected= new Money(26, "CHF");
   Money result= m12CHF.add(m14CHF);
   assert(expected.equals(result));
}
```

If you want to write a test similar to one you have already written, write a Fixture instead. When you want to run more than one test, create a Suite.

FIG. 2.2 – Exemple de recette livrée avec JUnit.

+ : Guidage de l'utilisateur ;
− : Utilisateur débutant.

2.3.3 *SEA-Preceptor*

Les *active cookbooks* ont inspiré l'outil SEA-Preceptor [SP98]. Cet outil représente graphiquement, grâce à des diagrammes de classes et de séquences, un framework. L'outil qui offre un support sémantique d'édition assure la cohérence de son framework à l'aide d'un méta-modèle. Cet outil offre deux facettes : une pour le développeur du framework, et l'autre pour l'utilisateur du framework (le développeur d'applications).

Bien que cette approche semble prometteuse, nous n'avons pas été en mesure d'en savoir plus, car aucune autre publication (en anglais ou en français) ne parle de cet outil, ni ne détaille le méta-modèle.

Dans le même style d'outils, on peut remarquer *Smartbooks* [OCM00] qui, basé sur un planificateur, offre plus de flexibilité dans la réalisation des étapes d'une recette d'un *cookbook*.

2.3.4 Documentation de frameworks par des *patterns*

La technique décrite par Johnson dans « *Documenting frameworks using patterns* » [Joh92] fournit un moyen de documenter de manière structurée (à la manière d'un language de *patterns*) un framework et ses grandes fonctionnalités.

Cette approche propose une technique de documentation constituée de *patterns* décrivant par raffinements successifs les fonctionnalités du framework. Ici, le mot *pattern* n'est pas utilisé dans le sens de schéma de conception, mais simplement comme un élément de structuration qui ressemblerait plus à une recette d'un *cookbook*.

Ainsi dans l'article [Joh92] qui présente cette technique de documentation, l'auteur donne 10 *patterns* décrivant le framework graphique HotDraw [Bra95], l'ajout des figures, le changement d'attributs d'une figure, etc.

La figure 2.3 présente les *patterns* que nous avons imaginés et qui pourraient documenter JUnit. Nous avons gardé une approche par raffinement : le premier *pattern* décrit le but du framework, les deuxième, troisième et quatrième expliquent comment ajouter certaines fonctionnalités et le cinquième donne quelques détails techniques pour améliorer la création de tests.

L'approche présentée dans [MCK97] propose une solution similaire d'organisation hiérarchique de la documentation. Cette organisation est de forme pyramidale : ainsi, au sommet de la pyramide, on trouve des données très générales sur le framework, qui sont détaillées grâce à des *patterns* (au sens de Johnson dans l'approche précédente), des schémas de conception et des tutoriaux dans les niveaux inférieurs.

Toujours dans la même direction, on notera les travaux d'Aguiar [Agu00] qui proposent l'utilisation de l'approche minimaliste de Carroll [Car90] pour la documentation d'un framework : les documents sont présentés à l'utilisateur au fur et à mesure de ses besoins.

Ces travaux ont en commun l'organisation de la documentation par raffinement et n'offrent en terme d'implémentation que des systèmes de navigation, mais pas d'assistance qu'elle soit génératrice ou vérificatrice.

Pattern 1 : Gestion de tests unitaires.
JUnit est un framework facilitant la gestion de tests unitaires. Il peut être utilisé pour tester n'importe quelle classe Java.
JUnit offre des fonctionnalités pour gérer des tests, des suites de tests, ainsi que faire des rapports....

Pattern 2 : Création d'un test.
Une infinité de tests peuvent être écrits pour des sujets très variés, mais tous ont besoin d'initialiser leurs valeurs.
Afin de créer un nouveau test l'utilisateur doit sous-classer la classe `TestCase`, surcharger les méthodes `setUp` et `tearDown` pour initialiser des variables, et enfin écrire une méthode contenant le test. Le but de la méthode `setUp` est d'initialiser des variables d'instances, celui de la méthode `tearDown` est de désinitialiser ces variables. ...

```
public class MathTest extends TestCase {
   protected double fValue1;
   protected double fValue2;
   public MathTest(String name) {
     super(name);
   }
   protected void setUp() {
     fValue1= 2.0;
     fValue2= 3.0;
   }
}
```

Pour éviter la prolifération des sous-classes voir pattern 4.

Pattern 3 : Création d'une suite de tests.
Généralement un test ne suffit pas à couvrir tous les cas d'une classe ou d'une application. Ainsi, plusieurs tests doivent être créés, et peuvent être organisés dans une suite de tests...

Pattern 4 : Gestion des résultats. ...

Pattern 5 : Création de tests anonymes. ...

FIG. 2.3 – Exemple d'une organisation possible des *patterns* décrivant JUnit.

+ : Présentation hiérarchisée de la documentation ;
− : Système de navigation.

2.3.5 Les *Hooks*

Les *Hooks* [FHLS97], en français crochets, servent eux aussi à la description de l'utilisation / adaptation d'un framework. À la manière d'une recette d'un *cookbook*, un *Hook* décrit les étapes à suivre pour adapter un framework, mais y ajoute un côté plus formel. Il indique précisément sous forme d'un algorithme les modifications à apporter et ainsi force l'utilisateur à se soumettre aux contraintes liées à l'extension du framework.

Les *Hooks* sont structurés. Un *Hook* est constitué d'un nom (`Name`), d'une partie `requirement` décrivant sous forme textuelle ce que fait le *Hook*, du type d'adaptation utilisé (activation, désactivation, remplacement, ajout, extension...), des parties du framework touchées en terme de fonctionnalité (`Area`), des composants qui participent au *Hook* (`Participants`), de la description des changements (`Changes`), de contraintes (`Constraints`)[3], et de commentaires (`Comments`). La figure 2.4 montre un tel *Hook* pour l'utilisation de JUnit.

Les *Hooks* facilitent l'utilisation d'un framework car les utilisateurs n'ont pas besoin de connaître à la perfection tous les rouages du framework puisque les étapes de la réalisation d'une recette sont détaillées. Ainsi les *Hooks* s'adressent plus particulièrement aux utilisateurs de framework (pour développer une application) qu'aux développeurs. Le problème de ces *Hooks* est que leur écriture, qui doit être faite par les concepteurs du framework, demande une phase supplémentaire lors du développement qui doit être coûteuse vu la complexité de chacun de ces *Hooks*.

Un outil mettant en œuvre les *Hooks* semble exister [FHLS98]. Cet outil interprète les *Hooks* et permet à l'utilisateur une visualisation de son framework sous la forme de diagramme de classes ou de séquences. Il permet l'adaptation du framework sur activation d'un *Hook*.

+ : Description algorithmique de l'application d'un point de variation ;
− : Requiert la production d'un documentation particulière.

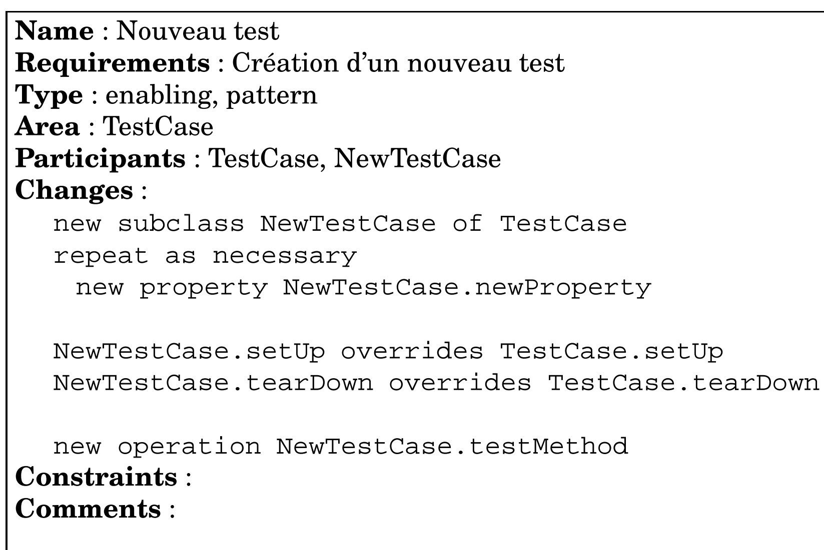
2.4 Représentations structurelles

Les outils et notations de cette section présentent des techniques de représentation structurelles qui décrivent directement les dépendances par opposition aux précédentes qui les masquaient sous un aspect fonctionnel.

2.4.1 Schémas de conception

Les schémas de conception constituent un excellent moyen de documenter un framework [BJ94], ainsi que d'en véhiculer en quelques mots des parties de la conception.

[3]Les contraintes sont exprimées en langage naturel, et ne sont pas vérifiables.


Name : Nouveau test
Requirements : Création d'un nouveau test
Type : enabling, pattern
Area : TestCase
Participants : TestCase, NewTestCase
Changes :

```
new subclass NewTestCase of TestCase
repeat as necessary
  new property NewTestCase.newProperty

NewTestCase.setUp overrides TestCase.setUp
NewTestCase.tearDown overrides TestCase.tearDown

new operation NewTestCase.testMethod
```

Constraints :
Comments :

FIG. 2.4 – *Hooks* décrivant la création d'un test pour JUnit.

Dans notre contexte ils permettent de décrire les dépendances entre les éléments. Malheureusement cette approche est limitée puisque toutes les dépendances ne peuvent pas être modélisées avec les schémas de conception (en particulier celles qui sont liées à l'application [BJ94]). De plus, on peut remarquer, sur la figure 2.5 qui montre l'organisation des schémas de conception dans JUnit, l'inadéquation de la solution avec l'expression des points de variation puisque les auteurs ont été obligés d'introduire une classe qui n'appartient initialement pas au framework pour montrer le schéma de conception Adapter. Les schémas de conception posent d'autres problèmes tels le nommage contagieux des entités (la mise en œuvre de certains schémas implique que le nom de certaines méthodes ou classes soit fixé) et leur dissolution dans le code et la conception.

On peut déplorer que des diagrammes comme celui de la figure 2.5 soient rarement fournis avec un framework. On notera que les auteurs ont recours à une notation graphique spéciale pour montrer le rôle de chaque classe dans les schémas de conception. Une notation équivalente, dite de collaboration [OMG01b, p 3-124], existe en UML (figure 2.6).

+ : Représentation de l'architecture interne du framework ;
− : Inadéquat pour représenter tous les points de variation.

2.4.2 *Meta patterns*

Les *Meta patterns* [Pre94, Pre99] ont été introduits afin de modéliser plus finement les points de variation d'un framework. Ainsi, après une analyse des schémas de conception et de quelques frameworks l'auteur a fait ressortir deux constructions

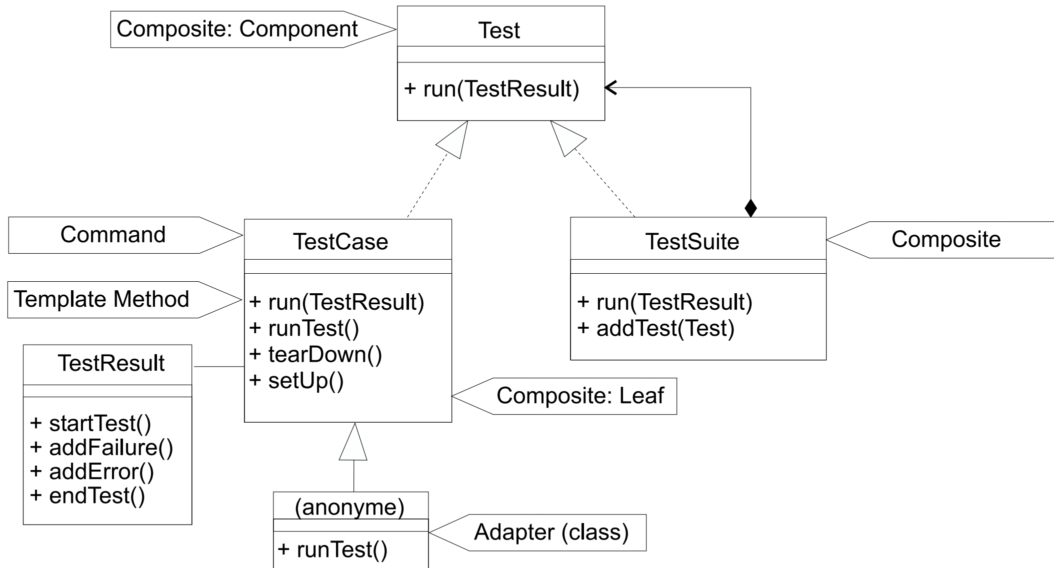


FIG. 2.5 – Les schémas de conception de JUnit, extrait de [GB].

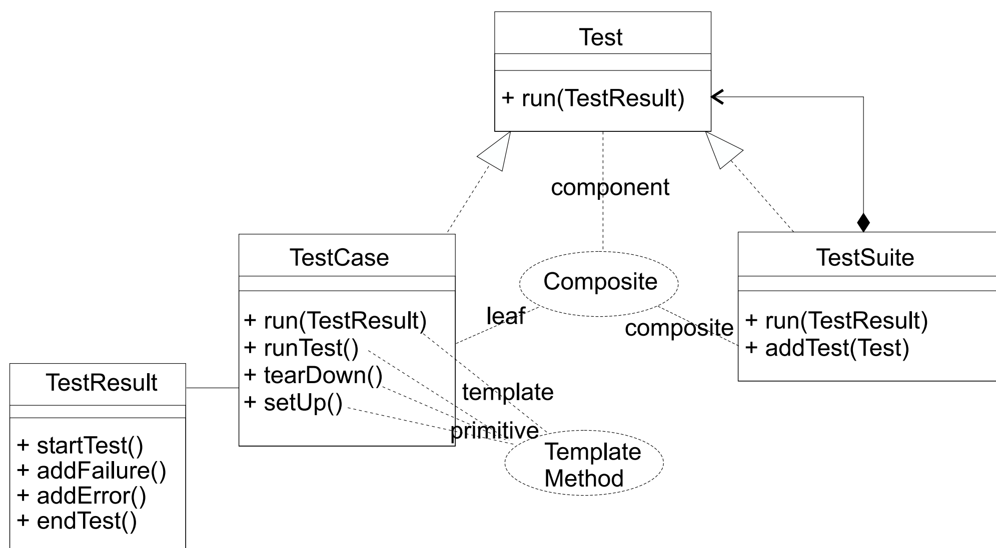


FIG. 2.6 – Représentation des schémas de conception de JUnit en UML.

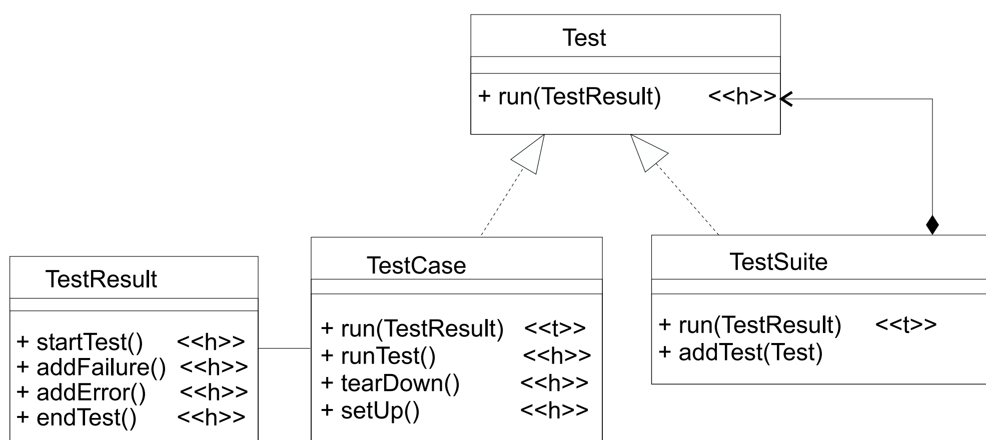


FIG. 2.7 – Utilisation des *Meta patterns* pour représenter les points de variations de JUnit.

essentielles permettant de construire toutes les possibilités d'adaptation. Il s'agit du `template` et du `hook`. Ces deux notions qui indiquent le rôle que joue une méthode varient selon le point de vue avec lequel on l'envisage. `template` indique une méthode qui appelle une autre méthode, et `hook` représente une méthode qui est redéfinie, la redéfinition pouvant être une méthode `template`. Une méthode `template` appelle généralement des méthodes `hooks`. La classe `template` et la classe `hook` sont des classes qui réciproquement contiennent une méthode `template` et une méthode `hook`.

À partir de ces deux constructions, l'auteur construit des *Meta patterns*[4], complétant les schémas de conception du GoF, qui représentent tous les axes de flexibilité.

Puisque l'un des buts des *Meta patterns* est l'amélioration de la visualisation des rôles d'un élément dans un framework, l'auteur propose d'annoter d'un `t` (`template`) ou d'un `h` (`hooks`) l'élément concerné. La figure 2.7 montre l'utilisation de la notation pour la documentation de JUnit.

Cette notation explicite le rôle de chacune des méthodes et des classes. Cependant nous pensons qu'elle est d'une granularité trop fine et qu'elle ne capture pas vraiment les dépendances structurelles. De plus elle n'est pas facile à comprendre pour un utilisateur novice, et s'adresse plus particulièrement à un développeur de frameworks. Comme on le voit sur la figure 2.7 on ne sait pas quel `template` utilise quel `hook`, et l'on peut imaginer le désordre engendré si une même méthode était à la fois `hook` et `template`.

+ : Rôle de tous les éléments ;
− : Ne présente pas vraiment les dépendances, peu lisible.

[4]Initialement il y en avait 7, mais ils ont été réduit à 5.

2.4.3 UML-F

UML-F est la suite des *Meta patterns* puisqu'il permet de représenter dans des diagrammes UML (principalement classes et séquences) les points de variation. Il comble ainsi l'impossibilité d'exprimer en UML les contraintes d'utilisation d'un framework. Ainsi grâce à UML-F, on peut voir les classes qui jouent un rôle dans les schémas de conception, les nouvelles méthodes, les méthodes héritées, redéfinies, etc.

UML-F [FPR00b, FPR00a] se présente sous la forme d'un profil (*profile*) UML, et fournit de nombreux stéréotypes textuels ou graphiques pour préciser la structure des frameworks, et plus facilement appréhender certains points de conception. Ce profil offre aussi le bénéfice d'être extensible par composition des éléments existants.

Ainsi, dans le tutorial [FPR00a] il est expliqué et même conseillé de créer ses propres marqueurs (*tags*) pour représenter les schémas de conception ou des éléments liés aux domaines d'application du framework. Cette création de marqueurs est faite par composition des marqueurs de base comprenant entre autre le `hook` et le `template` des *Meta patterns*. Des exemples de ces marqueurs sont présentés figure 2.8 : les marqueurs commençant par un `C-` marquent les éléments participant au schéma de conception Composite, les marqueurs commençant par `Template-` au schéma de conception Template Method, et les marqueurs commençant par `Strategy-` au schéma de conception Strategy. Un utilisateur familier du schéma de conception Template Method notera qu'il manque la partie concrète de ce schéma implémentant les `Template-PrimitiveOp()`. Cela sera normalement fait dans une sous-classe de `TestCase`.

Nous pensons qu'à terme cette prolifération de marqueurs entraîne une perte du bénéfice de la notation (tout comme la prolifération des schémas de conception entraîne une perte des bénéfices de ceux-ci [AC98]). De plus alors que certains marqueurs sont liés à des aspects applicatifs, nous pensons que certains, comme ceux qui indiquent si toutes les méthodes de la hiérarchie sont montrées, sont plus pertinents dans le contexte d'un outil que pour la modélisation proprement dite des points de variation.

Bien que nous n'en montrions pas ici, des contraintes OCL peuvent être utilisées pour indiquer certaines propriétés sur les éléments qui vont être créés. On regrettera l'absence d'opérationalisation de ce système.

« *The most important claims of this paper is that frameworks should be modeled through appropriate design constructs that allow the representation of variation points and their intended behavior* »[FPR00b]

+ : Notation simple ;
– : Prolifération de stéréotypes.

2.4.4 FRED et les *specialisation patterns*

Le concept de *specialisation pattern* présenté dans [HHK+01] et implémenté dans un outil nommé FRED[5] est une technique d'une granularité similaire aux *Hooks* (sous-section 2.3.5) et *cookbooks* (sous-section 2.3.1) pour décrire comment étendre un framework. Les *specialisation patterns* sont utilisés par FRED pour guider l'utilisation du framework de manière plus souple qu'un mode strictement pas à pas. Ils s'adressent

[5] http ://practise.cs.tut.fi/fred/.

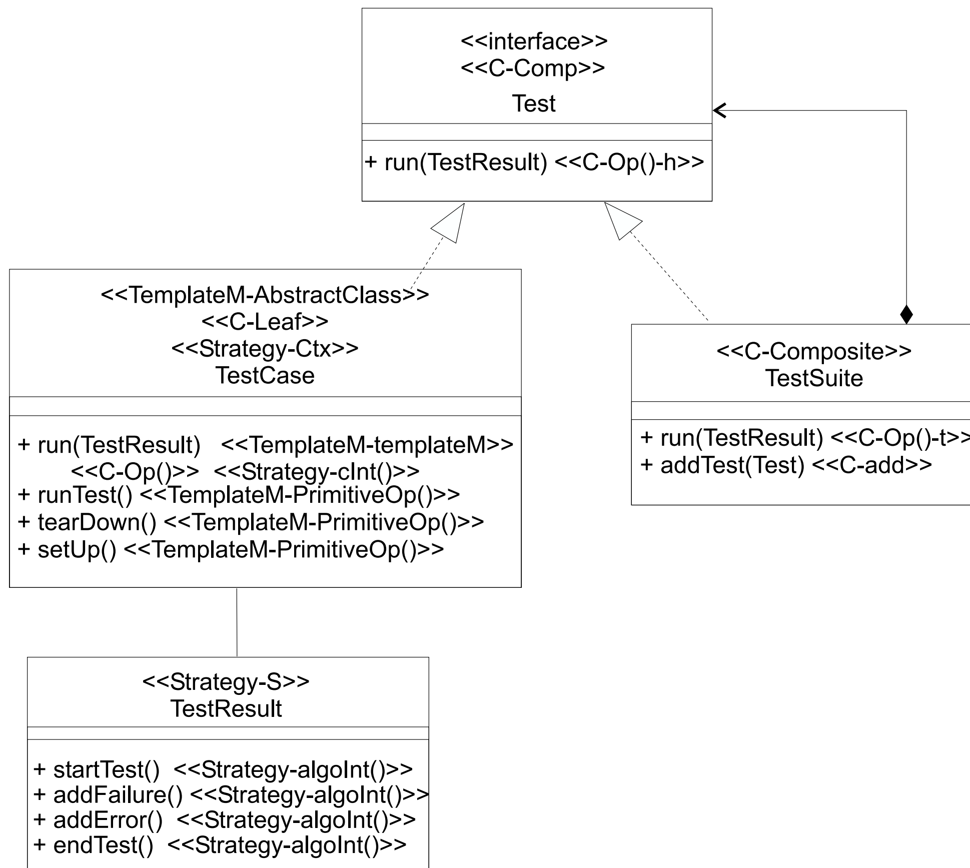


FIG. 2.8 – Représentation de JUnit en UML-F, recomposé à partir de [FPR00a].

Nouveau Test		
Bound roles	*Properties*	
TestCase : class	description	Classe de base à tous les tests.
runTest : method	description	Lance le test.
setUp : method	description	Initialise les données du test.
tearDown : method	description	Désinitialise les données.
UnBound Roles		
MyTestCase+ : class	description	Classe spécifique à chaque test.
	title	Donner un nom à la nouvelle classe de test sous-classe de <#TestCase>.
	inheritance	<#TestCase>
newTest : method	task title	Donner un nom à la méthode de test.
newField : variable	task title	Donner un nom au champ
setUp : method	task title	Changer le code de la méthode <#setUp> pour surcharger <#TestCase.setUp>
	overriding	<#TestCase.tearDown>
tearDown : method	task title	Changer le code de la méthode <#tearDown> pour surcharger <#TestCase.tearDown>
	overriding	<#TestCase.tearDown>

FIG. 2.9 – Exemple d'un *specialisation pattern* pour JUnit.

donc en priorité aux utilisateurs de frameworks, mais offrent une approche plus structurelle que fonctionnelle si on les compare aux *Hooks* auxquels ils sont très semblables.

Un *specialisation pattern* contient la spécification d'une structure récurrente d'un programme. Idéalement il n'est pas lié à un framework, mais les auteurs indiquent qu'il est préférable de le particulariser afin d'en tirer un plus grand bénéfice pour l'assistance.

Un *specialisation pattern* est décrit en terme de rôles (`Bound Roles` et `Unbound Roles`), rôles que des éléments du framework devront jouer. Lorsqu'un élément joue un rôle, on dit qu'il est en contrat avec celui-ci. Lorsqu'un contrat est passé, il entraîne généralement la modification du code source de l'élément selon les règles qui régissent le rôle. Par exemple, pour le rôle `MyTestCase+` présenté figure 2.9, le passage de contrat devra être fait avec une classe (indication `class` après le nom du rôle), et celle-ci devra hériter de la classe qui joue le rôle de TestCase (`inheritance <#TestCase>`).

Il est à noter que des relations peuvent être posées entre les rôles afin d'exprimer une dépendance. « *It is also equally necessary to define mutual relationships between the different parts of the specialization, an important aspect often overlooked* » [HHK+01].

On peut regretter que cette approche incrémentale, orientée génération de code soit trop liée à un langage. De plus la création d'un *specialisation pattern* requiert un travail supplémentaire au développeur.

Il est à noter que cette approche est la suite des travaux sur les *specialisation template* [HHT+99], eux-mêmes une extension / encapsulation des contrats de Holland [Hol92] d'où les aspects comportementaux ont été retirés.

+ : Détail de l'architecture, possibilité de le découpler d'une application ;
− : Requiert la production d'un documentation particulière.

2.5 Techniques non dédiées aux frameworks

Cette section recense brièvement un ensemble de techniques pouvant s'apparenter à l'expression des dépendances structurelles mais qui ne sont pas spécifiques aux frameworks.

2.5.1 *Law-governed regularities*

Les travaux intitulés *Law-governed Architecture* (LGA, Architecture gouvernée par des lois) [Min96] ont pour but d'exprimer des contraintes de validité globale sur les éléments d'un système. L'approche associe donc un ensemble de règles explicites à la totalité d'un projet. Ces règles forment les lois du projet qui devront être respectées tout au long de celui-ci. Au-delà des aspects conception qui nous intéressent, l'approche gère aussi des aspects génie logiciel.

Les lois gèrent à la fois l'aspect structurel et comportemental du projet. Ainsi, on peut vérifier à la compilation si la structure de l'application est correcte, et à l'exécution si les composants de l'application communiquent bien ensemble.

L'approche est indépendante du langage, cependant il est regrettable qu'elle ne fasse que de la vérification et que celle-ci ne soit faite qu'à la compilation ou à l'exécution.

+ : Flexibilité du concept ;
− : Pas d'assistance active, vérification à la compilation.

2.5.2 *Logic meta programming*

Les travaux « *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation* » de Roel Wuyts [Wuy01], proposent une technique de méta-programmation déclarative pour conserver la synchronisation entre une conception et son implémentation. Le système mis en place repose sur deux frameworks mis au point par l'auteur. Un premier framework permet d'accéder aux objets du langage à travers une API à la Prolog, et un second gère la synchronisation entre deux modèles en utilisant le premier framework. Les règles de synchronisation sont aussi écrites dans un style Prolog.

Au regard des travaux, nous pensons que ce dernier framework de synchronisation pourrait être utilisé pour gérer les dépendances structurelles d'un framework. On peut regretter que cette approche soit trop liée à Smalltalk et plus généralement aux langages de classes.

2.5.3 Outils dédiés aux schémas de conception

Les outils pour les schémas de conception résolvent un problème similaire à celui que nous considérons. En effet, pour supporter des opérations comme l'instanciation,

ou la vérification ils représentent les dépendances entre les différents éléments d'un schéma de conception. Nous présentons très succinctement les approches qui ont retenu notre attention et laissons au lecteur le soin de consulter [BR99] pour une description détaillée et une comparaison des outils mettant en œuvre les schémas de conception.

Ainsi on notera la flexibilité de l'approche décrite dans [FMvW97]. Cette approche propose une représentation des schémas de conception en utilisant un ensemble de fragments connectés par des rôles. Un fragment est un morceau de programme représentant méthodes, classes, relations, etc. On peut regretter que le type des fragments soit fixé, et que la représentation du schéma de conception en fragment ne soit pas abstraite et se contente de recopier un des exemples du livre du GOF [GHJV95], ce qui donne une représentation particulièrement volumineuse.

Nous retiendrons aussi les approches basées sur une méta-représentation du schéma de conception [SLJ00, RF00], ou un méta-langage de manipulation [EYG97]. Ainsi les deux premières approches proposent une solution où les éléments du schéma sont réifiés, alors que la troisième décrit l'application d'un schéma de conception par un algorithme. L'avantage de telles approches est leur souplesse, cependant on regrette là encore le lien trop fort à un langage donné, et la nécessité d'une représentation spécifique dont la réalisation est coûteuse en temps.

2.5.4 *Design constraints*

Les travaux de Klarlund [KKS96] sur les *design constraints* (contraintes de conception) utilisent une variation de la logique du premier ordre afin de vérifier l'utilisation d'un style architectural et de vérifier des contraintes particulières à une application. Ces contraintes, décrites en Category Description Language, expriment des contraintes sur les arbres de syntaxe abstraite de l'application à vérifier. Afin d'indiquer les contraintes à appliquer, le code source de l'application (IDL dans la publication) a besoin d'être annoté. On peut regretter que le système ne fournisse que la vérification et la nécessité de marquer les endroits à vérifier, impliquant une extension de la syntaxe du langage utilisé.

+	: Approche descriptive ;
−	: Marquage des éléments.

2.6 Récapitulatif des techniques d'expression des dépendances structurelles

Avant de présenter notre solution pour l'expression des dépendances structurelles, nous synthétisons les techniques présentées dans les sections précédentes dans le tableau 2.1.

Il existe un besoin commun à toutes ces techniques qui est, de manière directe ou indirecte et à une granularité plus ou moins fine, d'exprimer les dépendances structurelles. Notre idée n'est pas de refaire un outil d'assistance à l'utilisation des *cookbooks*, ou de développer une notation à la *Hook*, mais de fournir un modèle d'expression des

	Représentation des points de variation	Type de description	Niveau de description	Génération	Vérification	Intégration cycle de vie	Type d'utilisateur
Cookbook	★★	Txt	Fwk	-	-	✗	U
Outils *Cookbook*	★★	?	?	✓	✓✗	✗	U
Hooks	★★	Algo	Fwk/Méta	✓	✗	✗	U/M
Schémas de conception	★	Decl	Fwk	-	-	✓	D/M
Meta patterns	★★★	Decl	Fwk	-	-	✗	A/D/M
UML-F	★★★	Decl	Fwk	-	-	✓	A/D
Specialisation pattern	★★	Algo	Fwk/Méta	✓	✗	✗	A/D/M

✓ : oui Decl : Déclaratif Méta : méta D : Développeur
✗ : non Txt : Textuel U : Utilisateur M : Mainteneur
Algo : Algorithmique Fwk : framework A : Utilisateur avancé - : non pertinent
? : inconnu ★...★★★ : Évaluation subjective du critère

TAB. 2.1 – Comparatif des techniques d'expression des dépendances structurelles.

dépendances qui pourrait servir de base à ces outils, et pouvant fournir de l'aide à toutes les catégories d'utilisateurs.

Ce modèle devra donc permettre l'expression des dépendances structurelles à différents niveaux d'abstraction (du fonctionnel au structurel), tout en offrant une possibilité d'introduction des dépendances dès la conception afin d'éviter l'écueil d'une documentation supplémentaire fastidieuse à réaliser. De plus ce modèle devra être découplé de tout langage voire du concept de langage de programmation afin de pouvoir exprimer des dépendances entre tout type d'éléments. Afin d'offrir un maximum de souplesse, les dépendances seront exprimées de manière déclarative entre des éléments réifiés. Bien sûr, le modèle devra offrir comme fonctionnalité de base la génération d'éléments et la vérification de la consistance des dépendances.

Problématique des dépendances structurelles

En regardant le type d'aide que fournissent les outils présentés dans l'état de l'art, on constate que le support de représentation des dépendances structurelles n'est pas nécessairement adapté à l'utilisateur par rapport à son niveau de connaissance du framework et que de surcroît aucun système ne convient à toutes catégories d'utilisateurs.

Ainsi, pour les débutants, les *cookbooks* facilitent l'utilisation du framework par fonctionnalité (but), mais ils ne permettent pas de prendre conscience des dépendances sous-jacentes, ni même de l'architecture du framework, à moins de consulter le code source.

Pour un utilisateur averti (familier du framework), les *cookbooks*, qui offrent un type d'aide plutôt fonctionnel, ne suffisent pas. Cette catégorie d'utilisateurs étend le framework sans suivre de besoins fonctionnels et n'est donc guidée par aucune démarche comme le fait un *cookbook*. Cela peut facilement conduire à une mauvaise utilisation du framework. Cependant, un utilisateur averti peut bénéficier d'assistance avec des mécanismes tels les *Hooks* [FHLS97] qui mettent en avant l'aspect structurel du framework en documentant les points de variation. Cependant les *Hooks*, tout comme les *cookbooks* ou les outils dédiés à l'extension d'un framework, nécessitent un travail supplémentaire de la part du développeur du framework qui doit créer l'information qui servira de base aux outils.

L'expert ne se contente pas seulement d'étendre le framework aux points de variation, il modifie généralement les classes du cœur du framework. Cet utilisateur a pour seule aide, s'ils sont disponibles, les documents de conception et les commentaires du code source puisque les outils se limitent à la documentation des points de variation.

Afin d'aider au mieux toutes les catégories d'utilisateurs du framework tout en minimisant le travail du développeur, nous présentons un modèle générique d'expression des dépendances structurelles. L'objectif de celui-ci est de permettre au développeur du framework d'expliciter les dépendances entre les entités du framework, et cela dès la conception, de telle sorte qu'à l'utilisation du framework une assistance sous forme d'un *wizzard*, d'une note de conception à la manière ArgoUML [arg], voire même la génération d'éléments soit fournie. Cela n'exclut pas les supports sous une forme plus fonctionnelle tels les *active cookbooks*.

Ainsi le chapitre suivant présente le fonctionnement général de notre modèle d'expression des dépendances structurelles avant d'en donner les détails techniques.

Le chapitre 4 présente comment ces dépendances, pouvant constituer des bibliothèques, sont utilisées pour faciliter le développement de frameworks. De plus, avant de faire l'esquisse d'un outil, il montre l'intégration de ce mécanisme de « documentation » au sein du cycle de vie du framework et les implications sur les différents utilisateurs.

Chapitre 3

Modèle d'expression des dépendances structurelles

Comme l'affirme Butler et Dénommée [BD99], « *the framework insists that the customization preserves a protocol of collaboration between several subclasses, so customization requires the parallel development of these subclasses and certain of their methods* ». Ces dépendances, inhérentes aux frameworks, abondent à la fois dans les frameworks usuels, et à un grain plus fin dans les schémas de conception. Par exemple, le framework JavaBeans requiert qu'à chaque composant soit associé un composant de description [Mic97b][1]. De la même manière le schéma de conception Abstract Factory requiert, pour gérer un nouveau produit, la création de plusieurs classes et de méthodes [RF00]. Deux sources de dépendances sont donc identifiables : une première liée au contexte applicatif du framework et une seconde liée à des considérations purement architecturales (ex : utilisation d'un schéma de conception). La validité de la structure d'un framework une fois étendu (i.e. une application dérivée) n'est donc garantie que si toutes les dépendances structurelles le constituant sont préservées. Cependant cette notion de validité est différente pour chaque framework et il paraît impossible d'établir une base exhaustive des dépendances communes à tous les frameworks.

Lors du développement d'une application à partir d'un framework, la détection précoce d'incohérences dans la structure est d'autant plus cruciale que la compilation de l'application ou l'exécution d'une séquence de tests peut être longue. Nous présentons donc un modèle d'expression des dépendances structurelles permettant de guider l'utilisateur (vérification, création automatique d'éléments) du framework afin qu'il en respecte l'esprit (le style architectural) et surtout la structure lors de l'écriture d'une application dérivée. Ce modèle, par sa flexibilité, et ses fonctionnalités peut aussi servir de base à des systèmes comme les *Hooks* ou les *Active cookbooks*.

Les informations concernant les dépendances structurelles sont détenues par le concepteur et/ou le développeur du framework. Il sait, alors même qu'il conçoit / met

[1] La spécification des JavaBeans indique qu'à un composant `Bean` peut être associé une classe `BeanInfo` contenant une description explicite du composant. Cette classe doit avoir pour nom, le nom du composant suffixé par BeanInfo (par exemple le composant bean Display devra avoir une classe DisplayBeanInfo).

en œuvre son framework, où se trouvent les points de variation et comment ils devront être utilisés, quels sont les schémas de conception utilisés [Joh92], ou encore les autres contraintes entre les éléments. Afin d'éviter la perte de ces informations, de réduire la création de documents spécifiques, et de faciliter l'utilisation du framework, nous proposons que toutes ces informations soient conservées avec le framework de manière à être utilisées par un système d'assistance. Cela constitue un premier pas pour combler le manque de continuité dans la documentation entre les développeurs et les réutilisateurs du framework.

3.1 Présentation générale de notre modèle d'expression des dépendances structurelles

Afin d'introduire le modèle et ses fonctionnalités sans entrer dans les détails, nous présentons un cas d'utilisation de celui-ci pour l'exemple des JavaBeans. Ainsi, on aimerait qu'un utilisateur du framework des JavaBeans respecte le fait qu'un composant Bean soit toujours accompagné d'une classe de description BeanInfo.

Dans notre modèle, l'expression de cette dépendance peut être faite de deux manières complémentaires :

– sous la forme d'une règle de création* indiquant la nécessaire création d'un élément ;
– sous la forme d'une assertion (à la manière de Klarlund [KKS96]) vérifiant qu'à tout composant Bean est associé un BeanInfo (on parlera encore de règle de vérification*).

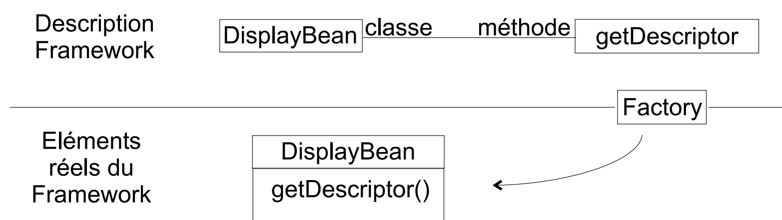
Ces deux expressions permettent à un utilisateur du framework d'avoir un support environnemental [BR99] dont les fonctionnalités sont :

– un système d'assistance actif alors qu'il étend le framework afin de créer une application. Cette assistance active sera aussi désignée sous le nom de *réalisation* ;
– un système passif de vérification à utiliser une fois l'extension du framework terminée (le système actif pouvant avoir été utilisé ou non). On parlera encore de *vérification* ;
– une approche mixte permettant de vérifier à n'importe quel moment la validité d'une application construite à partir du framework.

La réalisation et la vérification utilisent respectivement les règles de création et les règles de vérification. Dans notre exemple les règles de création indiquent sous une forme cause / conséquence que *la création d'un nouveau Bean doit créer un nouveau BeanInfo* et les règles de vérification décrivent la validité du système en indiquant que *le nombre de Bean et le nombre de BeanInfo doit être le même*[2].

Bien que le contenu des règles de création et de vérification soit corrélé, il est difficile à notre connaissance que l'une soit automatiquement déduite de l'autre. Ce qui fait la différence entre une règle de création et une assertion est leur objectif. L'une désigne un processus de création qui est une opérationnalisation de l'autre. Cette opérationnalisation est nécessaire car il n'est pas évident de savoir à partir d'une assertion comment celle-ci doit être mise en œuvre afin d'assurer la validité du système tout au

[2]Cette contrainte est nécessaire mais pas suffisante.



La flèche symbolise que la Factory a généré `DisplayBean`.

FIG. 3.1 – Description d'un framework.

long de son évolution [RR01]. De plus, l'une et l'autre peuvent être utilisées de manière complémentaire et ne pas nécessairement décrire la même situation. Le problème de cohérence entre les règles de création et de vérification ne sera pas abordé dans ces travaux.

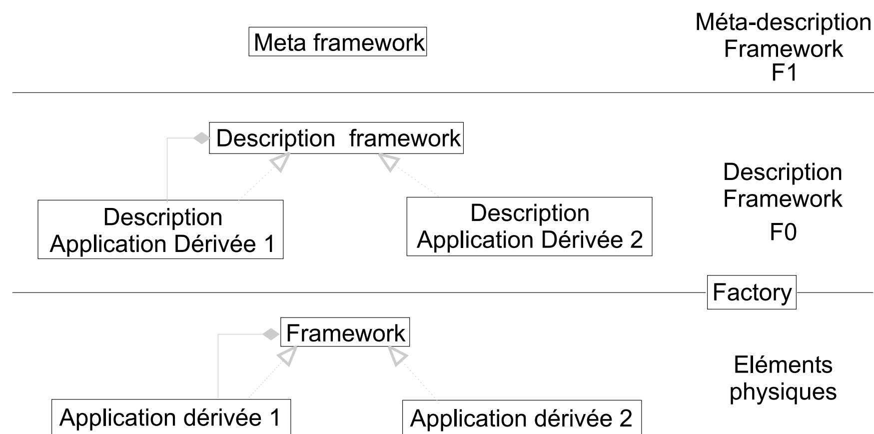
3.1.1 Représentation flexible d'un framework

Afin d'exprimer les dépendances, nous ne nous sommes pas focalisés sur la création d'une notation, mais sur la mise en œuvre d'un modèle flexible et extensible offrant les fonctionnalités de réalisation et de vérification d'un framework.

L'une des solutions aurait pu consister en la création d'un modèle spécifique aux frameworks orientés objets, lié à un langage comme le propose l'outil FRED [HHK+01], ou lié à une classe de langages comme les *Hooks* [FHLS97]. Cependant, nous avons décidé de nous abstraire de tout lien vers un langage d'implémentation du framework (que ce soit UML, Java...) mais aussi d'un type de langage de programmation afin de permettre la représentation des dépendances entre toute sorte d'éléments, à la manière de LGA [Min96]. Cette indépendance vis à vis du langage nous a semblé essentielle car elle permet de travailler aussi bien sur des frameworks écrit dans des langages procéduraux, que de traiter des frameworks qui manipulent des scripts, des fichiers XML ou encore des bases de données. Tel est le cas dans Eclipse [OTI01] ou dans les EJBs [Mic] où des informations de configuration sont conservées dans des fichiers XML.

Afin d'obtenir la flexibilité recherchée, nous avons choisi de ne pas directement manipuler les constituants d'un framework mais une description de ceux-ci. Cette description du framework* le représente comme un ensemble d'entités, appelées aussi descripteurs*, connectées les unes aux autres par des relations indiquant leur rôle à la manière d'un graphe conceptuel, ou comme le sont les fragments utilisés dans « *Tool Support for Object-Oriented Patterns* » [FMvW97]. Ainsi, cette description du framework, qui ne contient que des éléments existant dans le framework représenté, lie par exemple un élément représentant une méthode et un autre représentant une classe par une relation `"classe / méthode"` (figure 3.1). De plus, cette description contiendra aussi des informations comme le fait qu'un *BeanInfo donné est celui d'un Bean identifié*. L'avantage de ce type de représentation est qu'elle permet de décrire le framework à n'importe quel niveau de précision d'une recette d'un *cookbook* à un morceau de code.

La connexion entre cette description du framework et ses constituants réels est faite grâce à un composant spécifique qui porte le nom de Factory.



Les flèches représentent symboliquement les liens possibles entre les éléments.

FIG. 3.2 – Méta-description, description et applications dérivées d'un framework.

3.1.2 Réification d'un framework et de ses dépendances

Au regard de l'exemple des JavaBeans, on constate que la dépendance entre Bean et BeanInfo met en jeu des éléments qui ne sont pas créés. En effet, la dépendance n'existe pas directement entre `DisplayBean` et `DisplayBeanInfo` mais entre toutes les classes qui sont des Beans et des BeanInfos : une réification de Bean et BeanInfo. Ce besoin de réification n'est pas nouveau et bien que non explicité de cette manière il était déjà présent dans les *Hooks* (exemple : `NewTestCase.subclass` (figure 2.4)).

Nous avons donc choisi de modéliser les dépendances (règles de création et règles de vérification) dans un niveau de méta-description. Ainsi les dépendances portent sur une réification du framework et assurent, en contrôlant les opérations effectuées sur le framework, la correction de l'application dérivée à partir de celui-ci. Au fur et à mesure de sa création, la description de cette application dérivée sera ajoutée aux côtés de celle du framework (figure 3.2).

Puisque nous ne travaillons pas directement sur le framework mais sur une description, ce niveau de réification constitue donc une méta-description du framework*, encore appelé méta-framework* (niveau F1 sur la figure 3.2 ou 3.3). Ce niveau est en fait une représentation conceptuelle* du framework puisqu'il en décrit certains concepts qui n'ont pas nécessairement de représentant au niveau framework. Tout comme les constituants de la description du framework, les éléments conceptuels*, constituants de cette réification, sont connectés par des rôles (figure 3.3).

Par abus de langage nous appellerons framework la description du framework (niveau F0 sur la figure 3.3) et de l'application dérivée. Nous identifierons par framework initial* le framework sans l'application dérivée.

Ainsi, chaque framework initial peut être modélisé par un méta-framework qui décrit un framework valide, c'est-à-dire un framework initial et toutes les applications qui peuvent en être dérivées (figure 3.2).

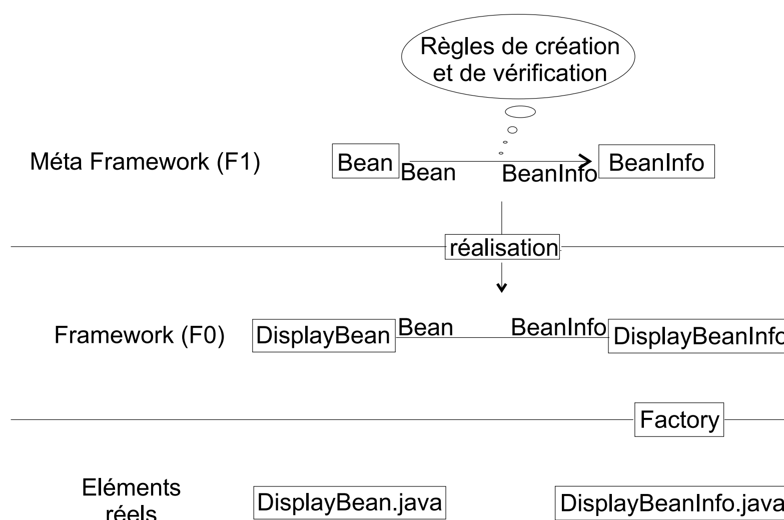


FIG. 3.3 – Modélisation à deux niveaux.

Nous désignons la relation entre le méta-framework et le framework par le terme de réalisation conceptuelle* : F0 est la réalisation de F1. Ainsi, les dépendances s'appliquent au niveau F1 et sont utilisées pour contrôler le niveau F0.

La notion de réalisation utilisée ici n'est pas celle qu'UML utilise pour par exemple connecter une classe et une interface. Notre relation de réalisation décrit simplement le lien entre les niveaux F1 et F0 et l'opération qui permet de passer du premier au second.

Nous avons donc une modélisation d'un framework à deux niveaux plus un niveau d'éléments physiques. Ces niveaux sont représentés sur la figure 3.3. La figure 3.4 montre pour JUnit les différents niveaux tout en séparant le framework initial (partie grisée) de l'application dérivée.

La création du méta-framework fait partie du travail du développeur de framework, cependant nous verrons que cette tâche n'est pas aussi contraignante que la rédaction d'une documentation ou l'écriture de *Hooks*, et peut même faciliter la création du framework. De plus, ni le méta-framework, ni la description du framework ne doivent décrire intégralement le framework. Nous verrons que seul les éléments qui font l'objet de dépendances sont décrits et réifiés.

L'utilisation de cette méta-modélisation peut être faite de deux manières : via le niveau F1 où le développeur identifie le méta-élément qu'il désire réaliser puis reçoit alors l'assistance prévue, ou via le niveau F0 lorsque l'utilisateur crée un élément puis l'identifie comme étant la réalisation d'un élément de F1 et reçoit alors l'assistance prévue.

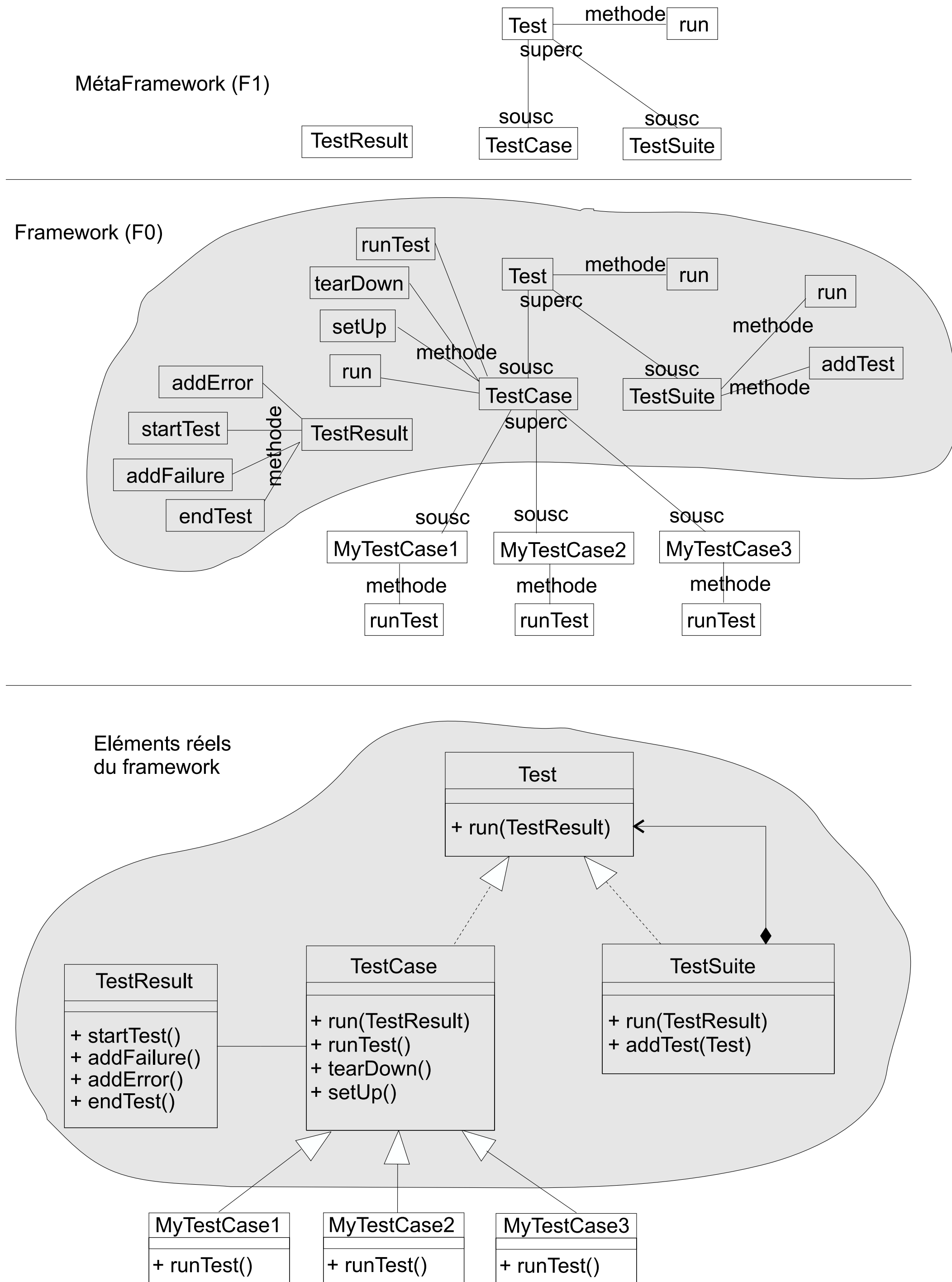


FIG. 3.4 – Représentation des niveaux pour JUnit. La partie grisée désigne le framework initial.

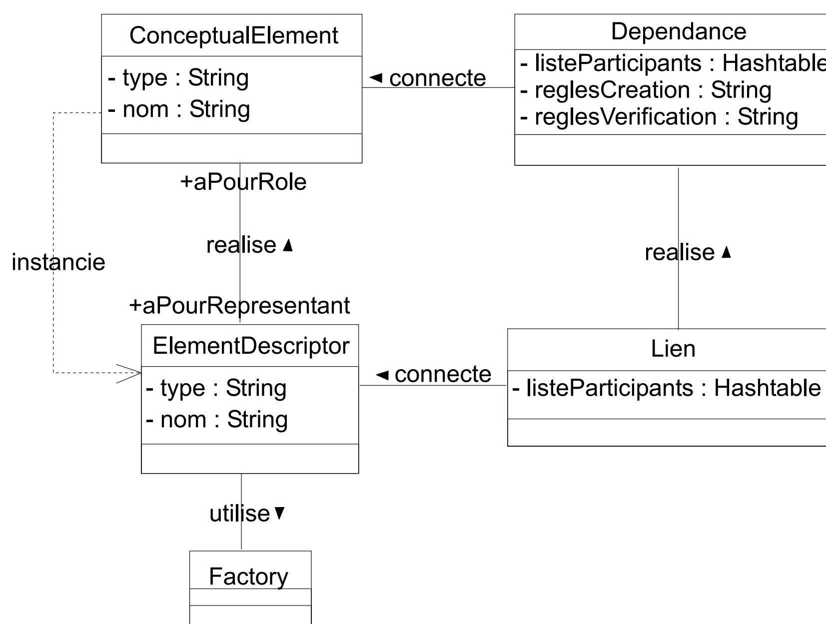


FIG. 3.5 – Diagramme de classes du modèle des méta-frameworks.

3.2 Représentation d'un framework

Cette section présente les entités qui constituent la représentation contextuelle (le méta-framework (F1)) et la description du framework (le framework (F0)). Ainsi, puisque nous nous sommes abstraits de toute contrainte liée à un langage, notre modèle est minimal. Il est constitué des entités suivantes :

- `ConceptualElement` dont les instances représentent les éléments conceptuels et constituent la méta-description du framework (niveau F1) ;
- `Dependance` dont les intances représentent les dépendances qui existent entre les éléments conceptuels du méta-framework (niveau F1) ;
- `ElementDescriptor` dont les instances constituent la description du framework modélisé (ce sont des réalisations conceptuelles des instances de `ConceptualElement`) (niveau F0) ;
- `Lien` dont les instances connectent les instances d'`ElementDescriptor` (niveau F0) ;
- `Factory` qui permet la création des entités réelles de l'application à partir des `elementDescriptors`.

Ces cinq entités et leurs rôles respectifs sont détaillés ci-après. Leur organisation est décrite figure 3.5.

3.2.1 Éléments du modèle

Afin d'alléger la rédaction, nous utilisons la convention suivante[3] pour différencier instances et classes : un nom de classe capitalisé désigne une classe (ex : `ElementDescriptor`), alors qu'un nom de classe commençant par une minuscule désigne une instance (ex : `elementDescriptor`).

ConceptualElement

La classe `ConceptualElement` est le constituant de base de la modélisation du méta-framework. Cet élément est le représentant dans le méta-framework d'un élément du niveau framework ; il en est une réification. Les `conceptualElements` contrôlent les opérations qui sont faites sur le framework et instancient `ElementDescriptor`.

La relation entre un `conceptualElement` et un `elementDescriptor` est une réalisation conceptuelle*. Les instances d'un `ConceptualElement` représentent dans le méta-framework des concepts du framework, par exemple un Bean ou un visiteur qui sont des concepts qui n'ont pas nécessairement un représentant au niveau framework. On dit aussi qu'un `elementDescriptor` joue le rôle conceptuel* d'un `conceptualElement`. Par exemple `DisplayBean` est une réalisation de `Bean` ; `DisplayBean` joue le rôle de `Bean`.

Le méta-framework est donc un ensemble d'instances de `ConceptualElement` connectées par des `dependances`. C'est cette toile d'instances qui décrit, à un niveau méta, la structure du framework et sa validité. Le partie supérieure droite de la figure 3.6 présente le méta-framework pour les JavaBeans.

Chaque `conceptualElement` conserve un lien vers les éléments du framework qu'il représente (lien `realise` rôle `aPourRepresentant` figure 3.6), et inversement. Cette relation a la cardinalité 1-n : un `conceptualElement` peut être le rôle de plusieurs `elementDescriptor`, tel serait le cas pour le `conceptualElement Bean` si plusieurs composants `Bean` étaient créés. Il est aussi représenté dans la partie droite de la figure 3.6 par un lien `réalise`.

Il est à noter qu'un méta-framework ne contient que la réification des éléments qui prennent part à une dépendance. Ainsi quand bien même une classe est réifiée toutes ses méthodes ne le sont pas nécessairement et donc l'intégralité du framework n'est pas représentée au niveau méta.

Afin que le modèle soit indépendant de tout langage, le type réel de l'élément du framework qui sera créé est indiqué par une chaîne de caractères (champs type figure 3.5, `type = "classe"` figure 3.6). Chaque élément est aussi muni d'un nom.

ElementDescriptor

La classe `ElementDescriptor` est utilisée pour représenter les constituants du framework (niveau F0) : classes, méthodes, fichiers, etc. La création de ces éléments est contrôlée par le méta-framework pendant l'opération de réalisation. C'est la validité de leur assemblage qui est contrôlée lors de vérification. Sur la figure 3.6, les instances

[3]Cette convention est celle utilisée dans la spécification d'UML [OMG01b].

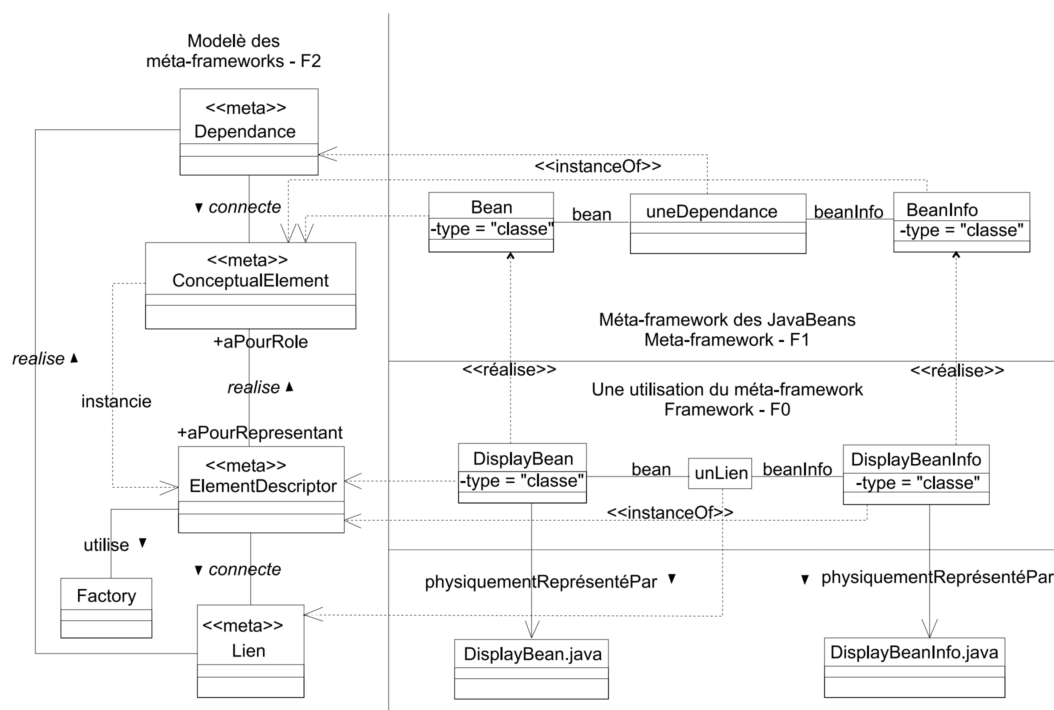


FIG. 3.6 – Modèle de dépendances et exemple des JavaBeans.

d'`ElementDescriptor` sont `DisplayBean` et `DisplayBeanInfo`. `DisplayBean` réalise `Bean`, ou encore il joue le rôle de `Bean` dans le framework. On parlera encore de rôle conceptuel*.

La classe `ElementDescriptor` ne représente pas directement les constituants du framework mais, comme son nom l'indique, une description de l'objet réel qui peut être un fichier, une classe ou une autre ressource.

Le lien entre la représentation en mémoire de l'élément du framework et l'élément réel (on parlera encore d'élément physique) est montré figure 3.6 par la relation `physiquementReprésentéPar`. Le type de l'élément physique et son nom sont respectivement déterminés par le type et le nom portés par l'`elementDescriptor`. Le type de l'`elementDescriptor` est le même que celui du méta-élément qu'il réalise.

Eléments physiques

La séparation entre les descripteurs et les éléments physiques est nécessaire pour garder une indépendance complète vis-à-vis d'un langage et afin de représenter des relations qui ne se manifestent pas parmi les éléments physiques. Par exemple, le lien entre `DisplayBean` et `DisplayBeanInfo` n'a pas de représentant au niveau des éléments physiques (figure 3.6). Si nous utilisions directement les éléments physiques à la place des `elementDescriptors`, ceux-ci devraient implémenter une interface spéci-

fique afin que les conceptualElements gardent leur indépendance. Cela impliquerait une duplication du code de gestion des dépendances et lierait les éléments physiques avec notre modèle, ce qui n'est pas souhaitable d'un point de vue conceptuel. Ainsi, les éléments réels ne font pas partie du modèle. Un elementDescriptor conserve, pour un aspect purement pratique, une référence vers l'élément qu'il représente réellement. La création et l'accès aux éléments physiques sont la responsabilité de la Factory.

Dependance

La classe Dependance est utilisée pour représenter les dépendances structurelles dans le méta-framework. Ses instances interviennent lors de la réalisation et lors de la vérification. Elles contiennent des informations utiles aux deux opérations : des règles de création pour la réalisation et des règles de vérification pour la vérification.

Une dépendance met généralement plusieurs conceptualElements à contribution et fixe le rôle de chacun d'eux. Ce rôle, qui ne doit pas être confondu avec le rôle conceptuel qui relie un conceptualElement et un elementDescriptor, a pour but l'identification de chacun des constituants de la dépendance. Il est représenté par une chaîne de caractères. Dans la représentation graphique que nous utilisons (par exemple figure 3.7) le nom du rôle est porté par la relation connectant les éléments[4]. Ce nom est utilisé pour naviguer entre les éléments dans les règles de création ou dans les règles de vérification[5]. La règle de création décrit le comportement qui sera exécuté lorsque la réalisation sera demandée. Le langage utilisé, qui est présenté section 3.4.1, est ISL [Ber01]. Dans cet exemple, il indique que l'activation de la dépendance a pour cause la réalisation de Bean (Bean.rm ->). La conséquence de cette activation est donnée à droite de la flèche, et indique que BeanInfo sera réalisé et que les résultats des réalisations (des descripteurs) seront liés.

Les règles de vérification sont exprimées en OCL [OMG97]. Dans cet exemple la règle présentée vérifie qu'à chaque descripteur de Bean est associé un BeanInfo.

Une dépendance est utilisée de manière différente selon qu'il s'agit de la vérification ou de la réalisation. Des détails concernant ce point sont donnés dans la sous-section suivante (sous-section 3.2.2).

Les dépendances impliquent d'appréhender de manière inhabituelle certaines caractéristiques du modèle de classes. Ainsi, dans notre modèle la relation entre une méthode et une classe est considérée comme une dépendance. Nous avons donc deux grandes catégories de dépendances : des dépendances « existentielles »* et des dépendances ontologiques* qui représentent des dépendances liées à la nature du modèle ; cependant cette distinction n'a aucune conséquence sur le modèle. Dans l'exemple des JavaBeans de la figure 3.7, seule la première catégorie de dépendances est utilisée. Elle matérialise la relation qui existe entre un Bean et sa description (BeanInfo).

Tout comme les conceptualElements, les dependances ont une représentation au niveau framework. Il s'agit d'instances de la classe Lien. Elles connectent les elementDescriptors en utilisant les rôles de la dépendance qu'elles réalisent (unLien, figure 3.6).

[4]Dans la suite du document, les dépendances qui sont ici représentées par une boîte, seront simplement représentées par un nom sur un trait.

[5]Pour faciliter la lecture, les règles sont représentées dans une note UML attachée à la dépendance à laquelle elles correspondent

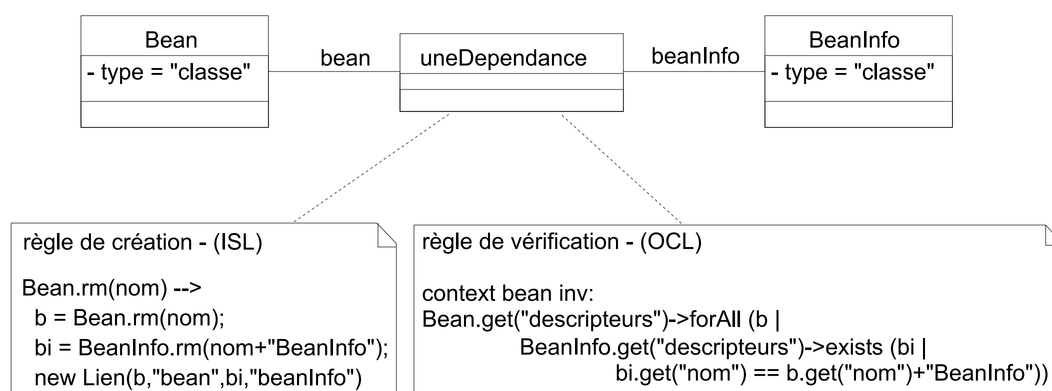


FIG. 3.7 – Méta-modélisation détaillée de `Bean` et `BeanInfo`.

Lien

Les instances de `Lien` connectent les instances d'`elementDescriptor` entre elles. Ces instances sont des réalisations des `Dependances`. Ainsi pour connecter les `elementDescriptors`, elles utilisent les mêmes noms de rôle que ceux utilisés par les dépendances pour connecter les `conceptualElement` (figure 3.6). Pour les liens, on retrouve la distinction faite entre les dépendances ontologiques et existentielles. On a donc des liens ontologiques* et des liens existentiels*.

Factory

La classe `Factory` permet d'abstraire la représentation physique des éléments. Elle est spécialisée en fonction du langage cible et regroupe en une seule place toutes les manipulations d'éléments physiques. C'est en fait l'élément clé de l'indépendance par rapport à un langage.

La `Factory`, bien qu'étant très simple, répercute aux éléments physiques (fichiers, classes...) les modifications que subissent les instances d'`ElementDescriptor`. Cependant, son fonctionnement dépendant du langage cible, nous ne détaillerons pas cet élément.

Il est juste important de noter que la `Factory` utilise le type porté par le descripteur (`elementDescriptor`) pour créer l'élément correspondant et utilise le nom de rôle des `liens` pour associer les élements physiques du framework entre eux. Par exemple, si l'on a une Factory Java, la présence du type `methode` dans un `elementDescriptor` entraînera l'ajout d'une méthode dans une classe, classe qui sera représentée par un fichier .java. On notera que seuls les liens ontologiques auront une représentation parmi les éléments physiques.

Pour résumer, la `Factory` connaît donc la signification de tous les types et des rôles qui ont une répercussion immédiate sur les éléments physiques.

Puisqu'à terme notre modèle est fait pour être intégré dans un environnement de développement dans lequel l'utilisateur manipule principalement des éléments phy-

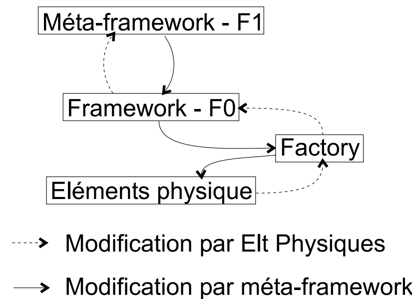


FIG. 3.8 – Fonctionnement de la `Factory`.

siques, la Factory permettra de notifier la description du framework et le méta-framework d'une modification au niveau des éléments physiques (figure 3.8). Cependant ce point n'a pas été davantage exploré à ce jour.

Pour résumer nous avons trois niveaux de modélisation ainsi qu'un niveau de représentation :

1. le premier est celui du framework physique composé de fichiers par exemple .java, .xml, ou de représentation en mémoire des éléments dans le cadre d'un environnement comme Eclipse (niveau de représentation) ;
2. le second est celui de description du framework, la représentation en mémoire de ces ressources à des fins de manipulations (F0, un framework : instances d'`ElementDescriptor` et de `Lien`). Il est indépendant du langage ;
3. le troisième est celui de modélisation des méta-éléments et des dépendances, aussi appelé méta-framework (F1, un méta-framework : instances de `ConceptualElement` et de `Dependance`) ;
4. le quatrième qui est le niveau du modèle de la modélisation des dépendances (F2, le modèle des méta-frameworks), dont cette section constitue la description.

3.2.2 Réalisation, navigation et vérification

Après avoir présenté les éléments de notre modèle, nous détaillons les différentes fonctionnalités de celui-ci et le rôle de chacun des éléments. La figure 3.9 montre le diagramme de classes complet de notre modèle (niveau F2).

Réalisation conceptuelle

Comme nous l'avons vu, la réalisation permet d'assurer la cohérence d'une application développée à partir d'un framework en utilisant le méta-framework associé.

La réalisation met principalement à contribution les instances de `ConceptualElement` puisque ce sont elles qui contrôlent les dépendances. Elle est représentée par l'opération `rm` (Réalisation de Modèle). Cette opération prend comme paramètre le nom de l'`elementDescriptor` qui devra être créé.

FIG. 3.9 – Modèle détaillé des méta-frameworks (F2).

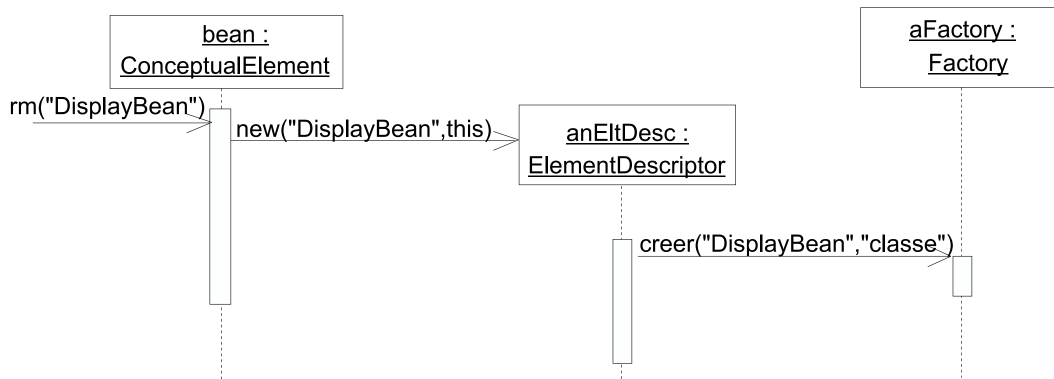


FIG. 3.10 – Réalisation d'un `conceptualElement` sans dépendance.

Réalisation sans dépendance. Le comportement de cette méthode est simplement de créer une instance d'`ElementDescriptor` ayant pour rôle le `conceptualElement` qu'elle réalise. Ce processus est résumé dans le diagramme de séquences de la figure 3.10 pour l'exemple des JavaBeans : lorsque le message `rm` est reçu par `Bean` avec comme nom `"DisplayBean"`, l'instance `DisplayBean` de type `"classe"` est créée, et l'élément physique (le fichier DisplayBean.java[6]) en utilisant l'information de type portée par l'`elementDescriptor` est créé par la `Factory`.

Réalisation avec dépendances. Lorsqu'un `conceptualElement` est réalisé, les dépendances qui lui sont associées sont à leur tour réalisées (méthode `realiser`). La réalisation d'une dépendance est uniquement effectuée si le `conceptualElement` réalisé est identifié comme sa cause. Dans ce cas, la réalisation normale est alors court-circuitée et la conséquence de la dépendance exécutée. Ainsi, pour l'exemple des JavaBeans repris dans la note de la figure 3.11, la réalisation de Bean (`Bean.rm(nom) ->`) a pour conséquence la réalisation effective de Bean (`Bean.rm(nom) ;` [7]). Ensuite, la réalisation de BeanInfo (`BeanInfo.rm(...)`) est effectuée, et enfin les deux descripteurs créés par la réalisation sont liés (`new Lien(...)`). Il est à noter que l'ordre des opérations est fixé par la règle ISL et que la création du lien entre les descripteurs n'est jamais automatique. Ce processus de réalisation avec dépendance est explicité sous forme d'un diagramme de séquences figure 3.11.

Si le `conceptualElement` est cause de plusieurs dépendances, c'est le résultat de la fusion du comportement de chacune des dépendances qui est exécuté. La sémantique de la fusion utilisée est celle décrite dans [Ber01, DBFM02]. Elle sera reprise sous-section 3.4.1. En attendant, il suffit de retenir que la fusion s'effectue sur les causes des dépendances et que le résultat ne contient qu'un seul appel à la réalisation de la cause, quand bien même elle apparaît plusieurs fois dans les dépendances.

[6]En supposant que la Factory utilisée est celle qui projette vers du Java.
[7]Puisque l'exécution initiale a été court-circuitée, il est nécessaire d'appeler le message initial à la manière d'un *wrapper* [BFJR98].

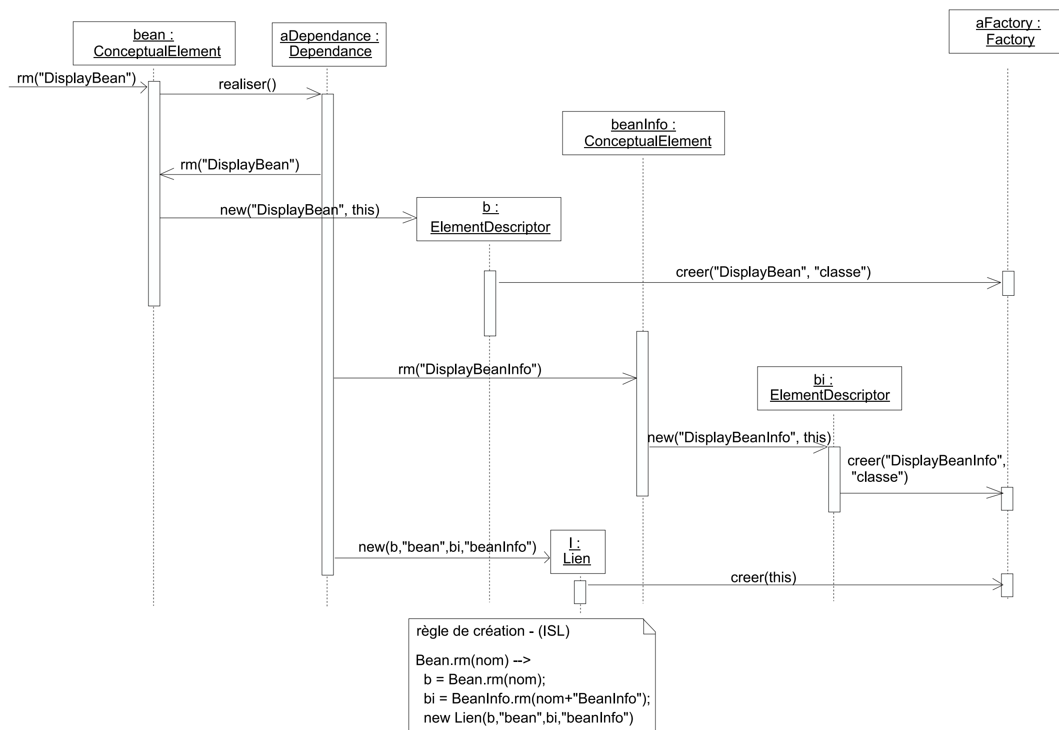


FIG. 3.11 – Réalisation d'un élément conceptuel et d'une dépendance associée.

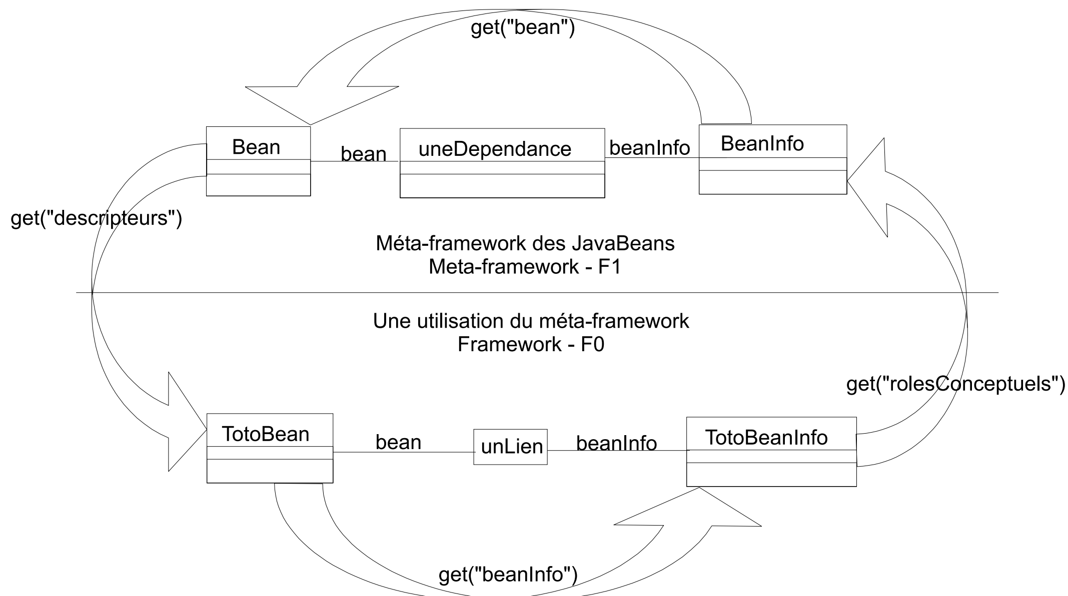


FIG. 3.12 – La navigation dans un modèle, quelques opérations.

Navigation

Puisque le méta-framework et le framework référencent les éléments qu'ils ont engendrés et qui les a engendrés, il est possible de naviguer entre les niveaux de représentation d'un framework. Cette navigation se fait à l'aide des méthodes `get` de `ConceptualElement` et d'`ElementDescriptor`. Ainsi, sur une instance de `ConceptualElement`, un appel à `get` avec comme paramètre `"descripteurs"` donne accès à toutes les instances créées (figure 3.12), alors qu'un appel `get("roleConceptuels")` sur une instance d'`ElementDescriptor` retourne tous les rôles que joue l'élément (c'est-à-dire une instance de `ConceptualElement`) . Cela permet par exemple à `DisplayBean` de savoir que `BeanInfo` est son rôle conceptuel, et à `BeanInfo` de connaître tous les `Beans` dont il est le rôle conceptuel (figure 3.12).

Cette navigabilité, qui est possible entre les niveaux, est possible au sein même d'un niveau en utilisant les mêmes méthodes. Pour le niveau F1, la navigation est effectuée en utilisant les rôles des dépendances. Pour le niveau F0, elle utilise les rôles des liens qui sont créés par la réalisation des dépendances (sous-section 3.2.2). Cela permet par exemple à `DisplayBean` de connaître qui est son `BeanInfo` associé en utilisant `get("beanInfo")` et à `BeanInfo` de connaître son `Bean` (`get("bean")`) (figure 3.12).

Les informations sur le type et le nom des `conceptualElements` et des `elementDescriptors` sont accessibles par `get("type")`, et `get("nom")`.

La valeur retournée d'une navigation pouvant être un ensemble (de `conceptualElement` ou d'`elementDescriptor` selon les cas), un ensemble de primitives de manipulation est fourni : `size()`, `union()`, `choose()`. `Choose` permet à l'utilisateur de choisir un élément dans un ensemble.

```
bean.get("descripteurs")->forAll (b |
   beanInfo.get("descripteurs")->exists (bi |
     bi.get("nom") == b.get("nom")+"BeanInfo"))
```

FIG. 3.13 – Exemple d'une règle de vérification.

Vérification

L'opération de vérification consiste en la validation de la structure du framework non plus pendant son extension, mais une fois l'extension terminée.

Ainsi, les règles de vérification expriment des assertions sur le niveau framework à partir du méta-framework. Elles fournissent entre autre un moyen de vérifier ce qui a été mis en place par la réalisation. Elles sont représentées par une contrainte OCL [OMG97] et sont évaluées sur demande. Elles n'ont aucun effet de bord.

Grâce à la navigabilité inter-niveaux, nous pouvons avoir à un niveau méta-framework, des contraintes s'appliquant au niveau framework. Ainsi sur la représentation des JavaBeans on peut avoir une contrainte vérifiant qu'à toute réalisation de `Bean` correspond un `BeanInfo` au niveau méta (cf figure 3.13).

3.2.3 Création d'un méta-framework

La création du méta-framework est une tâche qui incombe au développeur du framework. Afin de créer le méta-modèle du framework (réification du framework), l'utilisateur va être amené à créer des instances de `ConceptualElement` et de `Dependance`.

La création d'un `conceptualElement` requerra du développeur un type, ainsi qu'un nom. La création d'une `Dependance` est plus lourde puisque les rôles de `conceptualElements`, les `conceptualElements`, ainsi que les règles de création et de vérification doivent être fournis. Sur le même principe, des règles de suppression peuvent être créées. Nous ne les détaillerons pas car elles relèvent plus d'un aspect lié à la création d'outil qu'à une fonctionnalité de notre modèle.

La difficulté pour ce développeur est de décider des éléments qu'il veut réifier. Son choix doit inclure les éléments qui sont amenés à varier, et principalement ceux de ces éléments qui ont des contraintes avec d'autres. Ainsi, nous préconisons qu'au moins tous les points de variation soient méta-modélisés puisqu'ils sont les plus à même d'être utilisés.

Comme nous l'indiquions précédemment, la réification peut n'être que partielle. Ainsi lorsqu'une classe est réifiée toutes ses méthodes ne doivent pas nécessairement l'être. Afin de connaître les éléments à réifier, il faut à partir d'un élément jouant un rôle dans une dépendance réifier tous les éléments qui le contiennent jusqu'au niveau du framework (par exemple si l'on veut réifier un paramètre d'une méthode, il faut réifier la méthode qui le contient, et donc la classe qui contient la méthode). Les dépendances qui lieront le résultat de cette réification sont des dépendances ontologiques.

Lors de cette création, le développeur peut décider de cacher certains éléments conceptuels afin d'en empêcher que l'utilisateur final ne les réalise. Cela permet de contrôler les endroits sur lesquels des modifications peuvent être effectuées.

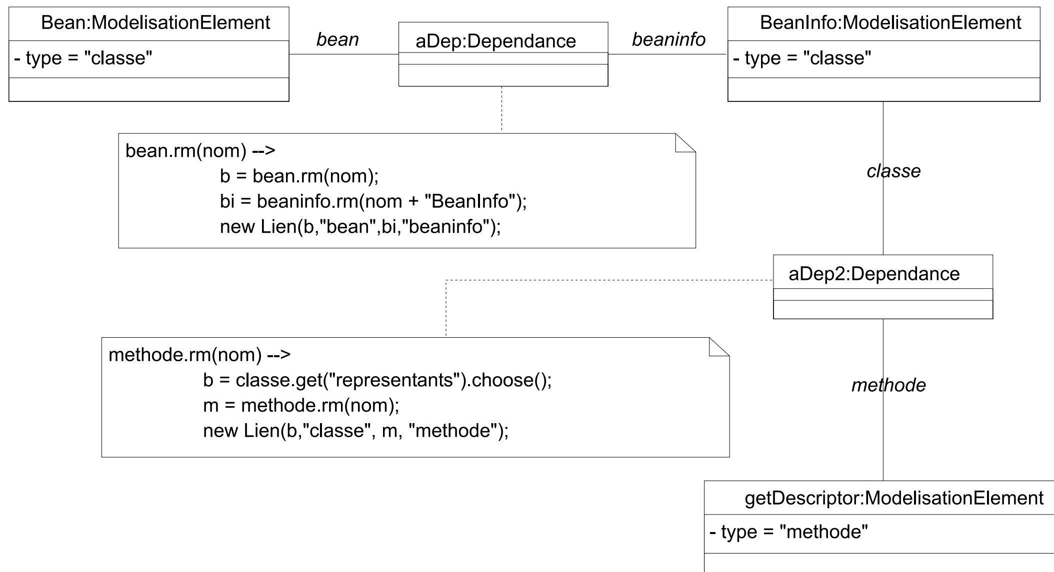


FIG. 3.14 – Modèle des JavaBeans avec la méthode `getDescriptor`.

3.3 Modèle Bean complet

Les fonctionnalités de notre modèle détaillées, nous complétons l'exemple des JavaBeans afin d'exemplifier plus largement l'utilisation du modèle.

La spécification des JavaBeans indique que la classe `BeanInfo` doit contenir une méthode appelée `getDescriptor`. La figure 3.14 montre l'élément conceptuel représentant cette méthode, ainsi qu'une instance de `Dependance` reliant la méthode et sa classe d'appartenance. Cette dépendance ontologique représente une information structurelle dont l'unique but est de connecter une instance de la méthode `getDescriptor` à une instance de `BeanInfo`. Elle n'implique aucune création d'élément. Si l'on voulait forcer la création de cette méthode, il faudrait alors ajouter une dépendance entre `BeanInfo` et `getDescriptor` dont le déclenchement serait la réalisation de `BeanInfo` (dépendance similaire à celle entre `Bean` et `BeanInfo`).

On notera que dans l'exemple, la méthode `getDescriptor` est réalisée sans contexte (sans classe connue), et que lors de la réalisation de la règle il est demandé à l'utilisateur de choisir la classe à laquelle il souhaite attacher cette méthode (appel à la méthode `choose`). Nous pensons que dans la cadre d'une implémentation dans un environnement de programmation, le contexte peut être déterminé et le `choose` rempli automatiquement sans interroger l'utilisateur.

Dans l'état actuel du modèle des Beans, l'utilisateur peut créer autant de méthodes `getDescriptor` qu'il veut. Afin de le restreindre à une unique méthode `getDescriptor` par `BeanInfo`, une dépendance contrôlant la cardinalité est posée entre `getDescriptor` et `BeanInfo`. Cette règle est présentée figure 3.15.

```
getDescriptor.rm(nom) ->
   cl = BeanInfo.get("descripteurs").choose();
   taille = cl.get("methode").size();
   if ( taille < 1 )
    getDescriptor.rm(nom);
   else
    throws ERROR;
```

FIG. 3.15 – Règle vérifiant la cardinalité de la méthode `getDescriptor`.

3.4 Réutilisation des dépendances

Au vu des quelques dépendances déjà décrites, il semble évident que les dépendances qui connectent une méthode et sa classe ou qui connectent Bean et BeanInfo sont susceptibles d'être réutilisées. Afin que le développeur ne soit pas obligé de réécrire ces mêmes dépendances encore et encore, il est souhaitable qu'elles soient encapsulées en tant qu'entités réutilisables, afin de constituer une bibliothèque de dépendances. À partir de cette bibliothèque extensible par le développeur, l'expression des dépendances et la documentation du framework deviennent moins fastidieuses.

Nous détaillons ci-après les langages d'expression des dépendances et les concepts qui permettent la réutilisation des dépendances.

3.4.1 ISL, le langage d'expression des règles de création

Comme nous l'avons déjà indiqué, le langage utilisé pour l'expression des dépendances est le langage ISL (Interaction Specification Language) [Ber01]. « Ce langage permet d'exprimer des interactions entre des entités communicantes et cela sans avoir à modifier le code des objets. En effet, ISL permet la connexion d'instances en modifiant le comportement des objets à la réception de messages »[DBFM02]. Une présentation plus détaillée des concepts qui régissent ce langage sont présentés annexe A.

Concepts d'ISL

Le langage ISL permet d'exprimer deux concepts : le schéma d'interactions* et l'interaction*. Un schéma d'interactions décrit des connexions possibles (en terme d'envoi de message) entre des classes. Une interaction instancie un schéma pour représenter les connexions réelles entre des objets. Le second ne peut exister sans le premier et l'instanciation est explicite.

ISL offre une syntaxe assez simple pour décrire un schéma d'interactions sous forme de règles. Une règle ISL décrit comment le comportement d'un objet (c'est-à-dire la réception d'un message par cet objet) doit être modifié quand il interagit avec un autre. Une règle modifie donc le comportement attendu d'une méthode ; elle peut aussi être considérée comme une règle de réécriture. Elle est composée d'une partie gauche qui décrit une réception de message et d'une partie droite qui décrit le nouveau compor-

```
interaction BeanInteraction(ConceptualElement b,
                            ConceptualElement bi) {
  b.rm(nom) ->
    x = b.rm(nom);
    y = bi.rm(nom+"BeanInfo");
    new Lien(x, "bean", y, "beanInfo");
}
```

FIG. 3.16 – Exemple de schéma d'interactions pour la dépendance Bean / BeanInfo.

```
Bean.rm(nom) ->
   x = Bean.rm(nom);
   y = BeanInfo.rm(nom+"info");
   new Lien(x, "bean", y, "beaninfo");
```

FIG. 3.17 – Instanciation du schéma d'interactions BeanInteraction (figure 3.16).

tement à exécuter en utilisant le langage ISL. Le symbole séparant partie gauche et droite est l'opérateur « `->` ».

Bien qu'ISL ait comme base un modèle classe / message il s'adapte très simplement au couple `ConceptualElement` / `Realisation` et constitue un moyen simple d'exprimer le comportement des dépendances puisque les causes sont représentées par des parties gauches d'interaction et la conséquence par des parties droites.

Utilisation d'ISL pour les dépendances

Jusqu'alors, les règles de création de nos dépendances étaient écrites avec des interactions (l'équivalent de schémas d'interactions déjà instanciés) (figure 3.14). Cependant, puisque certaines règles sont récurrentes, nous les décrirons à l'aide de schémas d'interactions. La figure 3.16 montre le schéma d'interactions qui aurait pu servir à la description de la règle de création entre `Bean` et `BeanInfo`. Ce schéma d'interactions fait donc interagir des `conceptualElements` en redéfinissant la méthode `rm`. Il indique que lorsque l'appel de méthode `rm` est reçu par l'objet qui sera joué par `b` il faudra alors réellement exécuter `rm` sur `b`, puis réaliser `bi` et enfin créer un lien (`new Lien()`).

Lorsque le schéma est instancié (avec des instances de `ConceptualElement`) les règles sont partiellement instanciées, c'est-à-dire que les variables dénotant les éléments participant à l'interaction sont connues. Le résultat de cette instanciation est décrit figure 3.17 et l'on reconnaît la règle présentée dans les précédentes figures. Ce schéma a été instancié en passant en paramètre les éléments conceptuels `Bean` et `BeanInfo`. Cette phase d'instanciation n'a rien à voir avec la réalisation. Elle est effectuée par le développeur du framework lorsqu'il pose une dépendance entre des éléments conceptuels.

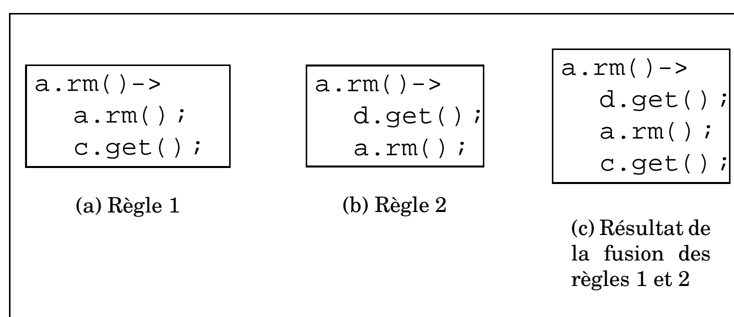


FIG. 3.18 – Exemple de la fusion de deux interactions.

Propriétés complémentaires

Bien que le langage soit muni d'autres opérateurs, nous n'utilisons que la conditionnelle, la séquence, la capture et l'émission d'exception. Afin de faciliter l'écriture des schémas, des variables et des fonctions peuvent être définies localement au schéma.

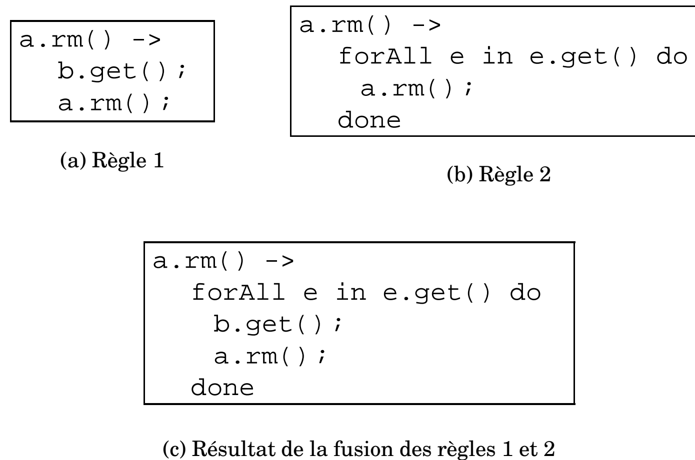
Lorsque plusieurs interactions ont pour source le même objet avec le même message déclencheur, elles sont fusionnées de manière à ce que le message déclencheur n'apparaisse qu'une fois dans la partie droite du résultat comme le montre la figure 3.18. Il est à noter que si le message n'apparaît pas dans la partie droite son émission n'a pas lieu. La fusion se formalise par un ensemble fini de règles de fusion définies en fonction des opérateurs du langage ISL et sur un ensemble fini de règles d'équivalence. La fusion respecte la cohérence de l'ensemble des règles. Elle est commutative (quel que soit l'ordre d'instanciation des schémas, le comportement résultant est sémantiquement le même) et elle est transitive. Toutes ses propriétés ont été montrées dans [Ber01]. On notera que la fusion s'opère au moment de l'instanciation des schémas, et que parfois la fusion peut être rejetée indiquant que les règles ne peuvent pas être combinées et qu'il faut alors en changer l'expression.

Extension d'ISL

Afin d'exprimer toutes nos dépendances nous avons du étendre ISL en y ajoutant la boucle. Cette structure de contrôle ne s'était jamais avérée nécessaire aux auteurs d'ISL. En effet, ils encapsulaient la boucle dans une méthode qui était embarquée dans un fonction de l'interaction. Cependant cette solution n'est pas satisfaisante car les fonctions embarquées sont ignorées lors de la fusion, et ne peuvent pas contenir d'appel au message déclencheur de l'interaction.

Afin de lever cette dernière contrainte nous avons donc introduit la construction `forAll ... in ... do ... done` qui permet d'itérer sur une collection et qui autorise dans le corps de la boucle l'appel au message déclencheur. La sémantique intuitive de la fusion de cette construction est que l'appel au message déclencheur est remplacé par le résultat de la fusion des autres messages (figure 3.19).

Cette sémantique intuitive a été mise en place dans l'implémentation Noah [Pro02],



(a) Règle 1

(b) Règle 2

(c) Résultat de la fusion des règles 1 et 2

FIG. 3.19 – Exemple de la fusion d'une interaction avec une boucle.

```
verification VerifBeanInfo(ConceptualElement M1,
                           ConceptualElement M2) {
  context M1 inv :
  M1.get("descripteurs")->forAll (b |
   M2.get("descripteurs")->exists (bi |
     bi.get("nom") == b.get("nom")+"BeanInfo"))
}
```

FIG. 3.20 – Exemple d'un schéma de vérifications pour Bean / BeanInfo.

cependant la preuve formelle de cette sémantique n'a pas été faite et fera l'objet d'un travail ultérieur.

3.4.2 Réutilisation des règles de vérification

Nous venons de présenter ISL, le langage de description des règles de création. Cependant nos dépendances sont aussi constituées de règles de vérification qui requièrent la même flexibilité que les règles de création.

Afin d'avoir un modèle similaire à celui proposé par ISL qui fournit des schémas d'interactions et des interactions, nous avons créé le concept similaire pour les règles de vérification que nous avons appelé schéma de vérifications*. Un schéma de vérifications décrit donc des règles de vérification possibles entre des éléments conceptuels, et une instance de vérification instancie un schéma pour représenter les règles entre des instances d'éléments conceptuels. Le second ne peut exister sans le premier et l'instanciation est explicite. Un schéma de vérifications est composé de contraintes décrites en OCL qui seront ensuite instanciées.

```
context Bean inv :
Bean.get("descripteurs")->forAll (b |
   BeanInfo.get("descripteurs")->exists (bi |
     bi.get("nom") == b.get("nom")+"BeanInfo"))
```

FIG. 3.21 – Instanciation du schéma de vérifications Bean / BeanInfo.

```
DependanceB-BInfo(ConceptualElement B, ConceptualElement Bi) {
  interaction {
     B.rm(nom) ->
       x = B.rm(nom);
       y = Bi.rm(nom+"BeanInfo");
       new Lien(x, "bean", y, "beanInfo");
    }

  verification {
     context B inv :
     B.get("descripteurs")->forAll (aBean |
       Bi.get("descripteurs")->exists (aBeanInfo |
         aBean.get("nom") == aBeanInfo.get("nom")+"BeanInfo"))
  }
}
```

FIG. 3.22 – Dépendance complète entre Bean / BeanInfo.

Jusqu'alors nous n'avons présenté que des contraintes OCL instanciées cependant nous écrirons désormais des schémas de vérification. Pour l'exemple des JavaBeans, un tel schéma est décrit figure 3.20. Lors de son instanciation, les variables dénotant les éléments conceptuels participant à la vérification sont connues, donnant lieu à une contrainte OCL comme celle décrite figure 3.21.

3.4.3 Réutilisation de dépendances

Une dépendance est donc constituée d'un couple schéma d'interactions, schéma de vérifications. Grâce à ces concepts les dépendances sont désormais de réelles entités réutilisables.

Ainsi, un développeur peut donc au choix écrire une nouvelle dépendance : c'est-à-dire écrire un schéma d'interactions et un schéma de vérifications qu'il devra ensuite instancier, ou simplement instancier une dépendance existant déjà. La figure 3.22 montre la dépendance complète qui est utilisée entre Bean et BeanInfo, ainsi que la forme sous laquelle nous exprimons les dépendances dans la suite de ce document. Les paramètres ont été factorisés dans la déclaration de la dépendance.

Afin de pouvoir être utilisées au niveau F1, ces dépendances réutilisables sont conceptuellement considérées comme des sous-classes de `Dependance` dans lesquelles

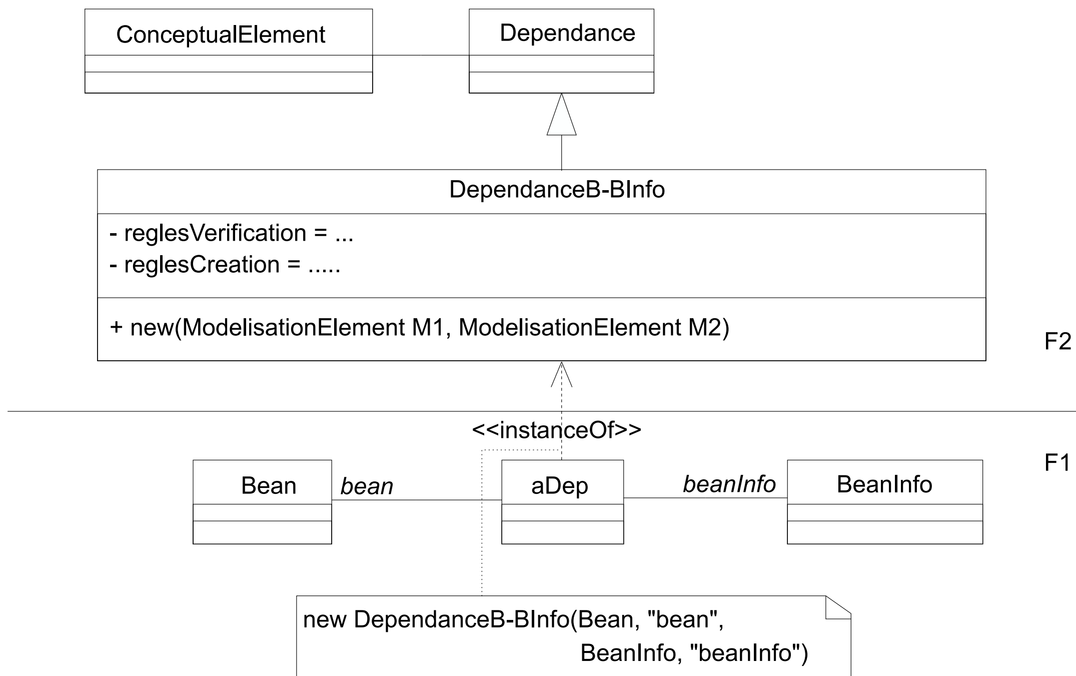


FIG. 3.23 – Réutilisation d'une dépendance.

les attributs `règlesCreation` et `règlesVerification` sont respectivement les schémas d'interactions et les schémas de vérification. La figure 3.23 montre l'organisation de la dépendance dans le cas de Bean / BeanInfo.

3.5 Dépendances structurelles et la hiérarchie de modèles de l'OMG

Devant les efforts de standardisation fait par l'OMG (Object Management Group[8]) pour établir des langages de modélisation interopérables, il nous a semblé essentiel de voir comment notre modèle pour les frameworks pouvait s'intégrer avec les différentes techniques de modélisation proposées pas l'OMG. Ainsi nous présentons les rapprochements possibles de notre modèle dans la modélisation à quatres couches proposée par l'OMG.

Puisque nous manipulons principalement des classes, des méthodes... nous avons décidé de rapprocher notre modèle d'expression des dépendances d'UML / MOF. Pour cette étude nous restreignons donc notre modèle à la représentation d'éléments constituant des diagrammes de classes. Nous ne considérons pas la notation graphique, mais simplement les problèmes liés à la modélisation.

[8]http ://www.omg.org.

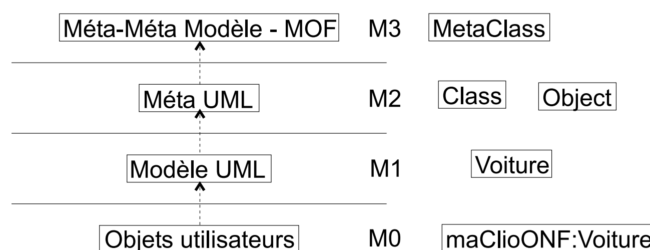


FIG. 3.24 – Modélisation en UML.

A première vue, un tel rapprochement ne devrait pas poser de problème. La modélisation en UML propose quatre niveaux qui sont présentés figure 16. Le niveau M3 (ou MOF) est le plus générique, il fournit une infrastructure pour spécifier des méta-modèles. Le niveau M2 définit le langage UML, et contient donc des éléments comme des classes, des objets, etc. Le niveau M1 représente le modèle des données de l'utilisateur en UML, par exemple une classe Voiture. Le niveau M0 constitue les objets de l'utilisateur. Chaque niveau est instance du niveau précédent.

Rapprochement naïf. Puisque nous avons trois niveaux de modélisation (sans compter le niveau des éléments physiques) et qu'UML en propose trois significatifs, une première solution consiste à faire correspondre nos niveaux par leur numéro (F0 avec M0, F1 avec M1, etc.). Ce rapprochement signifie donc que les éléments de F0 (des classes, des méthodes...) sont les objets utilisateur et donc que F1 correspond à M1, et que F2 est une extension de M2. Cette approche naïve pose cependant un problème puisqu'il ne reste plus de niveau dans UML pour représenter les instances des classes du framework.

F0 + ElementPhysique = M1. Une solution plus complexe part du rapprochement suivant : puisque le niveau F0 représente les constituants d'un framework (classes, méthodes...), il semble naturel de le mettre en relation avec M1. En effet, M1 et F0 représentent bien le même type d'éléments : F0 et les éléments physiques représentent la même information mais à un niveau de détail différent. Ainsi nous pensons que les éléments physiques appartiennent aussi au niveau M1, et que la relation qui existe entre F0 et les éléments physiques est une relation de « réalisation UML » comme le montre la figure 3.25.

Nous n'avons fait qu'un rapprochement partiel, il nous faut encore rapprocher F1 et F2 d'UML. Nous explorons deux possibilités.

F1 + F0 + ElementPhysique = M1. La première possibilité est de considérer F1 comme un élément du modèle utilisateur et donc de le considérer comme faisant lui aussi partie de M1. Ce type de modélisation est en phase avec ce qu'autorise UML avec le

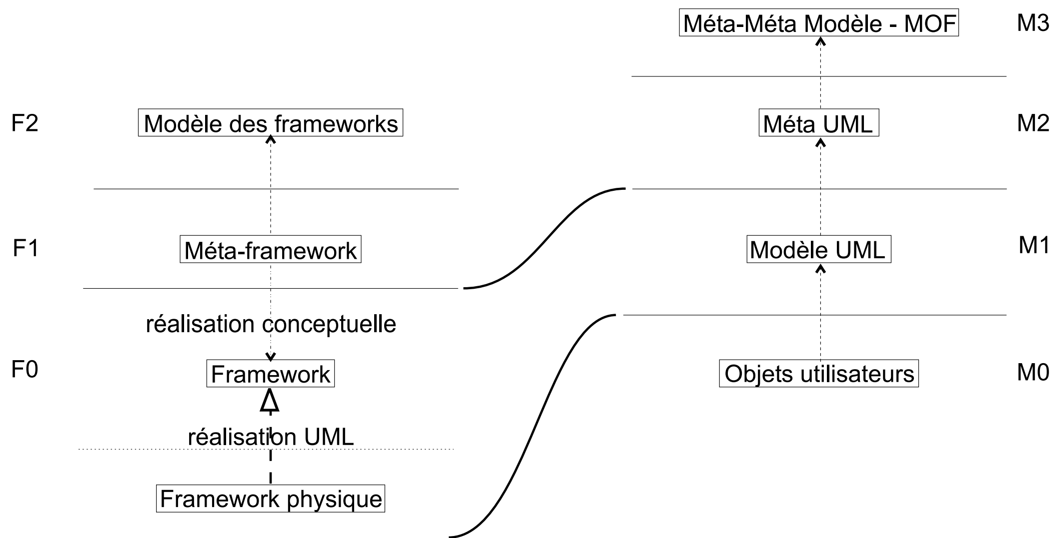


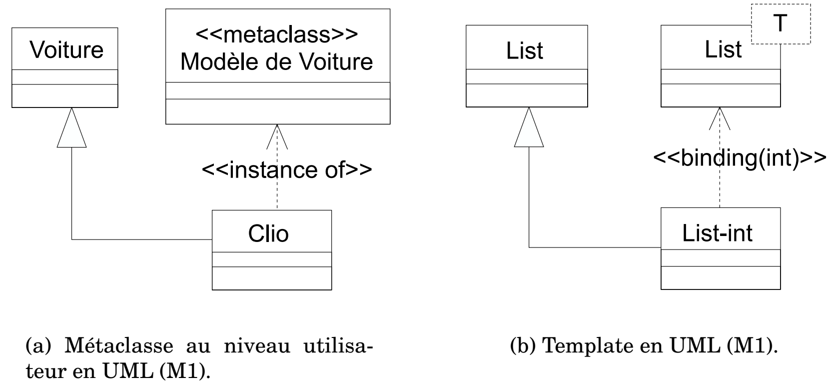
FIG. 3.25 – Rapprochement de F0 et ElementPhysique de M1.

stéréotype « `metaclass` » au niveau du modèle de l'utilisateur 3.26(a), ou encore le mécanisme des *templates* 3.26(b). En effet, le stéréotype « `metaclass` » permet de préciser la méta-classe d'une classe afin d'en augmenter le comportement ou de le raffiner. De même, le mécanisme de *template* est utilisé pour engendrer des éléments de modélisation par une opération spécifique dite de *binding*.

Cette modélisation rentre bien dans le moule UML en intégrant F2 comme extension de M2. Cette extension propose une spécialisation de la classe `Dependance` d'UML pour représenter le concept de dépendance et la définition de la réalisation conceptuelle comme une nouvelle sorte d'`Abstraction`. Cette solution est d'autant plus adaptée que nous conserverions alors les relations entre niveaux du modèle des frameworks : on instancie pour passer de F2 à F1 (comme pour passer de M2 à M1), et l'on réalise conceptuellement pour passer de F1 à F0 (ce qui est une relation tout aussi normale que le « `binding` » UML pour instancier un *template* ou la relation « `instance of` » entre une classe et une méta-classe.

Dans cette intégration F2 peut être vu comme un profil (*profile*) d'UML pour la représentation des dépendances d'un framework. En effet, tout comme un profil, F2 définit de nouveaux concepts au niveau M2 tels que la réalisation conceptuelle, la dépendance, le lien, etc. Dans une forme aboutie un tel profil pourrait proposer un ensemble de stéréotypes pour marquer dans un modèle (au niveau M1) les éléments appartenant à la représentation d'un framework.

F0 + ElementPhysique = M1 et F1 + F2 = M2. Cette solution considère F2 comme une spécialisation de M2, et F1 comme une spécialisation de cette spécialisation (figure 3.28). F2 étant un profil, F1 serait alors un sous-profil particulier à chaque framework représentant le méta-framework. Ainsi on aurait donc un sous-profil pour JUnit, un sous-profil pour JavaBeans, etc. Bien que cette solution soit viable, elle semble aller



(a) Métaclasse au niveau utilisateur en UML (M1).

(b) Template en UML (M1).

FIG. 3.26 – Métaclasse et *template* d’UML.

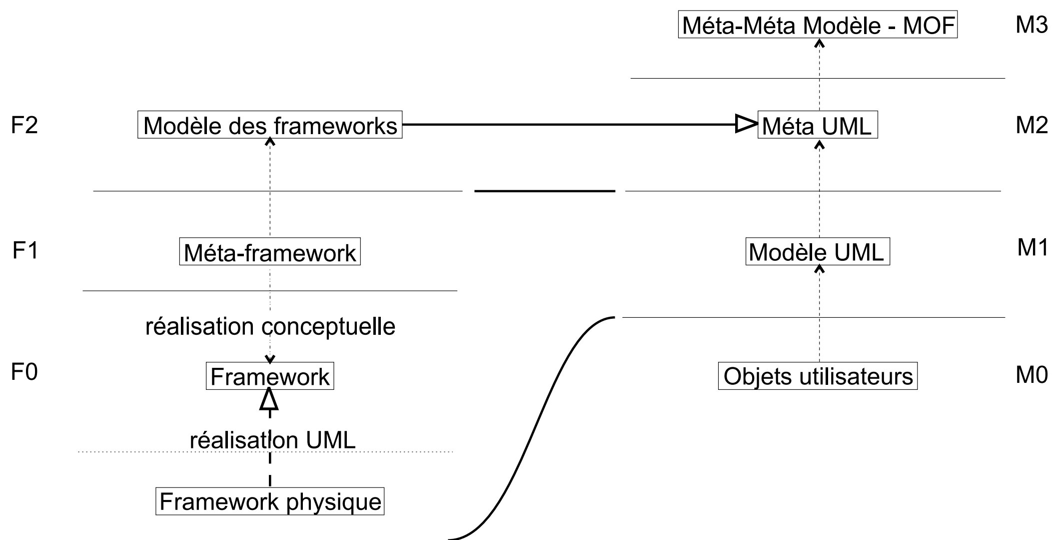


FIG. 3.27 – F1, F0 et ElementPhysique dans M1.

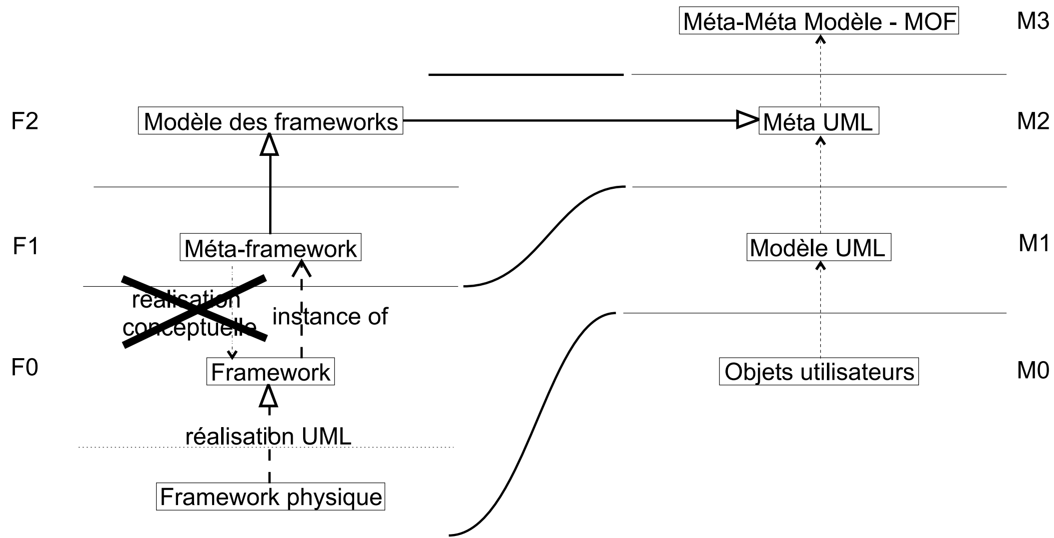


FIG. 3.28 – F1 et F2 dans M2.

à l'encontre du concept de profil qui se veut être une spécialisation d'UML pour la modélisation de familles d'applications (il existe par exemple un profil temps-réel).

De plus, puisque cette solution sépare F1 et F0 dans deux niveaux UML différents, la relation qui les lie ne peut plus être une réalisation conceptuelle mais doit être une instanciation puisque c'est la seule opération qui permette de passer d'un niveau à un autre en UML.

F2 = M3 (MOF). Étant donné la simplicité des constituants de notre modèle et aussi la non nécessité de concepts telles que classes, méthodes... on retrouve dans notre modèle le dépouillement du MOF. Ainsi, on peut envisager F2 comme spécialisation de M3. Cependant cette solution n'est pas celle qui est préconisée par l'OMG. En effet le MOF est le modèle des modèles dont on crée des instances (par exemple Meta-UML ou CWM).

Ainsi quelle que soit la solution retenue l'instanciation ou la spécialisation, nous obtenons un modèle parallèle à celui d'UML (figure 3.29). Si nous voulons connecter ce modèle parallèle à celui d'UML, il faut donc définir une passerelle, c'est-à-dire un élément qui va projeter nos éléments de description dans M1, comme le montre la figure 3.29. Cette notion de projection qui transforme un modèle donné dans un autre modèle existe dans notre modèle des frameworks avec la Factory. Ainsi la projection du niveau F1 vers M1 est équivalent à une projection de F1 vers F0 où F0 représenterait un diagramme UML et où la factory serait une factory UML.

Bien entendu d'autres solutions auraient pu être envisagées cependant elles paraissaient trop fantaisistes pour être discutées ici (tout dans le niveau M1, ou M2, etc.). Au regard de cette analyse, on constate la grande flexibilité qu'autorise la modélisation en UML. Devant les avantages et inconvénients de chacune d'entre elles, et tant que le méta-modèle d'UML ou le MOF ne seront pas le modèle des outils, il ne sera pas nécessaire d'en privilégier une plus qu'une autre.

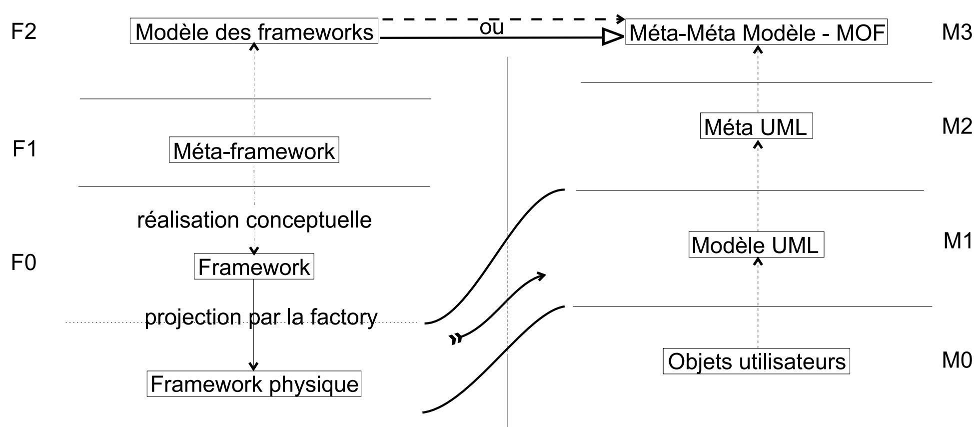


FIG. 3.29 – Rapprochement F3 / MOF.

3.6 Conclusion sur notre modèle d'expression des dépendances structurelles

Ce chapitre a donc présenté notre modèle d'expression des dépendances structurelles et les concepts sous-jacents aux deux opérations supportées : la réalisation et la vérification.

Ce modèle, qui propose une représentation des frameworks selon deux niveaux d'abstraction (le niveau conceptuel et le niveau de description), gagne en flexibilité en découplant le modèle de tout langage de programmation et de toute supposition sur la nature des éléments manipulés. Ainsi un framework et sa méta-représentation sont représentés par un graphe conceptuel où les entités représentent des concepts du framework (classes, méthodes, fichier, appel de méthode...) et où les relations représentent le rôle de chacun de ces éléments les uns par rapport aux autres. On regrettera simplement que la création de la méta-description ne puisse pas être automatisée, et que cette tâche qui peut parfois être lourde incombe au développeur du framework.

Afin d'exprimer simplement le comportement des dépendances nous avons choisi d'utiliser ISL et son concept de schéma d'interactions, puis de transposer ce concept pour OCL afin d'obtenir des dépendances réutilisables. Ces dépendances réutilisables qui sont alors manipulées comme des objets de premier ordre permettent alors de constituer des bibliothèques comme le montre le chapitre suivant (chapitre 4).

Nous pensons que notre approche peut s'inscrire dans les concepts de Generative Programming [CE00] et de Declarative Meta Programming [dmp]. En effet, notre représentation engendre les éléments d'un framework (opération de réalisation) en déclarant à un niveau méta la validité du niveau framework.

Nous avons donc un système d'expression des dépendances qui aide l'utilisateur du framework à créer des extensions valides d'un framework. Le chapitre suivant montre l'utilisation de ces dépendances pour documenter le framework. De plus il décrit comment l'intégration de ces dépendances lors de la conception permet de combler le manque de suivi dans la documentation entre le concepteur et l'utilisateur, et comment il peut aider le concepteur du framework à créer son framework.

Chapitre 4

Utilisation des dépendances structurelles

Le but de ce chapitre est de présenter quelques-unes des dépendances constituant notre bibliothèque, d'exprimer des dépendances fréquentes (héritage, classe / méthode, création...) ainsi que de montrer l'applicabilité de notre modèle sur les schémas de conception. Cette bibliothèque constituée et complétée de schémas de conception, nous montrerons comment au fil du développement du framework le développeur peut utiliser les dépendances afin de documenter à la fois les points de variation et l'architecture de son framework. Enfin, une dernière section décrira les fonctionnalités d'un environnement qui mettrait en œuvre les dépendances structurelles.

4.1 Bibliothèque de dépendances

Afin de montrer quelques-unes des dépendances de notre bibliothèque, nous prenons comme support le schéma de conception Composite [GHJV95]. Il permet de retrouver un grand nombre de dépendances ontologiques comme l'héritage, la relation entre une méthode et une classe, etc. De plus il ne contient aucune dépendance existentielle contrairement à l'Abstract Factory ou au Visitor, ce qui le rend plus simple à appréhender.

Afin de créer les éléments conceptuels, nous nous sommes inspirés du schéma de conception abstrait [GHJV95, p. 164] qui fait ressortir trois rôles essentiels pour les classes (`Component`, `Leaf`, `Composite`) et quatre pour les méthodes (`Add`, `Remove`, `getChild` et `Operation`). Les éléments conceptuels créés, nous avons instancié les dépendances les concernant. La méta-représentation de ce schéma de conception est donnée figure 4.1, et les dépendances utilisées sont détaillées dans les sous-sections suivantes.

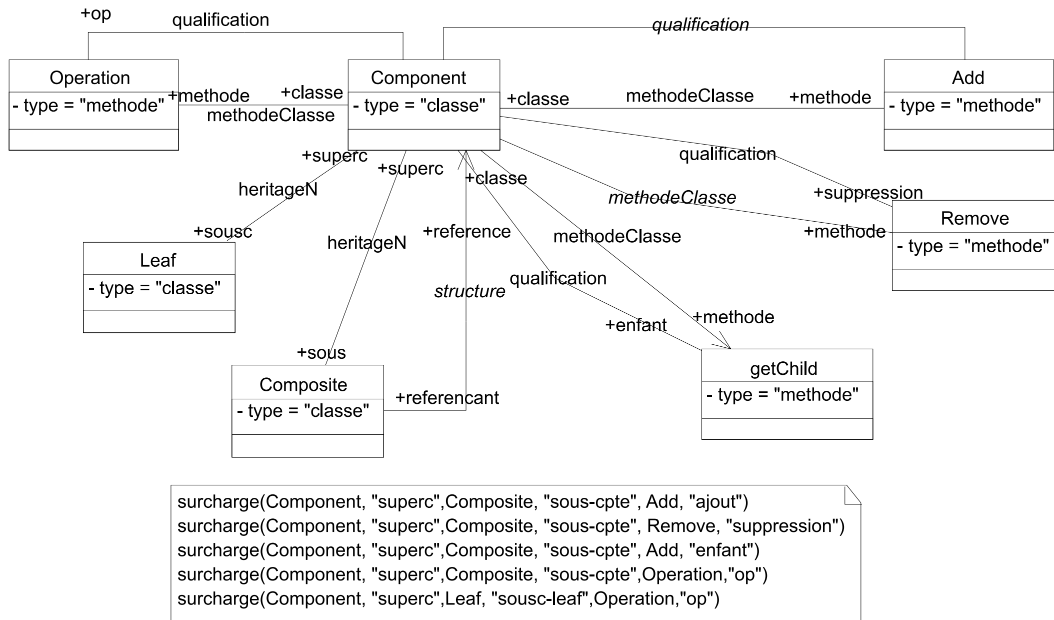


FIG. 4.1 – Méta-Représentation du schéma de conception composite.

4.1.1 Méthode / Classe

La dépendance ontologique `methodeClasse` (figure 4.2) assure au niveau F0 l'association entre une méthode et sa classe en créant un lien ad hoc. Aucune création n'est demandée, seuls les descripteurs de la méthode et de la classe sont liés. Ici, le rôle de la vérification est simplement d'assurer que chaque méthode est bien associée à une classe dont le rôle conceptuel est celui passé en second paramètre de la dépendance.

4.1.2 Qualification

Afin d'assurer la réutilisation des dépendances, leur expression doit éviter toute hypothèse concernant le nom, le type ou le rôle des éléments de F1.

Avec de telles hypothèses, on peut se demander dans l'exemple du schéma Composite, comment faire pour accéder à partir de la classe `Component` à la méthode `Add` ou `getChild`.

Nous proposons de marquer les éléments conceptuels de manière à y accéder sans ambiguïté. Pour cela, nous avons mis au point la relation `qualification` qui qualifie le rôle d'un élément par rapport à un autre, permettant ainsi de distinguer des éléments à la fois au niveau conceptuel et au niveau de la description. Le code de cette dépendance est montré figure 4.3. Elle est utilisée dans le schéma de conception Composite afin de distinguer les différentes méthodes : une `qualification` désigne la méthode qui gère l'`ajout`, une autre désigne la méthode qui gère la `suppression`, etc.

```
methodeClasse(ConceptualElement M, ConceptualElement C) {
   interaction {
    M.rm(nom) ->
      m = M.rm(nom);
      c = C.get("descripteurs").choose();
      new Lien(m,"methode",c,"classe");
   }
   verification {
    context M inv :
    M.get("descripteurs")->forAll(m |
      m.get("classe").size()==1 and
      m.get("classe").get("roleConceptuels")->exists(rc |
        rc == C))
   }
}
```

FIG. 4.2 – Dépendance connectant une méthode à sa classe.

```
qualification(ConceptualElement Ctx, ConceptualElement Qual,
              String qualif) {
   interaction {
    Qual.rm(nom) ->
      q = Qual.rm(nom);
      c = Ctx.get("descripteurs").choose();
      new Lien(c, "", q, qualif);
   }
}
```

FIG. 4.3 – Dépendance de qualification.

```
heritage1(ConceptualElement Sous, ConceptualElement Super) {
   interaction {
    Sous.rm(nom) ->
      ssc = Sous.rm(nom);
      spc = Super.get("descripteurs").choose();
      new Lien(ssc,"sousc",spc,"superc");
   }
   verification {
    context Sous inv :
    Sous.get("descripteurs")->forAll(sc |
      sc.get("superc").size()!= 0)
   }
}
```

FIG. 4.4 – Dépendance d'héritage.

4.1.3 Héritage

L'héritage, relation usuelle des langages à classes, est constitué de deux aspects complémentaires : l'un lié à la structure connectant une sous-classe à sa super-classe, et l'autre lié à un aspect « comportemental » de la relation qui force la redéfinition de certaines méthodes.

Aspect structurel de l'héritage

L'aspect structurel consiste simplement, lors de la création d'une sous-classe au niveau de la description du framework, à créer un lien la connectant à sa super-classe. Cette dépendance est présentée figure 4.4.

Lorsqu'elle est utilisée entre deux `conceptualElements` différents, elle limite les possibilités de création de sous-classe (au niveau F0) à une profondeur de 1. Cependant elle permet de séparer clairement l'élément conceptuel qui joue le rôle de super-classe de celui qui joue le rôle de sous-classe (niveau F1). Cette limitation est une conséquence de la liste des descripteurs proposés en choix à l'utilisateur. En effet, celle-ci ne contient que des descripteurs jouant le rôle de super-classe, et ne donne donc pas l'opportunité à l'utilisateur de connecter une nouvelle sous-classe à une sous-classe existant.

Si une séparation super-classe / sous-classe n'est pas nécessaire au niveau F1, on peut détourner la dépendance précédente de son but premier et l'utiliser pour la représentation d'une hiérarchie de profondeur illimitée en utilisant deux fois le même `conceptualElement` lors de l'instanciation de la dépendance. De fait, tous les descripteurs représentant des classes sont proposés à l'utilisateur.

Si la séparation super-classe / sous-classe au niveau F1 est inévitable on pourra utiliser la dépendance `heritageN` décrite figure 4.5, qui propose à l'utilisateur de choisir un descripteur parmi ceux qui jouent les rôles conceptuels de super-classe et sous-classe.

```
heritageN(ConceptualElement Sous, ConceptualElement Super) {
  interaction {
    Sous.rm(n) ->
      spc = (Super.get("descripteurs").
        union(Sous.get("descripteurs"))).choose();
      ssc = Sous.rm(n);
      new Lien(spc, "superc", ssc, "sousc");
  }
  verification {
    context Sous inv :
    Sous.get("descripteurs")->forAll(sc |
      sc.get("superc").size()!= 0)
  }
}
```

FIG. 4.5 – Dépendance d'héritage.

Aspect comportemental de l'héritage

Dans certains cas il peut être souhaitable de forcer la présence de certaines méthodes dans toute nouvelle sous-classe d'une classe donnée. Généralement cela est utilisé pour les méthodes abstraites mais nous l'avons étendu à toute sorte de méthodes. Ce mécanisme est séparable en deux dépendances : l'une qui lors de la création d'une nouvelle sous-classe force la redéfinition d'une certaine catégorie de méthodes (dépendance `aspiration` figure 4.6), l'autre qui lorsqu'une méthode est ajoutée dans une super-classe, crée la méthode dans toutes les sous-classes (dépendance `propagation` figure 4.7). Elles reçoivent toutes deux en paramètre le nom du rôle de la méthode qui doit être redéfinie (`RMet` dans la dépendance d'`aspiration` figure 4.6). La vérification assure que toute méthode devant être redéfinie se trouve bien dans les sous-classes.

La particularité de ces dépendances est que la surcharge d'une méthode n'introduit pas de nouvel élément conceptuel. Cependant la nouvelle méthode a besoin d'être créée ce qui explique la création « à la volée » d'un descripteur (figure 4.7). Nous verrons plus loin (sous-section 4.2.2) une extension de cette dépendance où l'opération redéfinie est représentée au niveau conceptuel.

C'est à des fins de clarté que nous avons présenté ces deux dépendances séparément. Réellement elles sont toutes deux définies dans une seule dépendance appelée `surcharge` dont la signature est donnée figure 4.8.

Étant donné le nombre d'éléments conceptuels mis en jeu dans la dépendance de `surcharge`, nous ne la représentons pas graphiquement, mais l'indiquons par une note. C'est le cas dans la représentation du schéma de conception Composite (figure 4.1)[1], où la dépendance est utilisée pour gérer la surcharge des éléments conceptuels représentant les méthodes `Add`, `Operation`, `Remove` et `getChild`. La ligne `surcharge( Component, "superc", Composite, "sous", Add, "ajout")` se lit de la ma-

[1]Pour des raisons que nous expliquons dans la sous-section suivante, c'est une variation de la dépendance surcharge (appelée surchargeN) que nous utilisons dans le schéma Composite.

```
aspiration(ConceptualElement Sous, ConceptualElement Super,
           String RSPC, String RMet)
  interaction {
   Sous.rm(nom) ->
     // On commence par laisser la réalisation se faire
     ssc = Sous.rm(nom);
     spc = Super.get("descripteurs").choose();

     //on duplique dans les sous-classes toutes les méthodes de
     la super-classe
     forAll m in spc.get(RMet) do
      new Lien( ssc, Sous.get("type"),
            new ElementDescriptor(m.get("nom"), m.get("type")),
            m.get("type") );
     done
  }
  verification {
   context Sous inv :
   Sous.get("descripteurs")->forAll(sc |
     sc.get(RSPC)->forAll( spc |
      spc.get(RMet)->( m |
       sc.get(RMet)->exists( scm |
        scm.get("name") == m.get("name")))))
 }
}
```

FIG. 4.6 – Dépendance de surcharge, partie aspiration.

```
propagation(String RSSC, ConceptualElement Super,
            ConceptualElement Methode, String RMet){
  interaction {
   Methode.rm(nom) ->
      m = Methode.rm(nom);
      spc = Super.get("descripteurs").choose();

      //On crée les méthode dans les sous-classes
      //on commence par les sous-classes immédiates
      x = spc.get(RSSC);
      forAll ssc in x do
        new Lien ( ssc, ssc.get("type),
              new ElementDescriptor(nom, Methode.get("type")),
              Methode.get("type") ),
        //on récupère les sous-classes de la classe courante
        x = x.union(ssc.get(RSSC));
      done
  }
  verification { //voir partie aspiration }
}
```

FIG. 4.7 – Dépendance de surcharge, partie propagation.

```
surcharge(ConceptualElement Super, String RSPC,
          ConceptualElement Sous, String RSSC,
          ConceptualElement Methode, String RMet)
```

FIG. 4.8 – Signature de la dépendance surcharge.

nière suivante : la méthode `Add`, qui est la méthode d'ajout définie dans la super-classe `Component`, est redéfinie dans la sous-classe `Composite`.

On voit ici l'utilité de la relation de `qualification`, puisqu'une classe composite peut impliquer la redéfinition des méthodes relatives au schéma de conception sans pour autant forcer la redéfinition d'autres méthodes.

4.1.4 Remarques sur la séparation des dépendances

Bien que nous ayons séparé l'héritage en deux dépendances différentes (`heritage` et `surcharge`), nous pensons que cette séparation n'est pas nécessairement pertinente car ces deux dépendances sont liées.

La conséquence de cette séparation est qu'afin d'être autonome chaque dépendance demande un contexte de réalisation (`choose()`). Cela est source d'incompatibilité entre `heritageN` et `surcharge` et montre bien la complémentarité d'`heritage1` et de `surcharge`. En effet, alors qu'`heritageN` propose tous les descripteurs à l'utilisateur, `surcharge` ne propose que ceux qui sont des super-classes. C'est aussi ceux que pro-

```
interaction {
   Sous.rm(nom) ->
     ssc = Sous.rm(nom);

     //premier demande ctx
     spc = Super.get("descripteurs").choose();
     new Lien(ssc,"sousc",spc,"superc");

     //seconde demande ctx
     spc = Super.get("descripteurs").choose();

     forAll m in spc.get(RMet) do
       new Lien(ssc, Sous.get("type"),
        new ElementDescriptor(m.get("nom"),m.get("type")));
     done
     //L'interaction sur Methode.rm(nom) reste inchangée
}
verification {
   context Sous inv :
   Sous.get("descripteurs")->forAll(sc
     | sc.get("superc").size()!= 0)

   context Sous inv :
   Sous.get("descripteurs")->forAll(sc |
     sc.get(RSPC)->forAll( spc |
       spc.get(RMet)->( m |
         sc.get(RMet)->exists( scm |
           scm.get("name") == m.get("name")))))
}
```

FIG. 4.9 – Résultat de la fusion d'héritage1, aspiration et propagation.

pose `heritage1`. Pour pallier ce problème nous avons créé une dépendance `surchargeN` de même signature que `surcharge` mais où les instructions `Super.get("descripteurs") .choose()` d'`aspiration` et de `propagation` ont été remplacées par `(Super. - get("descripteurs").union(Sous .get("descripteurs"))).choose()`. Ainsi, `surcharge` s'utilise de préférence avec `heritage1` et `surchargeN` avec `heritageN`.

L'autre problème posé par cette autonomie résulte de la fusion des dépendances. Comme nous l'avons expliqué sous-section 3.4.1, la fusion n'unifie que les messages situés en partie gauche des règles d'interactions. Ainsi pour `heritageN` et `surchargeN` elle ne laisse bien qu'un seul appel à `rm` mais laisse plusieurs demandes de contexte (figure 4.9) ce qui peut engendrer des incohérences si l'utilisateur ne choisit pas deux fois le même élément. Dans le cadre d'un environnement mettant en œuvre ce modèle, on peut supposer que cette demande multiple sera détectée et la valeur automatiquement remplie. Si l'on ne veut pas prendre de pari sur le comportement de l'environnement,

```
structure(ConceptualElement CElt1, String R1,
          ConceptualElement CElt2, String R2){
  interaction {
    CElt1.rm(nom) ->
      m = CElt1.rm(nom);
      c = CElt2.get("descripteurs").choose();
      new Lien(m,R1,c,R2);
  }
}
```

FIG. 4.10 – Dépendance de structure.

une dépendance unique (reprenant le résultat de la fusion, mais unifiant les choix) devra être écrite.

Même si cette séparation a les inconvénients mentionnés, elle reste nécessaire car on ne souhaite pas lier indissolublement l'héritage et la surcharge.

4.1.5 Structure

En analysant les relations `heritage1` (figure 4.4) et `classeMethode` (figure 4.2), on constate qu'elles ont le même contenu à la différence près des rôles qui connectent les éléments. Nous avons donc créé une relation `structure`, présentée figure 4.10, où le nom des rôles est paramétré, ce qui permet de représenter de manière générique une grande partie des dépendances ontologiques. Ainsi `structure(A,"sousc",B,"superc")` sera équivalent à `heritage1(A,B)` de même que `methodeClasse(M,C)` est équivalent à `structure(M,"methode", C, "classe").`

4.1.6 Conclusion sur la bibliothèque des dépendances

Avec ces dépendances qui constituent notre bibliothèque et l'exemple du schéma de conception Composite, nous avons vu comment créer des dépendances génériques utilisables dans plusieurs frameworks. Cette généricité, qui implique la suppression de toute hypothèse sur le nom ou le type des relations, est entre autre obtenue en utilisant la dépendance de qualification.

Nous avons aussi vu les avantages et inconvénients de séparer ou regrouper des dépendances dont le caractère peut être lié.

4.2 Représentation et utilisation de schémas de conception

Les frameworks étant constitués à plus petite échelle de schémas de conception, notre modèle en permet donc la modélisation. Cette étude est particulièrement intéressante car tout comme le modèle présenté dans [RF00], notre modèle permet de capturer

la structure et les dépendances du schéma de conception, et permet d'en automatiser l'application.

Cette expression des dépendances permet de mieux comprendre comment chaque élément d'un schéma de conception est relié aux autres, information qui est parfois difficile à trouver dans son texte. En effet, quel que soit le style adopté (GoF, Buschmann, PLoP...) pour leur représentation, ils souffrent d'une dilution de l'information pertinente dans le texte et d'une difficulté à exprimer leur caractère générique.

Pour illustrer ces problèmes nous utilisons le schéma Abstract Factory. Ainsi le problème d'expression de la généricité peut être constaté dans le diagramme de classes de la page 88 du GoF, où un suffixe sous forme de chiffres et de lettres est utilisé pour montrer la correspondance entre les noms des éléments. C'est bien là le besoin d'une information d'un ordre supérieur. De plus il faut déduire de certaines explications spécifiques à un exemple un fonctionnement plus générique du schéma de conception : *« There is a concrete subclass of WidgetFactory for each look and feel standard. »* [GHJV95, p. 88]. On trouve ensuite d'autres informations sur le fonctionnement du schéma ou sa structure un peu partout : *« ConcreteFactory implements the operations to create concrete product objects. »*[GHJV95] ; *« Normally a single instance of a ConcreteFactory is created at run-time. »* ; *« AbstractFactory defers creation of product objects to its ConcreteFactory subclass. »*[GHJV95]

Ainsi, notre modèle regroupe ces informations structurelles et les dépendances entre les éléments, cependant il est toujours conseillé de consulter l'ouvrage de référence et de lire les informations comme les schémas liés puisque ces informations ne sont pas retenues dans notre modèle. On peut imaginer que de telles informations le soient dans le cadre d'un outil dédié aux schémas de conception, mais ce n'est pas notre objectif.

Nous ne présentons pas ici tous les schémas de conception du GoF. L'ensemble que nous présentons a été sélectionné en utilisant les critères suivants : la difficulté reconnue à appréhender le schéma (Abstract Factory, Visitor), son appartenance à l'ensemble des schémas fondamentaux [AC98] (qui sont eux aussi trop nombreux (17) pour être détaillés ici), et les schémas présents dans JUnit (Composite, Template).

4.2.1 Abstract Factory

Afin de représenter ce schéma de conception nous nous sommes inspirés de sa représentation abstraite donnée page 88 du GoF. Ainsi, sa réification fait apparaître une hiérarchie d'usine (`AbstractFactory`, `ConcreteFactory`), des hiérarchies de produits (`AbstractProduct`, `ConcreteProduct`) et la méthode de création des produits. Ces éléments sont tous connectés par des dépendances déjà connues, et la redéfinition de la méthode `CreateProduct`, est gérée par la dépendance `surchargeN`.

Une dépendance `creation` a été utilisée entre `CreateProduct` et `AbstractFactory`, et `ConcreteFactory` et `ConcreteProduct`. Il s'agit d'une généralisation de la dépendance utilisée dans l'exemple des JavaBeans. Son contenu est présenté figure 4.12. Cette dépendance est utilisée pour indiquer que l'ajout d'une méthode de création de produits (`CreateProduct`) implique la création d'une hiérarchie de produits associée (`AbstractProduct`). On notera l'existence de la règle de création inverse, ce qui permet à l'utilisateur final de pouvoir étendre son schéma à la fois en ajoutant une nouvelle hiérarchie de produits ou une nouvelle méthode de création de produits. Afin

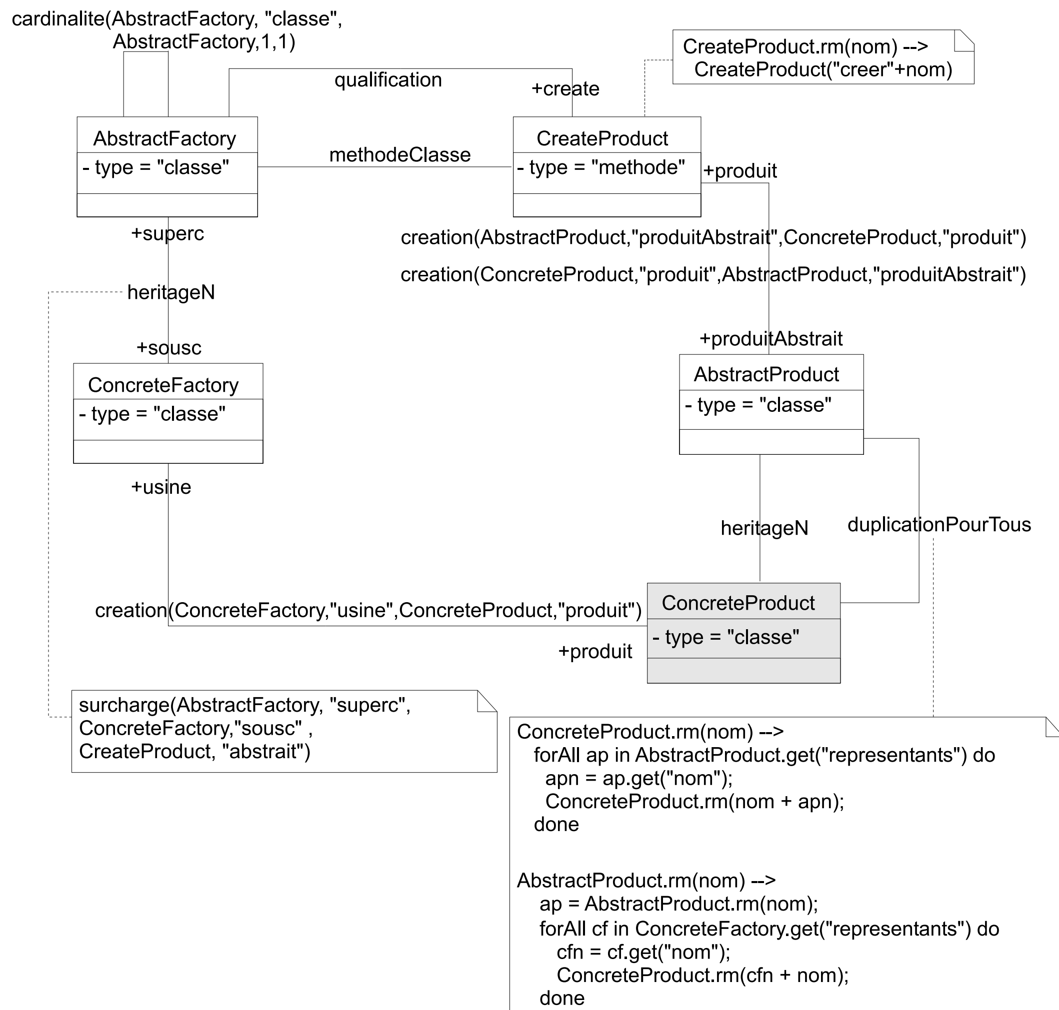


FIG. 4.11 – Modélisation des dépendances du schéma de conception Abstract Factory.

```
creation(ConceptualElement CElt1, String r1,
         ConceptualElement CElt2, String r2) {
 interaction {
   CElt1.rm(nom) ->
     x = CElt1.rm(nom);
     y = CElt2.rm(nom);
     new Lien(x, r1, y, r2);
 }
}
```

FIG. 4.12 – Dépendance existentielle.

```
cardinalite(ConceptualElement MElt1, String R1,
        ConceptualElement MElt2, int cardMin, int cardMax) {
   interaction {
    MElt1.rm(nom) ->
      elt2 = MElt2.get("descripteurs").choose();
      v = elt2.get(R1).size();
      if ( v >= cardMin and v <= cardMax )
       MElt1.rm(nom);
      else
       throw ERROR;
   }
   verification {
    context MElt2 inv :
    MElt2.get("descripteurs")->forAll(elt2 |
      elt2.get(R1).size() >= cardMin and
      elt2.get(R1).size() <= cardMin)
   }
}
```

FIG. 4.13 – Dépendance de cardinalité.

que l'utilisateur ne puisse pas demander la réalisation de `ConcreteProduct`, cette classe est cachée, ce qui explique le fond grisé de cet élément conceptuel. Cependant cela n'empêche pas que la réalisation se fasse en réaction à une dépendance.

Afin de gérer l'unicité de la classe `AbstractFactory` nous avons créé une abstraction de la cardinalité qui était initialement présentée section 3.3, figure 3.15. Cette abstraction est présentée figure 4.13.

La seule dépendance qui a nécessité l'écriture complète de règle de création et de vérification est celle qui, lors de la création d'une nouvelle hiérarchie de produits, recrée cette hiérarchie à l'image des autres. Le corps de cette dépendance (`duplicationPourTous`) est intégré à la modélisation du schéma (figure 4.11). Afin de calculer le nom des classes et des méthodes, cette dépendance utilise des fonctions usuelles de manipulation des chaînes de caractères. Grâce à cela, nous forçons le respect d'une convention de nommage. En effet, toutes les méthodes de création seront préfixées de « `creer` », le sommet d'une hiérarchie de produits portera le nom du produit, et les sous-classes auront le nom de l'usine qui les crée suivi du nom du produit. Le nom des usines n'est pas controlé.

Cette méta-description de l'AbstractFactory montre comment grâce aux dépendances déjà présentées et à quelques dépendances spécifiques on peut rapidement exprimer l'architecture d'un tel schéma de conception. De plus, on voit comment les dépendances offrent un contrôle sur la structure comme par exemple le nommage des entités.

4.2.2 Visitor

Ce schéma de conception est probablement, à cause de sa double hiérarchie de classes et de son comportement basé sur le mécanisme de multi-méthode, un des schémas de conception les plus compliqués à comprendre puisqu'il fait une séparation entre la structure d'une hiérarchie d'éléments et le traitement qu'on lui applique.

Son texte et les diagrammes associés font ressortir quatre acteurs majeurs : la hiérarchie des visiteurs (`Visitor`), la hiérarchie des éléments (`Elt`), la méthode Accept (`Accept`) et la méthode de traitement (`VisitConcreteElt`). Ces éléments constituent donc à quelques détails près toute l'architecture de la réification du schéma de conception qui est présentée figure 4.14.

Les dépendances existentielles que nous présentons ont été trouvées par l'analyse des exemples et déduites de phrases du texte explicatif du schéma comme : « *Visitor declares a visit operation for each class of ConcreteElement* » [GHJV95], ou encore « *ConcreteVisitor declares each operation declared by Visitor...* » [GHJV95, p. 334].

Afin de contraster avec la représentation de l'Abstract Factory où nous avons utilisé de nombreuses petites dépendances, ici nous avons choisi de les regrouper. Ainsi, en constatant que le nom du `Visitor` était le même que celui de l'élément mais suffixé de « `visitor` » et que la méthode `accept` avait pour type de paramètre le sommet de la hiérarchie des `Visitors`, nous avons créé une unique dépendance qui sur la réalisation d'`Elt` engendre la réalisation de `Visitor`, la réalisation de la méthode `Accept`, et la réalisation de son paramètre (`ParamAcc`). Cette dépendance, nommée `creationElt` est présentée dans la figure 4.14 sous une forme instanciée.

L'autre dépendance est liée à la création du `ConcreteElement`. La réalisation de cet élément cause la réalisation d'une nouvelle méthode `VisitConcreteElt` et aussi la réalisation de l'appel de méthode qui constitue le corps de la méthode `CAccept` des `ConcreteElts`. Cela se traduit par la dépendance nommée `creationConcreteElt` qui est explicitée sous une forme instanciée figure 4.14.

Dans ce schéma nous avons un cas particulier de surcharge. En effet, contrairement au cas où nous avions utilisé `surchargeN`, ici la méthode redéfinie joue un rôle particulier dans les sous-classes : elle fait un appel. Afin de représenter cette information liée à l'appel, il est nécessaire de réifier à la fois la méthode accept (`CAccept`), et l'appel qu'elle effectue (`AppelVisitMethod`). `CAccept` étant la redéfinition d'`Accept` il semble adapté de vouloir utiliser `surchargeN` (sous-section 4.1.3). Cependant, puisque cette dépendance crée des descripteurs sans utiliser d'élément conceptuel, nous ne pouvons pas l'utiliser. Nous avons donc créé une nouvelle dépendance `surchargeNExp` (figure 4.15) ayant le même comportement que `surchargeN` mais réalisant des descripteurs d'un élément conceptuel donné pour représenter les méthodes surchargées.

Grâce à la méta-description de ce schéma de conception, nous avons vu qu'il était possible tout en limitant les possibilités de l'utilisateur final en masquant un grand nombre d'éléments de gérer un schéma complexe. Cela nous conforte donc dans l'idée qu'une recette d'un *cookbook* peut aisément être représentée en utilisant les dépendances afin de guider au plus près l'utilisateur.

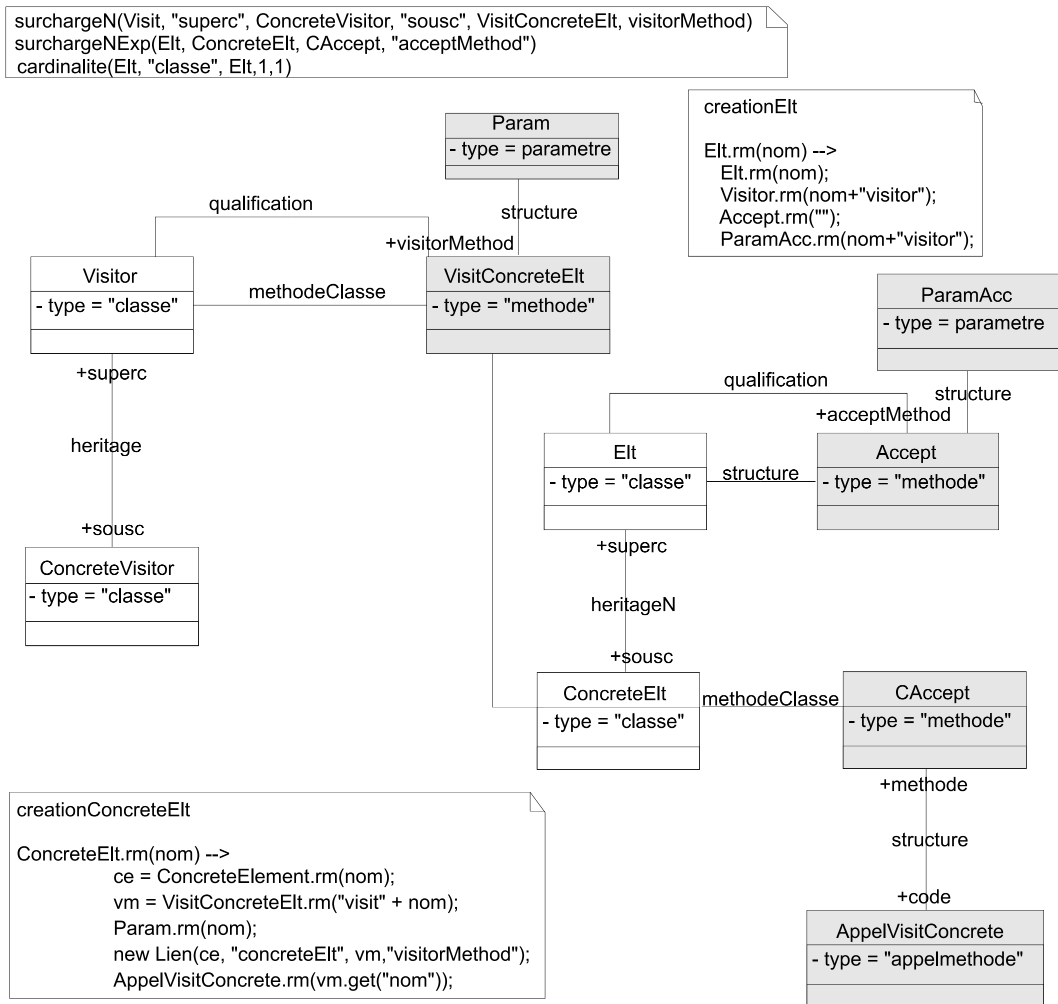


FIG. 4.14 – Méta-représentation du schéma Visitor.

```
surchargeNExp(ConceptualElement Superc, ConceptualElement Sousc,
              ConceptualElement MSC, String RM) {
  Sousc.rm(nom) ->
    ssc = Sousc.rm(nom);
    spc = (Superc.get("descripteurs").
      union(Sousc.get("descripteurs"))).choose();

    forAll m in spc.get(RM) do
      new Lien(ssc,"classe",MSC.rm(m.get("nom")),"methode");
    done
}
```

FIG. 4.15 – Extension de surcharge avec élément conceptuel pour la méthode.

4.2.3 Template Method

Le schéma de conception Template Method montre que l'on peut modéliser des schémas de conception sans nécessairement avoir à définir de nouvelles dépendances. Ainsi, la représentation donnée figure 4.16 ne contient que des dépendances constituant notre bibliothèque.

4.2.4 Conclusion sur la représentation des schémas de conception à l'aide de dépendances

Nous avons ici présenté les schémas de conception comme nous représentons les dépendances dans un framework. Au-delà de l'exercice de style, cette représentation avait pour but de montrer les capacités du modèle et de présenter les schémas de conception comme une dépendance accroissant la bibliothèque des dépendances. Ces dépendances pour les schémas de conception, tout comme les dépendances rencontrées précédemment, sont uniquement constituées de règles de création et de vérification et ne contiennent aucune information sur les types des éléments qu'ils connectent.

Grâce aux exemples présentés, on voit qu'il n'y a pas une unique représentation des dépendances d'un framework et que chacune de ces représentations offre des avantages comme la réutilisation maximale de dépendance, ou la facilité du suivi du processus de réalisation lorsqu'une dépendance spécifique est créée.

4.3 Scénario d'utilisation du Visitor

Afin de bien mesurer l'implication de la représentation à base de dépendances structurelles, nous donnons un scénario d'utilisation du schéma de conception Visitor par un utilisateur final. Le scénario reproduit l'exemple utilisé dans le GoF, c'est-à-dire la construction d'un visiteur d'arbre de syntaxe abstraite. Les étapes de construction sont :

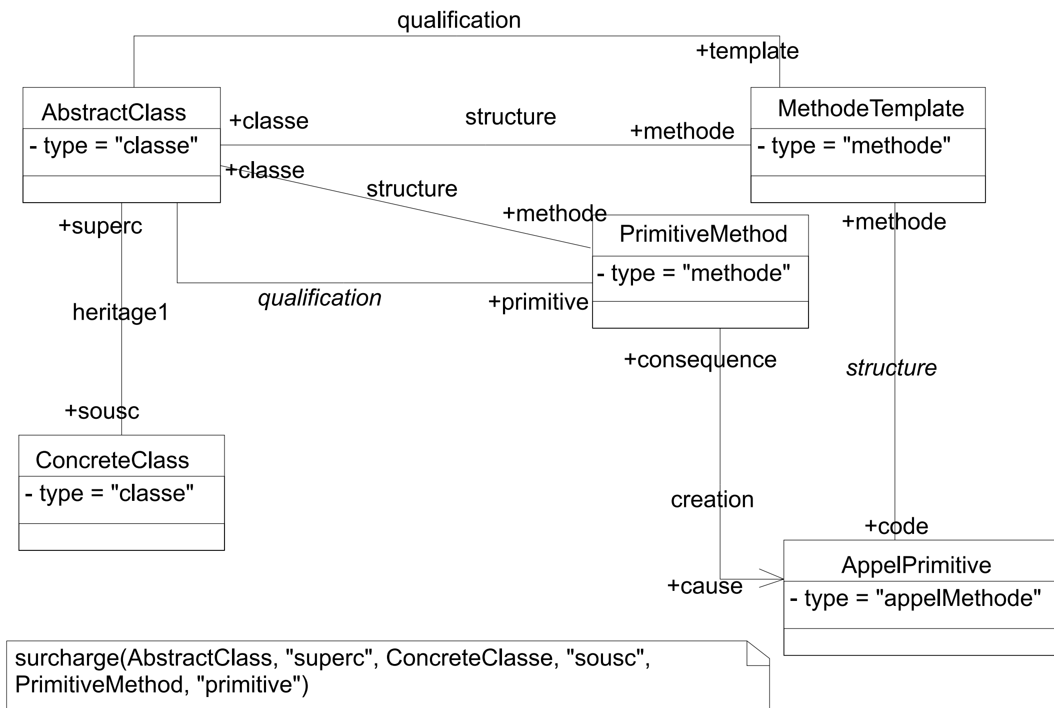


FIG. 4.16 – Méta-représentation du schéma template Method.

1. l'utilisateur crée une classe appelée `Node` jouant le rôle d'`Element`, la tête de la hiérarchie de classes qu'il faudra visiter (figure 4.17(a)). La création de cette classe engendre la création de la tête de la hiérarchie représentant `Visitor`. Cette classe porte donc le nom de `NodeVisitor`. La méthode `accept` est ensuite créée, ce qui entraîne la création du paramètre avec la valeur `NodeVisitor` ;
2. l'utilisateur ajoute deux sous-classes à la hiérarchie `NodeVisitor` : une première nommée `TypeCheckingVisitor` et une seconde `CodeGenVisitor` (figure 4.17(b)). Pour ces deux classes la dépendance `surchargeN` est réalisée, cependant aucun élément n'est créé puisque `NodeVisitor` n'a pas encore de méthode ;
3. une nouvelle classe jouant le rôle de `ConcreteElt` est créée. Il s'agit d'`AssignmentNode`. Puisque ce `ConcreteElt` est connecté à une dépendance d'héritage et de surcharge, la classe `AssignmentNode` est connectée comme sous-classe de `Node`, et une méthode `accept` y est ajoutée. De plus, la méthode permettant la visite du nouvel élément créé est ajoutée à la classe `NodeVisitor`. Puis grâce à la surcharge, elle est redéfinie dans `ConcreteVisitor`. Dans un dernier temps, le corps de la méthode `accept` créé dans `ConcreteElt` est rempli (`AppelVisitMethod` est réalisé).

Grâce à cet exemple, et la modélisation directive des dépendances des schémas de conception, on voit comment les recettes d'un *cookbook* pourraient être automatisées.

4.4 Intégration au cycle de vie

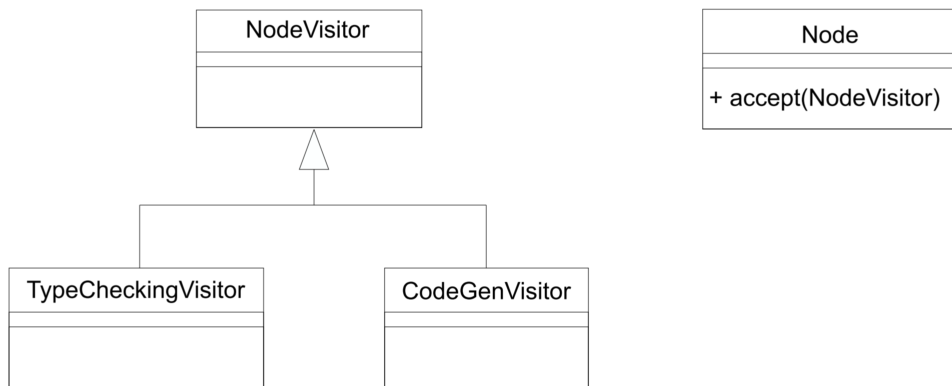
Le but de cette section est double. Elle détaille à la fois comment les dépendances structurelles s'utilisent pendant le développement d'un framework, et comment les schémas de conception présentés précédemment peuvent être réutilisés pour documenter JUnit. Contrairement à la section précédente, nous nous plaçons donc à présent comme développeur d'un framework. Bien que le scénario utilisé ne soit pas réaliste, puisqu'un framework est construit par généralisation d'applications, il permet de voir les situations auxquelles est confronté un développeur.

Ainsi nous retrouvons notre développeur alors qu'il conçoit l'architecture de son framework. Ce scénario nous est inspiré du *JUnit cook's tour* [GB].

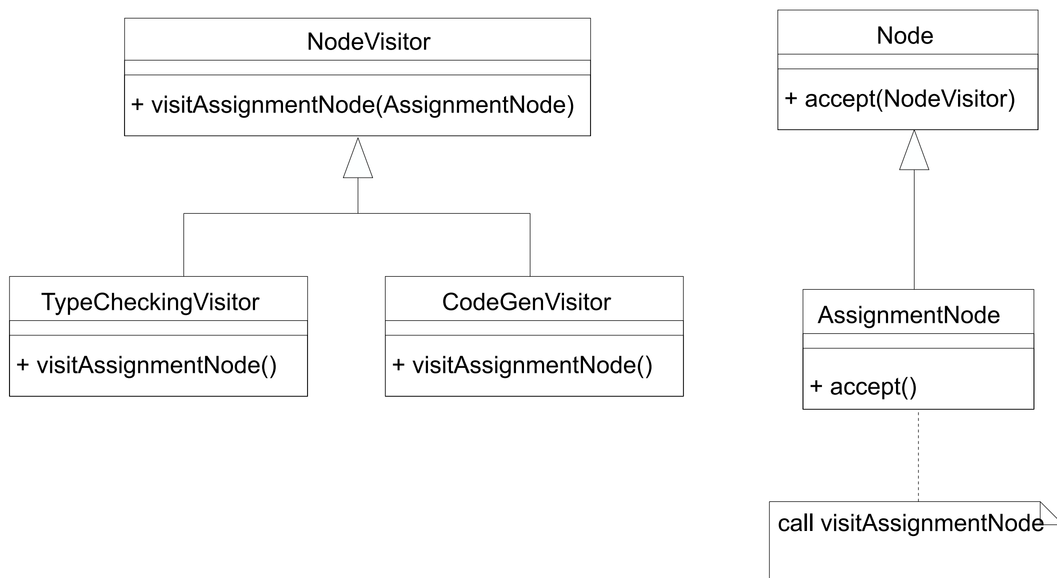
TestCase

Afin de créer les bases de son framework, le développeur commence donc par créer une classe `TestCase` contenant une méthode `run`. Lorsqu'il prend conscience que les tests sont constitués d'un début et d'une fin, il décide de mettre en œuvre le schéma de conception Template Method. Afin de l'expliciter au niveau méta-framework, le développeur commence par réifier les éléments qui l'intéresse dans la classe `TestCase` (figure 4.18). Ainsi il crée un élément conceptuel pour la classe `TestCase` mais aussi un pour représenter les sous-classes (`SousTestCase`). Pour les méthodes deux possibilités sont envisageables (figure 4.19) : la première qui consiste à créer un élément conceptuel par méthode (figure 4.19(a)), ou plus simplement une pour les méthodes qui jouent le rôle de primitive dans le schéma et une pour la méthode template (4.19(b)), conformément à la représentation abstraite du schéma de conception.

(a) Étape 1



(b) Étape 2



(c) Étape 3

FIG. 4.17 – Étapes de l'utilisation du schéma Visitor.

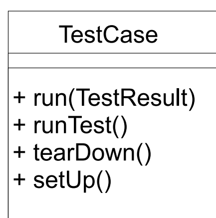
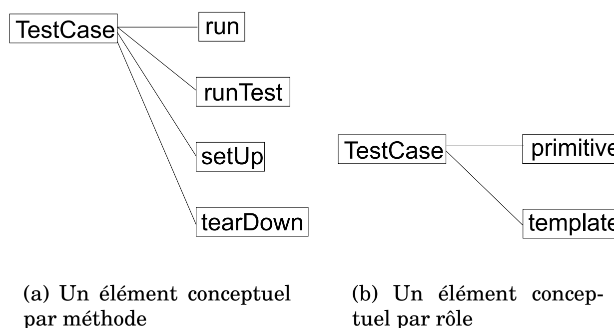


FIG. 4.18 – Classe TestCase.



(a) Un élément conceptuel par méthode

(b) Un élément conceptuel par rôle

FIG. 4.19 – Possibilités de réification pour les méthodes de TestCase.

Dans le contexte de JUnit, nous avons choisi la deuxième solution. Celle-ci est adaptée car les méthodes `runTest`, `setUp` et `tearDown` ne jouent pas un rôle particulier requérant un représentant particulier au niveau méta. De surcroît, cette solution permet de réutiliser les dépendances du schéma Template Methode tel quel. Il suffit de les instancier avec `TestCase`, `primitive` et `template`. Ici on constate donc que le schéma de conception peut être réutilisé comme une entité à trou apportant un ensemble de dépendances.

Notre développeur a commencé par créer les éléments de son framework (`Testcase`, `runTest`...) avant de le réifier. Ainsi il doit maintenant associer ces éléments aux rôles conceptuels qu'ils jouent dans le framework (figure 4.20). On notera que dans ce cas où la dépendance est principalement utilisée pour documenter le framework, le méta-framework est livré en partie réalisé.

TestResult

Afin de gérer les résultats de ses tests le développeur décide de créer une classe collectant les résultats (`TestResult`). Cette classe, composée des méthodes `startTest`, `endTest`, `addError`, n'est initialement pas faite pour être étendue. Après avoir constaté qu'il pouvait être nécessaire de particulariser les rapports de tests il décide d'implémenter cette variabilité par un schéma de conception Strategy. Cependant il voit que ce qui lui importe est le contrôle de la création d'une nouvelle sous-classe de manière à forcer la redéfinition des méthodes `result`, `startTest`, `endTest`.

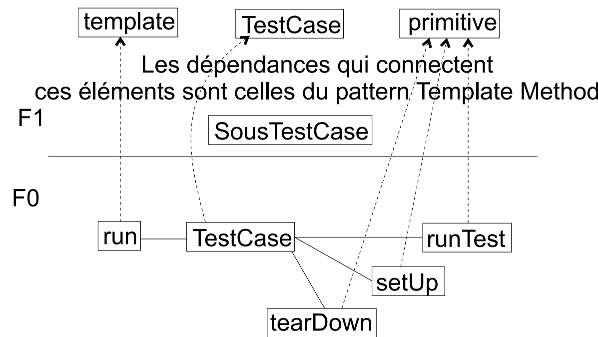


FIG. 4.20 – Template Method et sa réalisation dans JUnit.

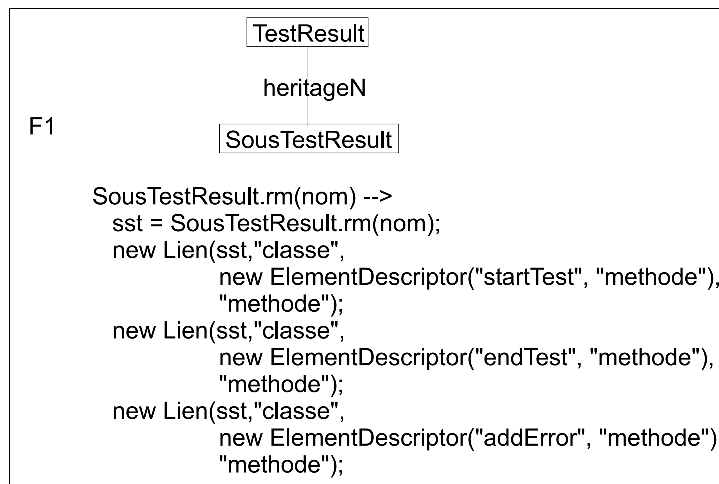


FIG. 4.21 – Dépendance pour la gestion des résultats.

Pour cette classe `TestResult`, l'utilisateur qui ne désire pas insister sur le schéma de conception, a choisi de définir une nouvelle dépendance mettant en place le comportement décrit précédemment. Il crée donc les éléments conceptuels et la dépendance qui sont présentés figure 4.21.

Pour cette dépendance seules les classes ont été réifiées. En effet, la représentation des méthodes n'apporterait rien puisqu'elles ne rentrent pas réellement en jeu dans les dépendances. Ainsi, tout comme dans la dépendance de `surchargeN` les éléments engendrés ne sont pas représentés parmi les éléments conceptuels.

Cela exemplifie la liberté offerte au développeur du framework, dans l'expression des dépendances structurelles. On voit ici qu'il a préféré créer une dépendance spécifique au point de variation, plus à la manière des *Hooks*, plutôt qu'utiliser le schéma de conception qui n'est pas absolument nécessaire pour savoir utiliser la classe. Bien que cette dépendance permette d'apporter de l'assistance à un utilisateur débutant, elle ne sera pas d'un grand secours pour un mainteneur puisqu'il n'aura pas l'information du schéma utilisé.

TestSuite

Dans un dernier temps le développeur décide d'introduire la notion de suite de tests. Pour cela il sait qu'il veut utiliser le schéma Composite et commence donc par créer un élément conceptuel représentant `Leaf`, `Composite` et la méthode `add`. Il décide alors d'utiliser la représentation qu'il a en sa possession pour gérer les dépendances. Cependant il se trouve confronté à l'inadéquation des dépendances par rapport à son besoin. En effet, il n'a besoin que d'une méthode `add` qu'il veut déclarer dans la classe `Composite` et non dans la classe du `Component` comme le préconise la représentation du schéma qu'il désire utiliser.

Notre développeur doit donc se résoudre à chercher de nouvelles dépendances dans sa bibliothèque ou à en écrire une autre. Cela fait, il pourra utiliser sa méta-représentation pour générer les descripteurs du framework grâce à l'opération de réalisation. En effet bien que ce ne soit pas le but initial de cette opération, elle peut être utilisée pour gagner du temps lorsque de nombreux éléments doivent être créés et ainsi éviter la phase d'association entre les descripteurs et les éléments conceptuels quand un méta-framework est créé après un framework.

4.4.1 Conclusion sur l'intégration des dépendances structurelles dans le cycle de vie

Cette section a permis de voir comment les dépendances pouvaient être utilisées afin de documenter un framework. De plus elles ont montré comment elles pouvaient faciliter la tâche du développeur à la fois dans le développement et la documentation de son framework, mais aussi comment elles modifiaient les habitudes de développement. Si l'on compare la méta-description donnée figure 4.22 avec le framework physique, on constate que l'intégralité du framework n'a pas fait l'objet d'une description et d'une méta-description. C'est par exemple le cas pour les méthodes de `TestResult`, ou encore la relation qui connecte `TestResult` à `TestCase`.

Nous avons aussi montré le compromis qui était toujours possible dans la modélisation d'une dépendance. Le développeur peut au choix modéliser de manière très générale une dépendance, ou à l'opposé écrire une dépendance dédiée à un problème spécifique. Comme nous l'avons vu, ce compromis a des répercussions sur l'assistance fournie aux utilisateurs.

4.5 Outils

Bien que la création d'outils ne fasse pas partie de notre but initial, nous pensons que dans sa forme actuelle notre modèle peut donner lieu à un environnement de développement orienté framework. Ainsi, dans une première sous-section, nous spécifions les fonctionnalités d'un tel environnement puis décrivons dans une seconde sous-section la mise en œuvre effective du modèle.

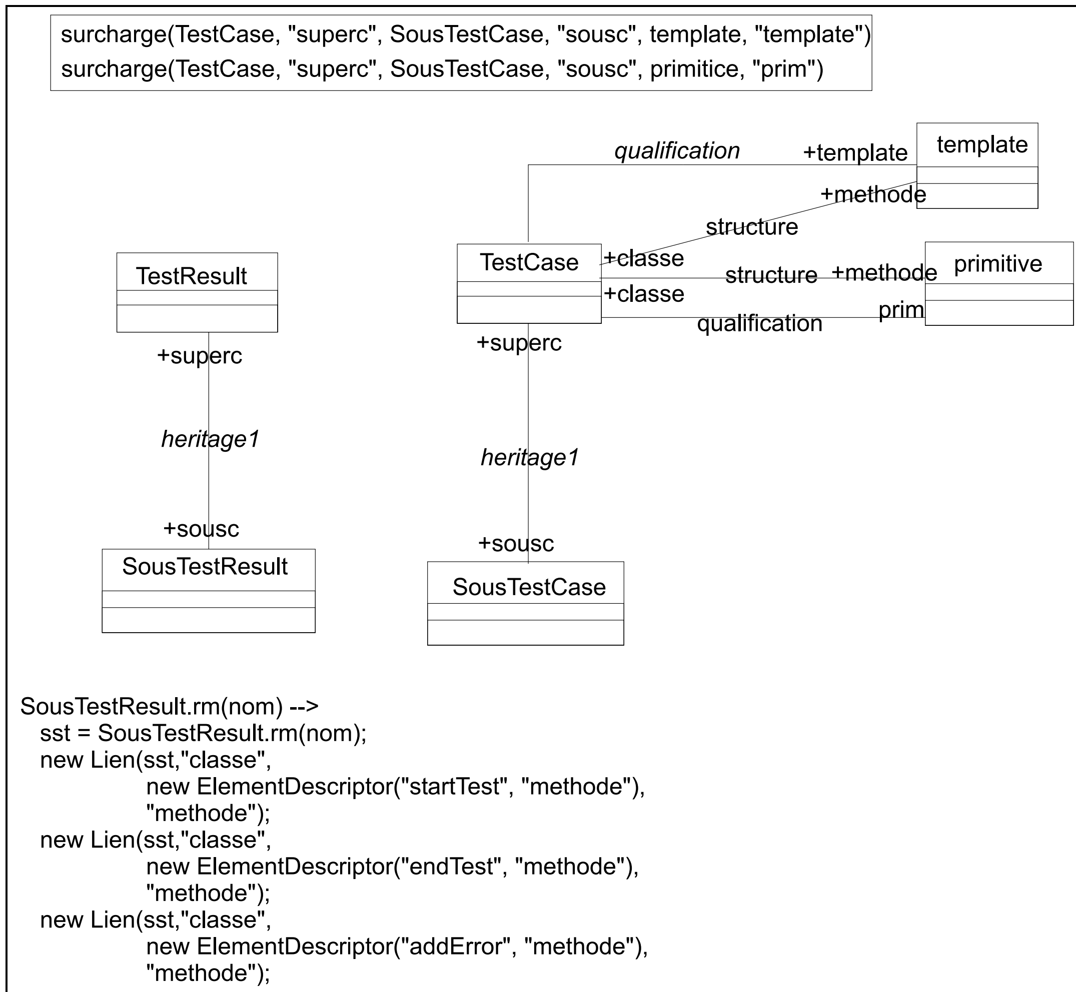


FIG. 4.22 – Méta-framework associé à JUnit.

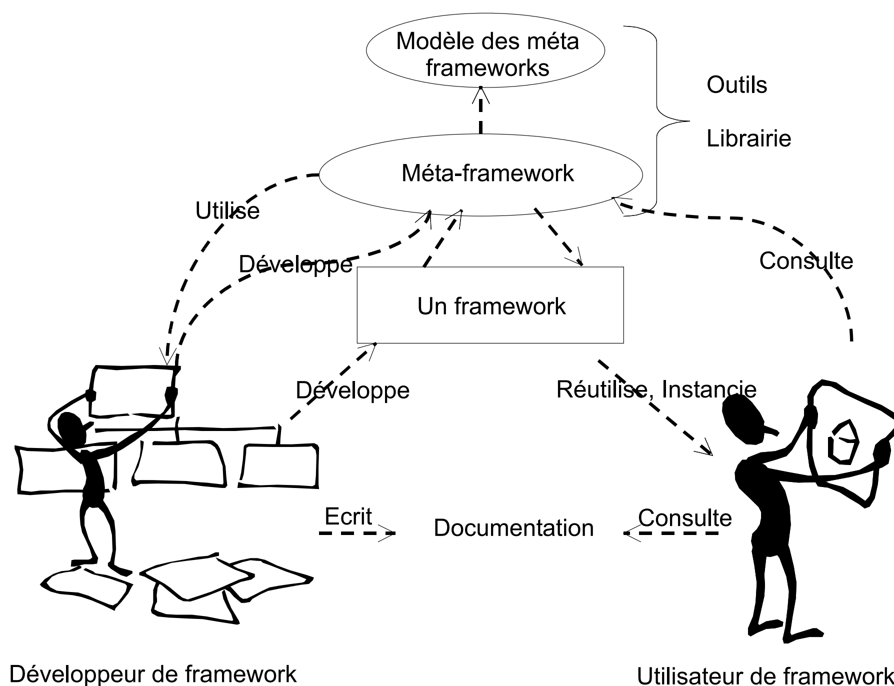


FIG. 4.23 – Framework, méta-framework et rôles associés.

4.5.1 Environnement de développement orienté framework

À partir de ce modèle nous avons imaginé, mais pas encore implémenté un environnement facilitant le développement et l'utilisation des frameworks, et cela en utilisant principalement le méta-framework et le framework qu'il représente.

Les deux types d'utilisateur ayant des besoins différents, l'environnement sera constitué de deux facettes offrant chacune des fonctionnalités différentes. Ainsi, celle de l'utilisateur mettra l'accent sur une navigabilité accrue entre le méta-framework et le framework, et offrira des fonctionnalités d'associations de rôles conceptuels ; alors que celle du développeur, munie évidemment de fonctionnalités de navigation offrira des outils de manipulation pour les bibliothèques de dépendances et l'instanciation de dépendances. La figure présente schématiquement pour chaque acteur l'utilisation qui est faite des données.

L'outil décrit ci-après s'apparente à un outil de conception assisté par ordinateur (CASE) puisqu'il présente le framework sous-forme de diagrammes de classes.

Facette utilisateur

La facette utilisateur présente une vue focalisée autour du framework. Sa fonctionnalité principale est de fournir une assistance active (réalisation) ou passive (vérification) à son utilisateur, assistance évidemment basée sur le méta-framework qu'il ne pourra que consulter.

Réalisation. Afin de créer une application à partir d'un framework l'utilisateur aura deux possibilités :

- la première, qui semble la plus classique, consistera en la création d'extensions du framework. Ainsi, en étendant son framework l'utilisateur recevra l'assistance en fonction des dépendances associées au framework. Idéalement l'identification de l'élément créé sera faite automatiquement par rapport à son contexte de création (par exemple une sous-classe, une méthode, etc.). En cas d'ambiguïté le rôle conceptuel de l'élément sera demandé à l'utilisateur (ce genre d'ambiguïté se rencontre par exemple lors de l'ajout d'une méthode dans le schéma Template Method). Lorsque l'utilisateur créera un élément et qu'aucune aide ne sera possible, l'association avec un élément conceptuel sera toujours possible après coup ;
- la seconde consistera à demander explicitement, en accédant au méta-framework, la réalisation d'éléments conceptuels. Cette fonctionnalité ne sera probablement utilisable que par des utilisateurs avancés puisqu'elle requiert la connaissance du rôle conceptuel de l'élément que l'on désire créer. Cette approche sera utile lorsque de nombreux éléments doivent être créés, et offre l'avantage de créer automatiquement les éléments.

Vérification. Alors que la réalisation était implicite, la vérification devra être demandée par l'utilisateur. Pour cela l'utilisateur sélectionnera un élément de son framework (ou du méta-framework) puis en demandera la vérification entraînant uniquement la vérification de la dépendance sélectionnée. Bien sûr il pourra demander une vérification complète.

Si le framework a été créé sans utiliser la réalisation, l'utilisateur devra auparavant identifier les rôles de chacun des éléments de son framework.

Interface graphique. Le support de ces opérations sera fourni à travers deux perspectives[2] : l'une présentant le méta-framework (figure 4.24), l'autre présentant le framework (figure 4.25).

Chacune des vues présentant le framework disposera des fonctionnalités pour passer de la représentation framework à la représentation méta-framework et réciproquement. Afin d'éviter les aller-retours entre les deux représentations, une vue située dans la partie inférieure de la perspective montrera des informations relatives à l'élément conceptuel ou au descripteur sélectionné.

Une fonctionnalité au niveau du framework permettra d'avoir en superposition sur le framework les dépendances qui sont réalisées et les éléments qui prennent part à cette dépendance. On aura donc une sorte de vue mixte entre le niveau méta et le framework.

Afin de différencier le framework initial des classes créées par l'utilisateur, une technique de coloration des classes [CLL99] sera utilisée.

[2]Une perspective définit un ensemble initial de vues et leur répartition [OTI01].

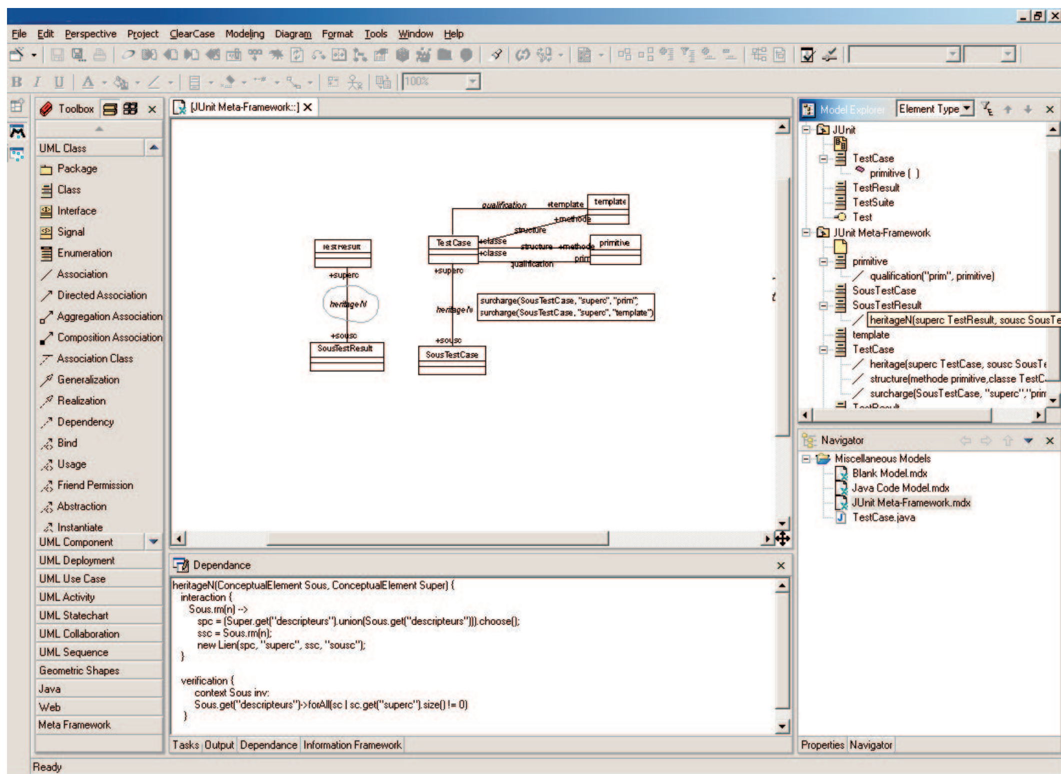


FIG. 4.24 – Perspective permettant la manipulation d'un méta-framework.

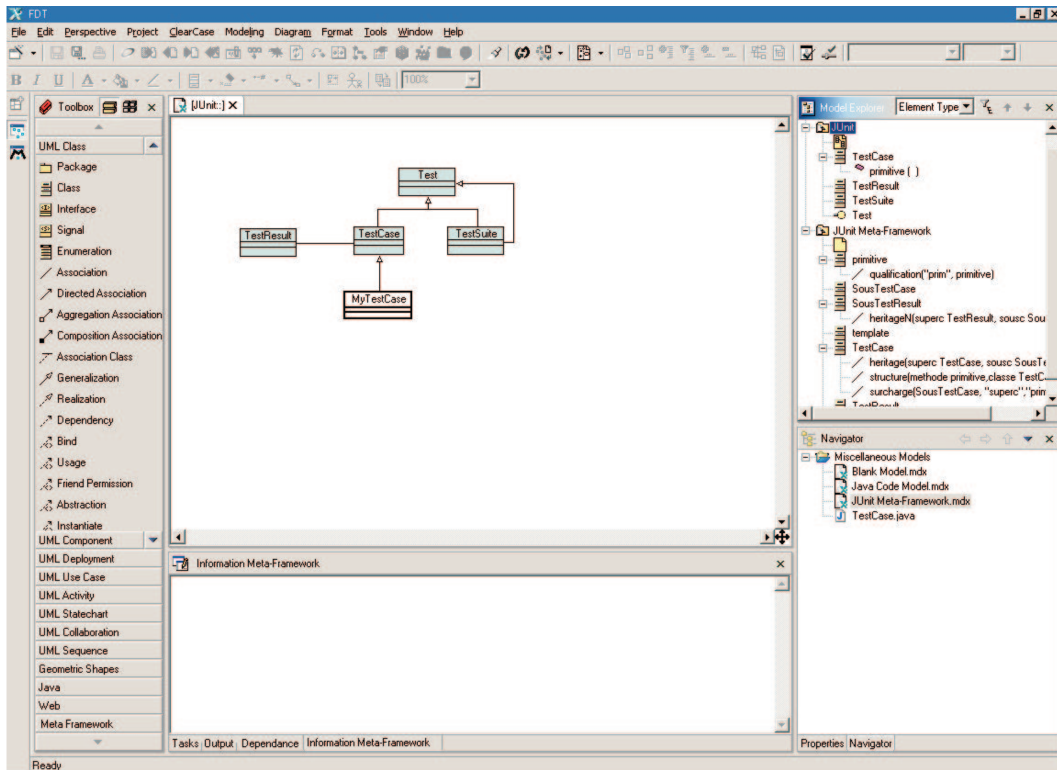


FIG. 4.25 – Perspective permettant la manipulation d'un framework.

Facette développeur

La facette développeur a pour but de faciliter le développement d'un framework et de son méta-framework en permettant la réutilisation de dépendances.

La facette du développeur reprendra les fonctionnalités proposées dans la facette de l'utilisateur (avec les droits en écriture et lecture) avec en plus un ensemble de fonctionnalités permettant la création du méta-framework, la recherche de dépendances, la gestion de la bibliothèque de dépendances, etc.

Création d'un méta-framework. La création des méta-frameworks pourra être abordée de deux manières :

- en réifiant les éléments pertinents du framework. Afin que cette réification ne soit pas trop fastidieuse, un outil créant les descripteurs à partir du framework sera fourni. Ainsi l'utilisateur n'aura plus qu'à instancier (ou créer) des dépendances de sa bibliothèque entre les éléments ainsi réifiés ;
- en créant explicitement les éléments conceptuels de son méta-framework. Pour créer son framework l'utilisateur n'aura plus qu'à réaliser les éléments qu'il vient de créer et suivre à la manière d'un utilisateur final l'assistance qui lui sera fournie.

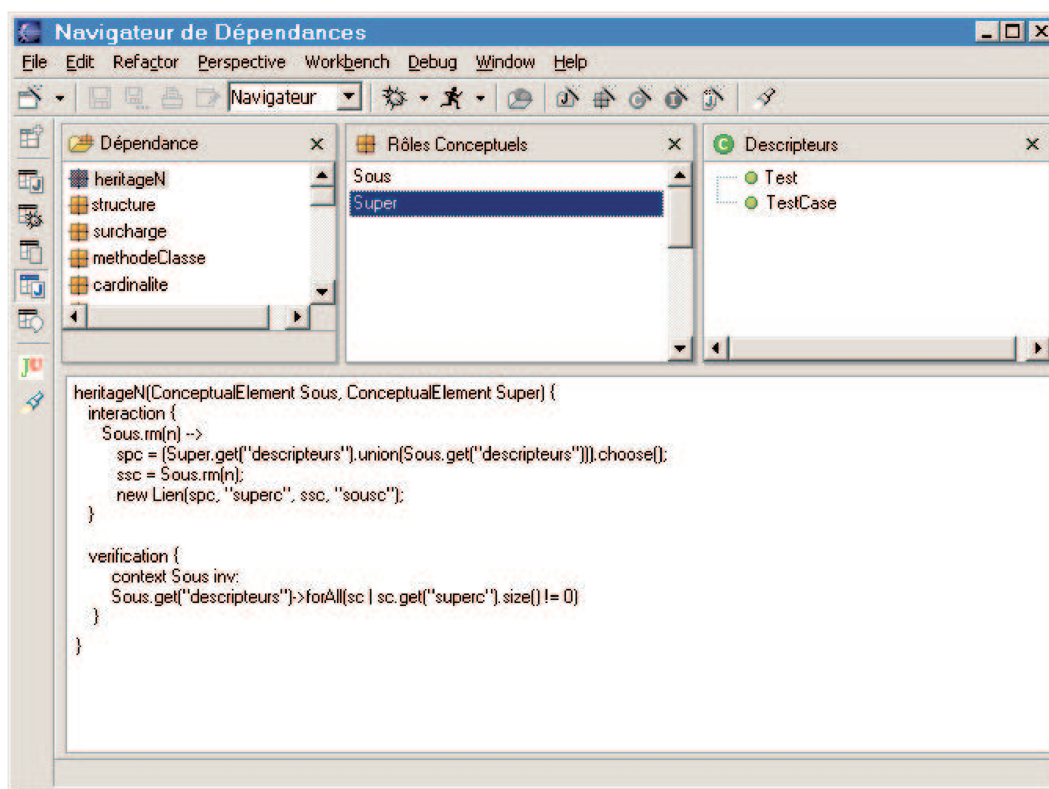


FIG. 4.26 – Navigateur de dépendances.

Bibliothèque de dépendances. La bibliothèque de dépendances étant un constituant important dans la réduction du temps passé à développer un méta-framework, il paraît important de fournir des outils facilitant sa manipulation. Cet outil, dont l'interface est présentée figure 4.26, aura pour principales fonctionnalités la création de nouvelles dépendances, la pose de dépendances, ainsi que la recherche de dépendances.

4.5.2 Implémentation du modèle

Deux implémentations en Java du modèle d'expression des dépendances structurelles existent. La première n'implémente pas directement le modèle présenté ici, mais une sorte de spécialisation dédiée à la manipulation des schémas de conception dans les langages à base de classes [RF00]. Dans ce modèle nous n'avions pas encore mis en place de langage d'expression des dépendances. Ainsi l'équivalent de nos éléments conceptuels étaient connectés par un système Observateur / Observé, le comportement de chaque dépendance était codé en dur et l'ajout de nouvelles dépendances nécessitait de modifier le code de l'application.

Due aux évolutions du modèle et aux problèmes rencontrés lors de la première implémentation, la nouvelle implémentation est fondamentalement différente et suit le modèle au détail près de la gestion de la réalisation et des règles de création. En effet, au lieu d'utiliser un classique modèle Observateur / Observé nous avons choisi d'utiliser Noah [Pro02], une implémentation du langage ISL.

Cette implémentation, dont le choix a été naturellement guidé par l'utilisation que nous faisions du langage de description, s'est révélée très avantageuse. Elle nous a permis de nous libérer complètement de l'interprétation des règles de création (cela est fourni par Noah qui interprète de l'ISL), mais aussi de simplifier le comportement de la méthode de réalisation. En effet, étant donné que Noah utilise une technique de capture d'envoi de message pour exécuter une interaction, il n'est alors plus nécessaire que les éléments conceptuels vérifient lors de leur réalisation s'ils sont connectés à une dépendance. Grâce à ISL, chacun des éléments conceptuels du méta-framework peut être considéré comme un méta-objet (au sens de Maes [Mae87]) qui contrôle l'exécution d'une opération qui est la réalisation.

Afin d'implémenter la vérification, qui n'est pas encore mise en place, nous pensons utiliser un interpréteur OCL, tout comme nous l'avions fait dans la première implémentation.

4.6 Conclusion sur l'utilisation de notre modèle d'expression des dépendances structurelles

Ce chapitre a montré quelques-unes des dépendances les plus courantes, ainsi que l'utilisation des dépendances pour représenter les schémas de conception.

À partir de ces dépendances organisées dans une bibliothèque, nous avons décrit comment un développeur de frameworks pouvait tirer bénéfice de celles-ci afin de représenter la structure et les dépendances d'un framework et ainsi le documenter. Nous avons aussi vu comment, grâce à l'opération de réalisation, il pouvait engendrer son framework.

De plus nous avons fait l'esquisse d'un environnement de développement orienté framework et avons présenté les grandes lignes de l'implémentation du modèle.

4.7 Conclusion sur le modèle des dépendances structurelles et travaux futurs

Nous avons présenté un modèle de dépendances structurelles flexible permettant l'assistance active ou passive d'un utilisateur. Afin d'en améliorer l'utilisabilité, les dépendances sont abstraites dans des entités réutilisables capitalisées dans une bibliothèque. Ainsi, une dépendance constitue une entité de premier ordre, permettant au développeur de documenter son framework tout en recevant une assistance lors de la création de celui-ci. En effet, une dépendance est manipulée explicitement par le développeur au même titre que des classes, des méthodes... et il peut l'utiliser pour générer son framework.

La flexibilité de ce modèle est acquise en découplant la représentation du framework de son implémentation, en exprimant les dépendances dans une extension d'ISL et une adaptation d'OCL, et en ne requérant pas l'expression complète du framework. Elle résulte aussi de l'absence de couplage entre notre modèle et un langage d'implémentation.

Les travaux futurs sont organisés selon trois axes qui sont : l'extension du modèle, la création d'outils, et le génie logiciel.

4.7.1 Extension du modèle

En ce qui concerne le modèle, trois pistes complémentaires peuvent être envisagées :

1. utiliser dans notre modèle toute la flexibilité offerte par le modèle des interactions (ISL). En effet, dans ISL les interactions peuvent être posées sur des interactions et aussi être ajoutées et supprimées à volonté. Dans notre modèle cela permettrait alors de modifier dynamiquement l'organisation des dépendances alors même que l'utilisateur développe son application. Cependant la possibilité de cette reconfiguration dynamique devra être exprimée par le développeur ;
2. réutiliser des modèles. Dans notre modèle nous ne savons réutiliser que des dépendances. Cependant nous pensons qu'afin d'aller encore plus vite dans la construction de la méta-description d'un framework, il peut être intéressant de réutiliser des modèles complets (éléments conceptuels et dépendances) ;
3. regarder la pertinence de l'utilisation de dépendances au niveau F0. Dans le contexte actuel cela permettrait de donner plus de responsabilité au niveau F0, comme par exemple le contrôle de la cardinalité. Cependant le contexte dans lequel ces dépendances semblent les plus intéressantes est celui de la définition d'une nouvelle opération de réalisation contrôlant la création des instances d'un framework. En effet, tout comme nous contrôlons la création de la description du framework, il est envisageable de contrôler la création des instances des classes du framework afin que certaines contraintes comme le nombre d'instances associées entre elles soit limité en fonction par exemple des informations fournies dans le diagramme de classes du framework.

4.7.2 Création d'outils

Le second axe concerne l'utilisation du modèle et de son implémentation afin de créer divers types d'outils comme des *Active Cookbooks* ou des outils à la FRED. Ici nous avons effectué l'implémentation de notre modèle en Java, cependant il pourrait être intéressant d'étudier l'implémentation de ce modèle dans d'autres langages comme Prolog, ou l'utilisation d'outils de graphes conceptuels. Dans cette perspective d'outillage, il sera important de choisir un format de sauvegarde pour le méta-framework, mais encore plus, de choisir la manière dont il sera fourni avec le framework. On pourra aussi regarder s'il n'est pas possible de créer un outil évitant à développeur de framework d'avoir à exprimer de manière explicite la méta-description du framework.

Bien que notre modèle soit dédié au framework, nous pensons qu'il peut être utilisé dans d'autres contextes. Ainsi, dans la suite de nos travaux sur l'opérationnalisation

du méta-modèle UML [RR01], nous envisageons d'opérationnaliser les contraintes du méta-modèle d'UML grâce à des dépendances structurelles. Cela est d'autant plus adéquat qu'UML est fourni avec des contraintes OCL qui sont indispensables à l'expression de sa sémantique.

4.7.3 Génie logiciel

Le troisième axe est plus lié à un aspect génie logiciel et à la quantification des avantages que peut procurer notre modèle. Ainsi, bien que notre modèle semble pouvoir exprimer les dépendances d'un framework relativement simplement, il serait intéressant de mesurer avec rigueur :

- le gain de temps procuré à l'utilisateur ;
- le temps consommé par l'expression des dépendances lors du développement ;
- le temps gagné par l'expression des dépendances par rapport au temps de création d'une documentation ;
- le pourcentage des besoins en documentation couverts par la documentation ;
- le nombre de dépendance réutilisées.

De plus une telle étude pourrait être complétée d'une enquête auprès des utilisateurs sur l'adéquation entre leurs besoins et le système d'expression fourni.

L'autre piste que peut nous ouvrir cette modélisation des dépendances est la quantification de la complexité des frameworks. En effet, on peut imaginer un système qui en analysant la complexité des dépendances exprimées en ISL donne une indication sur la complexité d'utilisation d'un framework sous forme d'une métrique [eAPSZ01].

Enfin, une dernière direction qui ne rentre dans aucun des points précédents mais semble aux yeux de l'utilisateur final presque la plus primordiale est la création d'une notation graphique appropriée à la représentation des dépendances.

Deuxième partie

Expression des dépendances comportementales

Chapitre 5

Différentes techniques d'expression des dépendances comportementales

L'objectif de ce chapitre est de répertorier un ensemble de techniques de génie logiciel permettant la description des dépendances comportementales.

Comme l'indiquent Helm et al. dans [HHG90] ces dépendances qui se traduisent par l'utilisation d'une classe par une autre (principalement en terme d'envoi de message entre leurs instances), ne facilitent pas la conception, la compréhension et la réutilisation de logiciels programmés à base de classes. « *While the recent literature recognizes the importance of inter-object behavior (expressed in terms of collaboration graphs, responsibilities, mechanisms, or views, for example), there is surprisingly little language support for its specification and abstraction. This means the existence of behavioral compositions in a system, and in particular the* behavioral dependencies *that they imply, cannot be easily inferred ; they are spread across many class definitions in method implementations. This causes subsequent problems in the design, understanding and reuse of object-oriented software.* » [HHG90]

Dans le cadre de cette thèse, nous représentons ces dépendances de manière externe dans le but de faciliter la réutilisation de frameworks. Ainsi, le comportement externe de l'objet n'est pas décrit par rapport à son état interne, mais simplement par l'ensemble des méthodes reçues et accessibles à un instant [1]. Cet ensemble de méthodes accessibles dépendant d'un ensemble de méthodes reçues est aussi identifié sous le nom de protocole (« *The sequences of requests that an object is capable of servicing constitute the object's protocol* » [Nie95]).

Ainsi, cet état de l'art décrit les techniques connues pour spécifier et abstraire les dépendances comportementales d'une application. Il commence par l'étude des différents diagrammes UML, qui en phase d'analyse et de conception, aident à la capture

[1]Il convient de préciser que l'état interne de l'objet est conséquence directe des stimuli reçus par l'objet, ce qui est comparable à l'équivalence de Nerode : un état représente l'historique de ce qui s'est produit dans le système.

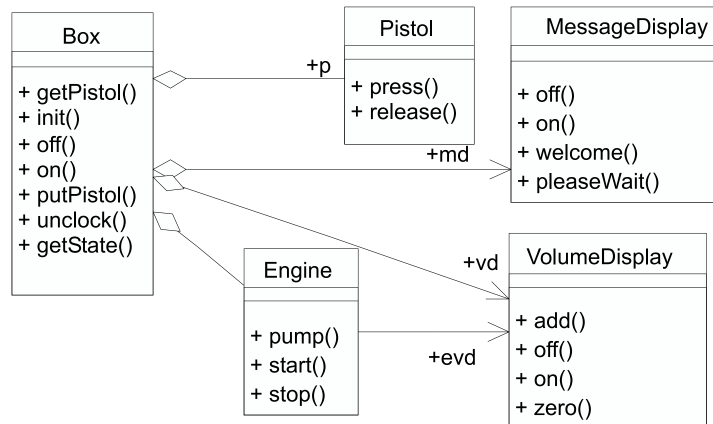


FIG. 5.1 – Diagramme de classes de la pompe à essence.

de ces dépendances, puis présente les langages de description d'architecture (ADL). Avant de présenter une synthèse des différentes techniques évoquées, une présentation de différents contrats sera faite.

5.1 Exemple : la pompe à essence

Afin d'illustrer cet état de l'art et de mieux comparer les méthodes de capture des dépendances nous utilisons l'exemple d'une pompe à essence. La pompe à essence que nous décrivons est l'objet usuel des stations-service. Elle est constituée d'un pistolet, de deux afficheurs (un pour le volume et un pour les messages), d'un moteur et d'une boîte (représentant la boîte en métal fédérant tous ces éléments). Nous considérons chacun de ces éléments comme un composant à part entière. Le diagramme de classes de cette pompe à essence est donné figure 5.1. Alors que les diagrammes d'objets ou de composants auraient été sémantiquement plus adaptés à la représentation de notre pompe à essence, nous avons dû utiliser un diagramme de classes. En effet, c'est le seul diagramme (en UML classique) qui fournisse des informations sur les interfaces d'entrée.

Le fonctionnement de la pompe est le suivant : après avoir été mise en marche (`on()`), la pompe est initialisée (`init()`). Cette phase d'initialisation est nécessaire à la mise en marche des constituants de la pompe. Avant de pouvoir se servir en essence, la pompe doit être débloquée (`unlock()`). Le service se fait en prenant le pistolet (`getPistol()`) puis en appuyant sur la gâchette de celui-ci (`press()`). Le pompage s'effectue jusqu'à ce que la gâchette soit relâchée (`release()`). Pendant cette phase, l'afficheur de volume est notifié (`add()`) du nombre de tours du moteur afin de pouvoir afficher le volume.

Au regard de la description, l'utilisation de la pompe peut paraître simple, (cela est probablement dû à notre familiarité avec l'objet), cependant son utilisation met en jeu plusieurs composants et sa description révèle leurs dépendances.

Nous ne discutons pas dans cette thèse du bien-fondé d'un couplage entre les classes. Ainsi des interactions existent entre les composants d'un système et même si certaines dépendances peuvent être évitées elles ne sont pas l'objet de notre propos. En effet, nous ne prétendons pas corriger les éventuelles erreurs de conception, mais offrons un moyen de représenter les dépendances.

5.2 Critères de comparaison

Les techniques présentées ici sont générales et visent à représenter des dépendances entre des « modules » au sens large (classe, composant, module...). Cela implique que les composants ne soient pas aussi indépendants que des composants pris sur une étagère (*components off the shelves*). En effet, lorsqu'on intègre un composant au sein d'un framework celui-ci doit répondre à des pré-requis sur ses interfaces (nommage et typage des méthodes), mais aussi sur le protocole de ses interfaces. Ainsi, nous regarderons avec attention la manière dont cette coopération et le protocole des objets peuvent être exprimés.

Au fil des techniques présentées nous regarderons plus particulièrement les notations et les concepts qu'elles proposent, en analysant plus particulièrement leur simplicité (critère qui est décisif dans l'adoption d'une technique), leur intégration dans le cycle de vie du logiciel et leur modularité. En effet, il est souvent trop contraignant de devoir utiliser une notation pour la modélisation complète d'un système quand seulement quelques informations sont pertinentes.

La notation étant le vecteur de l'information que nous voulons capturer, nous regarderons les concepts qu'elle supporte (objet, structure de contrôle, gestion des erreurs).

Bien que d'après sa spécification un composant puisse être utilisé dans un framework, il ne va pas obligatoirement fonctionner car son interface et son protocole ne sont pas en adéquation avec ceux attendus [ZW97]. Ainsi, afin de s'assurer du bon fonctionnement d'un système avant son exécution, nous regarderons avec intérêt les techniques qui permettent de faire des preuves afin d'assurer le bon fonctionnement des composants connectés, ce qui requiert donc un modèle formel. Si l'absence de modèle formel ou la modularité empêche la vérification statique, nous regarderons si les informations proposées peuvent permettre une vérification (même partielle) du système à l'exécution, et détecter des erreurs dans l'utilisation du protocole d'un composant.

5.3 Diagrammes UMLs

La notation UML (Unified Modeling Language) [OMG01b] introduite en 1997 offre, grâce à ses neuf diagrammes, un moyen graphique de décrire un système, et cela selon plusieurs points de vue. L'un de ces points de vue est lié au comportement (on parle encore de vues comportementales), et est constitué des diagrammes de collaboration, séquences, d'états-transitions, et d'activités. Bien qu'UML ne repose sur aucun modèle formel, nous étudierons ces diagrammes pour leur facilité d'utilisation et leur intégration dans le cycle de vie du logiciel (par exemple le RUP [JBR99] utilise ces diagrammes).

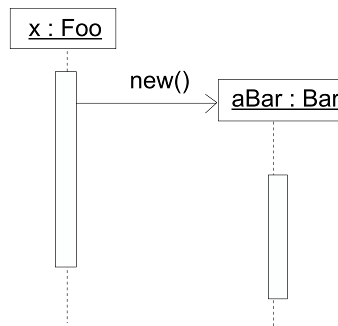


FIG. 5.2 – Instanciation dans un diagramme de séquences.

5.3.1 Diagrammes de séquences

Les diagrammes de séquences (Sequence Diagrams) sont utilisés pour montrer les échanges de messages inter-objets, et cela de manière séquentielle. Ce diagramme décrit un scénario (un exemple d'utilisation) du système en montrant comment s'enchaînent les différents appels de messages. La représentation graphique utilisée présente horizontalement les objets (acteurs) qui interviennent dans la séquence et verticalement le temps (de haut en bas). Sous les objets, une ligne verticale représente leur ligne de vie. Un envoi de message entre deux objets est représenté par une flèche qui part de la ligne de vie de l'émetteur et se termine sur celle du récepteur. L'étiquette portée par la flèche identifie le message émis ainsi que ses paramètres. La condition et les messages asynchrones sont aussi supportés par la notation. Les lignes de vie permettent de représenter l'activation des objets mais n'en indiquent pas pour autant l'instanciation. L'instanciation est généralement représentée en réduisant la ligne de vie de celui-ci de manière à ce que le message d'instanciation soit directement reçu par l'objet (figure 5.2). Le nombre d'instances géré est limité au nombre d'objets représentés. L'ajout dynamique d'acteurs n'est pas possible.

L'utilisation de ce diagramme sert généralement en phase d'analyse à mieux cerner les échanges de messages entre les différents objets d'un cas d'utilisation.

Bien que les interactions entre les composants soient explicites, elles ne sont décrites que dans le cadre d'une utilisation spécifique d'un composant. Ce diagramme n'est donc pas adapté à la description complète de l'ensemble des composants d'un système puisqu'il faudrait alors décrire l'ensemble des scenarii dans lesquels chaque élément est susceptible d'intervenir pour avoir une description complète du protocole. Ainsi, l'exemple de la figure 5.3 décrivant les messages échangés entre les composants `Box`, `VolumeDisplay`, et `MessageDisplay`, ne permet d'avoir qu'une information partielle sur le protocole d'utilisation de `Box`.

Bien que l'absence de sémantique formelle ne permette pas la mise en œuvre de vérification statique, les diagrammes de séquences restent utiles pour les tests et la vérification dynamique [WMB99].

+	: Notation simple, intégration au cycle de vie ;
−	: Ne représente pas l'intégralité du protocole.

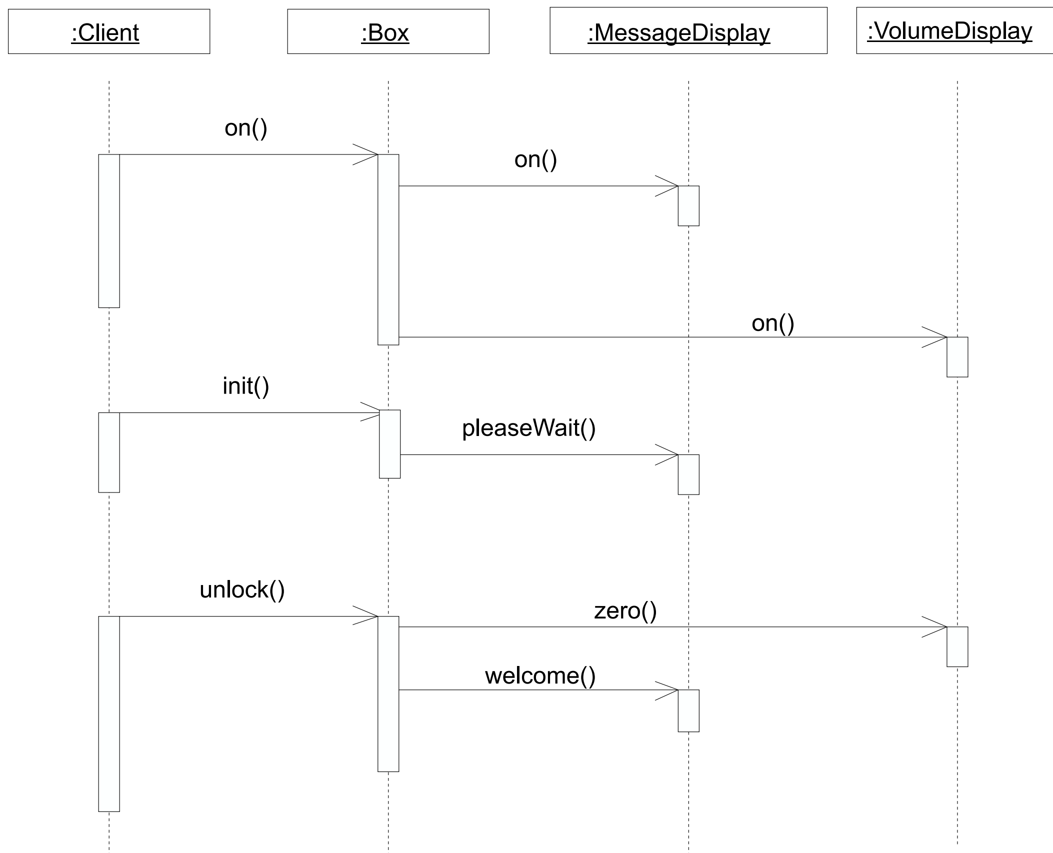


FIG. 5.3 – Diagramme de séquences.

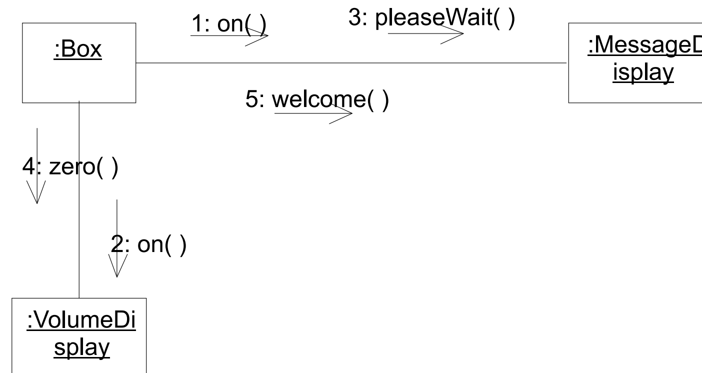


FIG. 5.4 – Diagramme de collaboration.

5.3.2 Diagrammes de collaboration

Les diagrammes de collaboration présentent des informations à la fois sur l'aspect statique et dynamique d'un système. En particulier, ils détaillent pour une configuration d'objets donnée, l'échange des messages entre ces objets.

Ils mettent en évidence le côté structurel de la collaboration plutôt que l'aspect temporel comme le font les diagrammes de séquences. Graphiquement on représente les instances actives du système (des objets, représentés par une boîte), la manière dont elles sont connectées (des liens, traits connectant les boîtes) ainsi que les messages qui transitent sur ces connexions. Puisque la représentation graphique ne permet pas un ordonnancement implicite des messages, ceux-ci sont préfixés d'un numéro indiquant l'ordre d'exécution.

Tout comme les diagrammes de séquences, les diagrammes de collaboration décrivent un scénario d'échange de messages entre des objets dont la configuration et le nombre est fixé. Ces diagrammes ne nous conviennent donc pas pour représenter entièrement le protocole d'un composant.

+	: Notation simple ;
−	: Ne représente pas l'intégralité du protocole.

Les diagrammes de collaboration et de séquences sont des diagrammes d'interactions dont Martin Fowler [Fow97] parle en ces termes : « *One of the principal features of either form of interaction diagram is its simplicity... if you try to represent something other than a single sequential process without much conditional or looping behavior, the technique begins to break down* ».

5.3.3 Diagrammes d'états-transitions

Les diagrammes d'états-transitions, aussi connus sous le terme de Statecharts[2], se focalisent sur le fonctionnement interne d'une classe en réponse à un stimulus. Ils

[2]Les diagrammes utilisés dans UML sont inspirés de la notation définie par Harel [Har87].

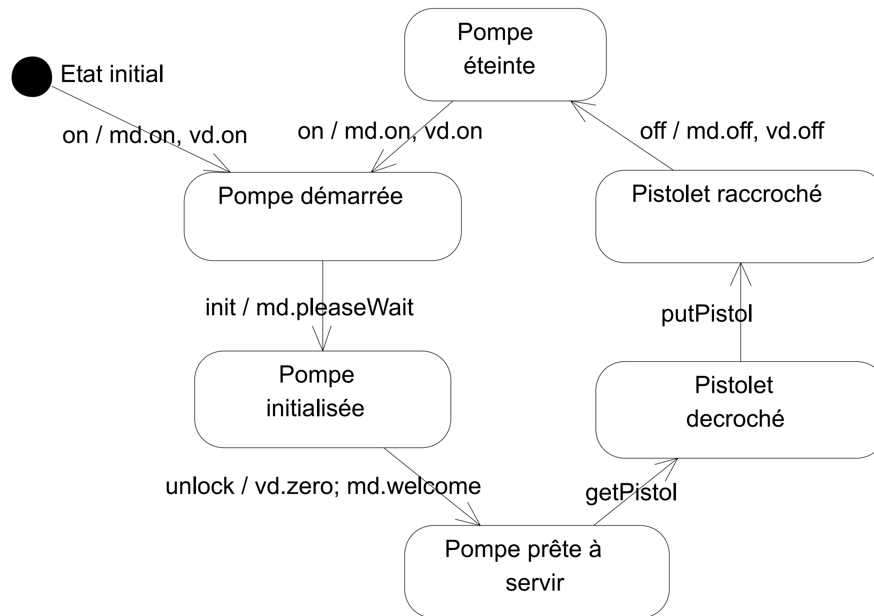


FIG. 5.5 – Diagramme d'états-transitions. Le nom des receveurs correspond au nom des relations dans les diagrammes de classes.

montrent comment une instance passe d'un état à un autre à la réception d'un signal (appel de méthode, exception, événement...) et détaillent les conditions qui font qu'un composant aura un comportement plutôt qu'un autre. Ce fonctionnement interne peut être plus ou moins détaillé, allant d'une représentation de tous les états concrets du composant à une description du comportement beaucoup plus abstraite comme le montre la figure 5.5.

Les Statecharts doivent notamment leur succès à la manière dont ils spécifient l'aspect dynamique du comportement des objets. Ils étendent les automates en permettant l'expression d'une hiérarchie d'états et celle du parallélisme. Dans les Statecharts, les états sont généralement nommés. Les transitions, représentées par des flèches, ont une étiquette qui permet d'indiquer dans quelles conditions une transition peut être effectuée ainsi que les conséquences de cette transition (les messages émis).

Grâce aux Statecharts, la communauté a apporté plus d'attention à la vérification de modèles (*model checking*). Cet effort résulte de l'existence de réels outils pour faire la vérification formelle de StateCharts. Cependant devant la recrudescence des sémantiques associées aux StateCharts, UML n'a su en tirer profit ne voulant ou pouvant pas en favoriser une en particulier. Il aurait été d'autant plus dur pour UML de sélectionner une sémantique, que les StateCharts proposés ne sont pas strictement hiérarchisés (des transitions peuvent traverser les frontières d'un état) empêchant la compositionnalité et donc la simplification des preuves.

Ainsi, les récents travaux qui visent à donner une sémantique aux StateCharts d'UML nécessitent d'abord la sélection d'un ensemble de constructions graphiques puis

la précision de leur sémantique afin de les traduire dans le formalisme d'un vérificateur de modèles (*model checker*) [LP99, Kwo00][3]. Cette formalisation permet alors de faire des tests d'interblocages (deadlocks), d'accessibilité d'états ou d'états invalides, de vérifier que des messages ne sont pas envoyés à des objets qui ne sont plus accessibles. Il est à noter qu'aucune de ces extensions sémantiques n'a encore été intégrée à UML (Les versions d'UML disponibles à ce jour sont la 1.3 et la 1.4).

Bien que les diagrammes d'états-transitions aient initialement pour objectif la modélisation du comportement interne d'une classe, ils sont aussi utilisés sous le nom de *protocol state machine* [OMG01b, p. 2-175] pour représenter le protocole des objets. Cela permet alors de bénéficier de tous les avantages offerts par les avancées faites en matière de vérification de modèles.

Les diagrammes d'états-transitions ont retenu notre attention car ils sont simples à comprendre et permettent la représentation complète du protocole du composant. De plus, vUML [LP99] les utilise couplé à PROMELA (un langage d'entrée pour SPIN, un outil formel de vérification de systèmes distribués) afin de vérifier les interactions entre objets[4].

Cependant nous n'avons pas retenu cette notation car les étiquettes des transitions ne permettent pas de représenter, sur les messages émis, un comportement complexe avec des itérations et conditions.

+ : Capture complètement le protocole d'entrée, modèle formel ;
− : Représentation du protocole de sortie.

5.3.4 Diagrammes d'activités

Les diagrammes d'activités sont une variante des diagrammes de flots (organigrammes, réseaux de Pétri). Ils s'organisent autour des actions et servent principalement à représenter le comportement d'un système ou d'une méthode. Bien que ces diagrammes permettent de représenter le comportement de plusieurs éléments en parallèle, la connexion entre ceux-ci n'est pas aisée [GK98]. Ces diagrammes ont l'avantage de pouvoir décrire un squelette de l'algorithme, cependant pour aller plus loin il serait nécessaire d'utiliser l'*action semantics* [OMG].

Les autres diagrammes UML, ne comportant pas d'aspects comportementaux, ne feront pas l'objet d'une description dans cette section. Pour plus de détails sur ceux-ci ou sur la notation des diagrammes présentés précédemment, on peut consulter le livre d'introduction à UML de Pierre-Alain Muller [Mul97], celui des Pères d'UML [BRJ99] ou encore de Sinan Si Alhir [Alh98] détaillant la notation graphique.

[3]De nombreuses autres références sur de tels travaux peuvent être trouvées dans la section Statecharts des actes de la conférence UML.

[4]La description complète du système (la connexion des instances) est faite à l'aide de diagramme de collaboration.

5.4 Langage de description d'architectures (ADL)

5.4.1 Généralités

Ces langages ont été introduits pour donner une sémantique aux diagrammes qu'établissaient les concepteurs de logiciels pour décrire la connexion entre les modules d'une application. Ils sont les successeurs des Langages d'Interconnection de Modules (MILs) [DK76, PDN86]. Ces MILs ont pour objectif de décrire l'architecture d'un programme, et la manière dont les différents modules qui le composent interfèrent. Cependant, bien que donnant des informations sur l'interaction des modules (quelle variable ou fonction un module fournit, requiert...), les MILs ne donnaient jamais d'information sur les appels de fonctions et comment ceux-ci s'enchaînaient. C'est pour cela qu'ont été mis au point les ADLs (*Architectural Description Language*). Les ADLs ont aussi fait la promotion des concepts de styles et patterns architecturaux dont le but est de décrire l'organisation d'un système, et ce faisant se structurer les dépendances. Par exemple dans une architecture en couche, les dépendances sont limitées entre les constituant d'une couche et ceux de la couche inférieure. Cependant nous ne donnerons pas d'avantage d'information concernant ces styles et patterns qui ne suppriment pas les dépendances mais les structurent.

Un ADL prend pour hypothèse que les composants qui vont constituer les briques de base de l'architecture existent déjà de manière indépendante (ils peuvent fonctionner seuls et fournissent une fonctionnalité, comme peuvent le faire les commandes Unix). Ainsi l'ADL va mettre en relation ces composants en décrivant leurs connexions (la relation peut être aussi simple qu'un *pipe* Unix).

Si un système est spécifié dans un ADL, on peut selon les outils associés à cet ADL et les fondements formels sous-jacents, l'analyser, faire des preuves, réaliser des simulations, etc. L'un des buts des ADLs est de pouvoir répondre à la question « Est ce que deux composants peuvent communiquer de manière sûre ? ».

Un grand nombre d'ADL existent[5] : Wrigth [AG97], Rapide [LKA+95], SADL [MQR95], Unicon, ACME [GMW97]... Malgré les différences de formalisme, de but, de domaine et de support pour les outils, ils ont en commun les concepts utilisés pour décrire les systèmes. Ces concepts sont les suivants :

- Composant : un composant est une entité représentant un élément actif. Il peut avoir plusieurs interfaces qui peuvent toutes être des points de connexion ;
- Connecteur : un connecteur décrit la connexion entre des composants et en contient la sémantique. Comme un composant, un connecteur est réutilisable (ex : un protocole de communication). Il décrit les interactions en terme d'envoi de message entre les composants qui sont mis en relation. Un connecteur peut mettre en relation plusieurs composants ;
- Système : un système décrit un ensemble de composants mis en relation par des connecteurs ;
- Propriété : une propriété est une donnée qui vient étendre la description des composants ou des systèmes. Par exemple, on y retrouve des informations comme le temps de réponse d'un composant.

[5] Voir http ://www.sei.cmu.edu/architecture/adl.html

Les ADLs sont généralement graphiquement représentés par des boîtes (représentant les composants) et des traits (représentant les connecteurs). Cependant afin d'accorder plus de sémantique à ces boîtes, on a recours à des descriptions textuelles dans un langage proche d'un formalisme existant. Quelle que soit la notation choisie, il s'avère qu'elle a bien peu de succès (car peu maléable et connue) face à une notation graphique comme UML qui peut faire office d'ADL. Ce dernier point sur l'adéquation d'UML en tant qu'ADL ne sera pas discuté ici. En effet, bien que Garlan dans [GK00, Gar01] décrive les avantages et inconvénients, et les cas où l'on pourrait utiliser UML comme un ADL, cette question semble relever d'un débat sur la nature des ADLs et d'UML, ainsi que leur faculté à être étendus.

5.4.2 *Wright*

Afin de mieux voir ce qu'est un ADL nous détaillons ici *Wright* [AG97]. Nous avons retenu celui-ci car il n'est pas spécifique à un domaine et les connecteurs sont décrits dans un formalisme proche de CSP (*Communicating Sequential Processes*) [Hoa78]. CSP, grâce à sa notion d'équivalence comportementale, permet de vérifier l'adéquation d'un composant et d'un connecteur. La notion de compatibilité entre un composant et un connecteur fait une utilisation directe des propriétés de CSP et est associée à la notion de raffinement.

Dans *Wright* un composant est constitué de ports (équivalent à une interface dans le modèle objet). Un port décrit une partie du protocole du composant.

Les connecteurs décrivent la manière dont les composants sont connectés. Un connecteur est constitué de deux sections : une section `rôle` indiquant le protocole que doit respecter le composant qui voudra se connecter à ce rôle (c'est le pendant du port du composant), une section `glue` coordonnant le comportement des rôles en indiquant les interactions entre les rôles.

Le système correspond à la définition que nous en avons faite précédemment (sous-section 23). La configuration de celui-ci est décrite à l'aide d'une section `Instances` et d'une section `Attachements`. La première définit, pour les composants et les connecteurs, un ensemble d'instances nommées et typées, et la seconde connecte ces instances. La section `attachement` met en relation les ports de chaque instance des composants avec les rôles des instances des connecteurs. Il est à noter que *Wright* est un ADL statique par opposition aux ADLs dynamiques comme Rapide [LKA+95] qui permettent de modifier dynamiquement la connexion entre composants.

La représentation de la pompe à essence avec *Wright* (figure 5.6) engendre un système constitué d'autant de composants ADL qu'il y a de composants dans notre système, ainsi qu'autant de connecteurs qu'il y a de relations entre nos composants.

Cette prolifération de connecteurs déjà constatée dans [AB01] n'est pas élégante, et va à l'encontre des concepts des ADLs qui prônent la réutilisation du connecteur.

En conclusion, bien que la représentation d'un framework avec un ADL soit possible [SG99, AB01], nous rejoignons l'avis de [AB01] qu'un ADL n'est pas la solution la plus adaptée à la représentation détaillée d'un framework. Nous pensons que cette inadéquation est due à la nature opposée des frameworks et des ADLs. En effet, les composants des ADLs n'ont pas de contexte et requièrent des connecteurs pour établir des liens avec d'autres composants alors que dans un framework, un composant

```
Configuration PetrolPump
   component Box
   component VolumeDisplay
   component MessageDisplay
   component Engine
   component Pistol

   connector Box-VolumeDisplay
   connector Box-MessageDisplay
   connector Box-Pistol
   connector Engine-VolumeDisplay

   instances
    b : Box
    vd : VolumeDisplay
    e : Engine
    p : Pistol
    md : MessageDisplay
    c1 : Box-VolumeDisplay
   attachements
   ...
end PetrolPump
```

FIG. 5.6 – Représentation partielle de la pompe à essence avec l'ADL *Wright*.

connaît les autres composants avec lesquels il se connecte. La flexibilité offerte par un connecteur séparé n'est donc pas nécessaire pour un framework et en rendrait plus compliquée la compréhension puisqu'il faudrait à la fois étudier composants et connecteurs. De plus, une telle flexibilité signifierait que l'on peut changer le comportement d'un composant, ce qui n'est pas dans l'esprit des frameworks.

+ : Modèle formel de CSP ; **−** : Séparation composant / connecteur inadaptée aux frameworks.

5.4.3 *Chemical Abstract Machines*

Comme les ADLs, les travaux de Inverardi et al. [IWY00] abordent le problème de description d'architectures. Les composants sont décrits de manière complètement externe par des *Chemical Abstract Machines* (CHAM) [BB90]. Pour chaque composant, cette description contient son protocole, ainsi que le protocole des autres composants. Contrairement aux ADLs, ce modèle de description n'utilise pas de connecteurs puisque la description de la connexion est donnée par le composant.

Une CHAM est une machine abstraite basée sur le modèle de la réaction chimique : un ensemble de molécules (des termes algébriques) forment une solution (un état). Cette solution réagit selon des règles fixées et donne une autre solution (c'est la transition). Il y a bien sûr une solution initiale et une « solution finale ».

Selon les auteurs, le bénéfice d'une telle abstraction est la simplification de la description des composants et de ses interconnexions. Ainsi après une phase d'instanciation puis de dérivation, on peut vérifier l'adéquation de composants, ainsi que l'absence d'inter-blocages. L'instanciation consiste à mettre en correspondance les CHAMs représentant les composants qui doivent interagir (c'est une phase équivalente à la section système dans les ADLs 23). La dérivation consiste à traduire les CHAMs résultant de l'instanciation en graphes décrivant la manière dont fonctionne un composant.

Un premier graphe décrit le protocole du composant (on parle aussi de comportement réel « *Actual behavior* »), et un second représente les hypothèses que fait le composant sur le protocole des composants avec lesquels il doit interagir (comportement supposé : « *Assumed behavior* »). Ces deux graphes permettent de vérifier, en utilisant un algorithme de bi-simulation étendu la composition des composants et aussi l'absence d'inter-blocages.

Bien que ces travaux permettent la vérification statique du système, la notation des CHAMs reste trop compliquée et rend son utilisation délicate. De plus, la modularité n'est pas possible et l'intégralité du système (ou une sous partie cohérente) doit être représentée afin de pouvoir prouver une quelconque propriété.

+ : Modèle formel, séparation protocole fourni / protocole requis ; **−** : Modularité.

5.4.4 Approches connexes

Les ADLs ont inspiré beaucoup de travaux. Ainsi, on peut noter les travaux sur les Gluons [Pin95] et plus récemment l'outil PacoWire [WV01]. Le concept de gluon

a été créé lors de la réalisation d'un framework financier pour lequel les concepteurs cherchaient à gagner en flexibilité pour connecter les divers composants. Ainsi un gluon abstrait sous forme d'automate, la communication entre deux composants, et en fait donc une entité réutilisable. Il est à noter que les automates ne sont pas utilisés à des fins de vérification.

Les travaux de Wydaeghe et al. (l'outil PacoWire) proposent une technique de documentation des composants et des connexions (renommées pour l'occasion *« Composition Patterns »*) à base de Message Sequence Charts (MSC) [IT94]. Tout comme les gluons, ils cherchent à faciliter la réutilisation de connexions. Afin de vérifier l'adéquation de composants deux à deux ou d'un ensemble de composants, ces MSCs sont traduits en automates. Si deux composants ne sont pas adéquats, un composant d'adaptation peut être généré.

Le concept d'« interactions » [Ber01] développé au sein de l'équipe RAINBOW et utilisé comme base de la mise en œuvre de notre modèle de dépendances structurelles, offre une solution similaire aux gluons et à PacoWire. Une interaction, concept qui est initialement prévu pour ajouter dynamiquement des relations entre des instances (cette mise en relation se fait par envoi de messages), peut être utilisée pour abstraire la connexion entre des composants. Une interaction est spécifiée pour un ensemble de participants pour lesquels certaines méthodes sont redéfinies. La description d'une interaction est faite en ISL, langage ad hoc offrant les structures de contrôle minimales. Nous n'avons pas retenu cette approche car une interaction ne s'intéresse pas à la communication initialement fournie par les objets qu'elle connecte et ne permet pas de capturer le protocole d'entrée d'un objet.

5.5 Approches formelles

En plus des ADLs et des CHAMs, de nombreuses approches ont utilisé des formalismes très divers pour décrire le comportement des composants. Nous parcourons entre autre les travaux de Nierstrasz sur les *regular types*, ceux de Yellin et de Rumpe.

5.5.1 Expressions de chemins

L'utilisation de séquences d'actions associées aux composants afin de décrire le comportement autorisé a été initialement introduit par les expressions de chemins dans [CH74]. Ce travail exprime le comportement à l'aide d'une expression régulière où les éléments atomiques représentent une opération du composant.

Nous n'avons pas choisi cette représentation et ce malgré les possibilités de preuves qui existent, car la représentation d'un comportement complexe devient très rapidement illisible. De plus le comportement en erreur n'est pas séparé du comportement nominal.

+ : Modèle des automates ;
− : Notation.

5.5.2 Rumpe

Les travaux de Rumpe [RK96] utilisent des automates pour décrire une abstraction du comportement interne du composant (*state-box-view*) : les états de ces automates sont décrits à partir des variables de la classe, et les transitions sont étendues par des pre- et post-conditions.

L'apport principal de ce travail est la mise en place de règles de raffinement syntaxique, basées sur une étude de la sémantique dénotationnelle et opérationnelle des automates utilisés. Lorsque ces règles sont respectées, elles permettent de créer des sous-classes en accord avec les notions de sous-types comportementaux.

Ce dernier point est particulièrement intéressant lorsqu'on cherche à remplacer un composant. Cependant les travaux se focalisent sur des automates qui ne prennent en compte, ni la hiérarchie, ni la concurrence. On peut penser que l'ajout de tels mécanismes entraînerait probablement des modifications dans les règles de raffinement. De plus on peut noter que les valeurs des variables d'instances sont utilisées dans le choix des transitions.

+ : Propriété de substituabilité ;
− : Accès à la valeur interne de l'objet.

5.5.3 *Regular types*

Les *regular types* [Nie95, Nie93] visent à modéliser le comportement d'un objet par un processus (algèbres de processus [Mil80]) : « Les processus modélisent naturellement un objet puisqu'ils représentent du comportement pur (i.e. envoi de message) » [Nie95].

Cette définition comportementale d'un type vient étendre la notion de type en représentant son protocole et se focalisant sur l'ordre des messages qu'il peut accepter. Cet ordre est défini en utilisant CCS [Mil89]. Cette représentation formelle permet alors la spécification de deux relations : *request substitutability* et *request satisfiability*. La première vérifie qu'un type peut être remplacé par un second (vérification de *trace containment*) et la seconde vérifie l'adéquation entre un objet et son client (elle vérifie qu'un client qui envoie des messages à un composant ne rencontre pas de problème). Ces deux relations sont expliquées et démontrées.

On peut regretter qu'aucune notation graphique ne soit associée à ce formalisme. Mais bien plus encore, l'absence de la représentation des messages émis par un objet empêche d'utiliser cette notation pour la représentation des dépendances comportementales dans un framework.

+ : CCS ;
− : Absence des messages émis, notation.

5.5.4 Yellin

Par sa définition des collaborations, Yellin [YS94] cherche à décrire l'interaction entre deux composants et à montrer leur compatibilité. Lorsque deux composants fonctionnellement compatibles ont des problèmes d'incompatibilité d'interfaces, un adaptateur est généré semi-automatiquement.

Une collaboration consiste à décrire dans une première section l'interface qui est partagée entre les composants (c'est-à-dire les messages qui peuvent être échangés entre les composants), et dans une seconde les contraintes existant entre ces méthodes par une grammaire d'états finis. Cette grammaire est composée d'un ensemble d'états nommés et de transitions. Les états sont abstraits et ne correspondent pas aux états internes de l'objet ; l'encapsulation est conservée. Une transition exprime le passage d'un état à un autre sous la forme suivante : `<state> : <direction> <message> -> <state>` où la direction indique si le message est reçu ou émis (« + » ou « - »). Ainsi l'exemple de la figure 5.7 décrit dans une première collaboration la relation entre le composant `Box` de la pompe à essence et le `MessageDisplay`, et dans une deuxième collaboration le protocole de l'afficheur.

La grammaire utilisée dans la description est ensuite transformée en un automate d'états finis afin de vérifier l'adéquation entre des protocoles. La vérification effectuée ne garantit que l'absence d'interblocage entre deux composants.

+ : Modèle des automates ;
− : Lisibilité.

5.6 Contrats

5.6.1 Contrats de Helm et al.

Les contrats [HHG90] proposent un formalisme textuel pour faciliter la compréhension des interactions entre objets, mais pour aussi contraindre la structure d'une classe, le tout dans un objectif d'aide à l'utilisation de framework. L'idée sous-jacente à l'utilisation des contrats est d'effectuer une description complète de l'application en les utilisant, puis de faire l'implémentation en s'y conformant (les auteurs décrivent l'opération de *contract conformance*).

Ainsi, un contrat contient la liste des participants qui sont mis en œuvre et décrit pour chaque méthode les appels que chacune d'elles doit effectuer (le comportement interne des méthodes est montré : affectations, conditions...). Le contrat fixe aussi des invariants entre les participants et contient des informations sur le type des variables de ceux-ci. Il y a donc deux catégories d'obligations : une obligation de type et une obligation causale.

Un contrat supporte deux opérations qui ont pour but de faciliter l'expression par réutilisation : le raffinement et l'inclusion. Le raffinement consiste en l'extension d'un contrat (on peut spécialiser le type d'un participant, étendre ses actions...). L'inclusion permet de réutiliser un contrat existant en l'intégrant à un nouveau.

La connexion entre les participants d'un contrat est faite par instanciation. Cette instanciation vise à décrire le nombre d'instances de chaque composant et leurs connexions. Cela est similaire à ce que l'on trouve dans les ADLs.

Ces contrats sont intéressants, cependant ils montrent les détails d'implémentation de chaque méthode, et ne spécifient pas de protocole pour les méthodes en entrée. Certes celui-ci pourrait être retrouvé si le système était entièrement décrit, mais cela serait fastidieux. De plus, afin d'offrir la modularité nous supposons que la description d'un composant dans un contrat est partielle. Dans le cas contraire, un framework

```
Collaboration Boîte{
 Receive Messages{
    on();
    init();
    unlock();
    off();
  }
  Send Messages{
    on();
    pleaseWait();
    welcome();
    off();
  };
  protocol {
    States{ A (init), B, C, D, E, F, G, H, I };
    A : -on -> B;
    B : +on -> C;
    C : -init ->D;
    D : +pleaseWait -> E;
    E : -unlock -> F;
    E : -off -> G;
    F : ...
    G : +off -> A;
  };
}
Collaboration Affichage{
  Receive Messages{
    on();
    pleaseWait();
    welcome();
    off();
  }
  Send Messages{ }
  Protocol {
    States{ A (init), B };
    A : -on -> B;
    B : -off -> A;
    B : -pleaseWait() -> B;
    B : -welcome() -> B;
  }
}
```

FIG. 5.7 – Utilisation des collaborations à la Yellin pour la représentation de la relation entre la boîte de la pompe à essence et l'afficheur de message.

```
contract BoxMessageDisplay
   Box supports [
    md : MessageDisplay
    on() |-> md->on()
    init() |-> md->pleaseWait()
    unlock() |-> md->welcome()
    off() |-> md.off()
   ]
   MessageDisplay supports [
    on()
    pleaseWait()
    welcome()
   ]
   invariant
end contract

contract BoxVolumeDisplay
   Box supports [
    vd : VolumeDisplay
    on() |-> vd->on()
    unlock() |-> vd->zero()
    off() |-> vd.off()
   ]
   VolumeDisplay supports [
    on()
    off()
    zero()
   ]
   invariant
end contract
```

FIG. 5.8 – Contrats entre la boîte de la pompe à essence et l'afficheur de volume, et entre la boîte et l'afficheur de message.

serait alors représenté par un unique contrat. Cette séparation nous amène à nous poser la question de la composition de contrats. En effet, si l'on regarde la description partielle de la pompe à essence fournie figure 5.8, on ne voit pas comment le code des méthodes `on`, `off` et `unlock` doit être entrelacé. Peut-être faut-il alors regrouper les interactions en un seul contrat ce qui impliquerait, si l'on applique cela de manière récursive, la description de la pompe à essence dans un seul contrat.

+ : Description du comportement en sortie ; − : Absence du protocole d'entrée, modularité, détail d'implémentation.

5.6.2 *Describing and Using Object Frameworks*

Afin d'améliorer l'assemblage et l'extensibilité de frameworks, les travaux de Mili et Sahraoui [MS99] sur « la description et l'utilisation de frameworks » proposent de décrire la structure d'un framework à la manière d'un contrat de Helm.

Contrairement aux contrats de Helm, ici la description du comportement est guidée par les fonctionnalités : pour chaque fonctionnalité du framework un graphe qui représente la fermeture des appels intervenant dans cette fonctionnalité est calculé. Ce graphe supporte des opérations d'assemblage (avec d'autres graphes) et de substituabilité qui servent de base pour tester la validité de l'assemblage de frameworks ou la validité d'une extension.

Ces travaux introduisent aussi une phase de *packaging*. Cette phase consiste à encapsuler dans une seule fonction tous les appels qui sont identifiés par le graphe d'une fonctionnalité, et cela de manière à ce que l'utilisateur n'ait pas à se soucier des dépendances comportementales et structurelles entre les différents objets mis en œuvre dans la fonctionnalité.

5.6.3 Contrats d'Eiffel

Les contrats dont on parle dans le contexte d'Eiffel [Mey97] sont différents de ceux présentés ci-dessus : ce sont des contrats d'interface. Ils reposent sur un ensemble d'assertions qui définissent pre- et post- conditions et invariants. Ainsi le contrat qui est passé entre une classe et son client est défini par ces assertions qui viennent documenter les méthodes. Ces pre- et post- conditions sont vérifiées à l'exécution. Les contrats passés avec les sous-classes sont principalement basés sur les invariants de classes. Un tel exemple d'utilisation des contrats est présenté dans [Sou99] pour documenter les frameworks.

Pour être mis en œuvre, ces contrats nécessitent une connaissance au moins partielle de l'état interne de l'objet. Bien que par l'ajout de variables ad hoc il soit possible d'utiliser ces contrats pour vérifier le protocole d'utilisation d'un composant, ce n'est pas leur utilisation principale.

5.7 Schémas de conception

Dans ce contexte de dépendances comportementales, les schémas de conception [GHJV95] ne sont pas adéquats. En effet, malgré l'aspect comportemental qu'ils four-

nissent, ils ne correspondent pas aux besoins que nous avons en matière d'expression de dépendances comportementales. En effet, un schéma de conception décrit des familles d'interactions entre des classes (ainsi que quelques aspects de sa structure), mais de par sa nature générique et son but il ne doit pas décrire le comportement d'un système particulier (par exemple, l'interaction entre la pompe à essence et le moteur).

5.8 Récapitulatif

Avant de présenter l'orientation générale de notre proposition d'expression des dépendances comportementales, nous synthétisons les différentes techniques présentées précédemment dans le tableau 5.1.

Les critères de comparaison utilisés ont été présentés section 5.2. On peut toutefois noter que le critère lié au protocole a été divisé en un critère protocole d'entrée* et un critère protocole de sortie*. Le premier se focalise sur la représentation du protocole des messages entrants, alors que le second se focalise sur la représentation des messages sortants (appelés). L'aspect vérification dynamique n'a pas été inclus au tableau car même si certaines techniques n'ont pas d'outil pour la vérification, il semble qu'il puisse être réalisés.

La solution que nous cherchons à atteindre est un mélange de notation graphique similaire aux diagrammes d'états-transitions d'UML, permettant de spécifier le protocole d'entrée des objets (comme les *regular types*) et leur protocole de sortie (comme dans les contrats) à un haut niveau d'abstraction proche de celui de Yellin, tout en gardant la possibilité d'effectuer des vérifications (ADLs).

Avant de décrire cette solution nous présentons au chapitre suivant (chapitre 6) le modèle synchrone qui constituera la base de notre modèle.

« *The best practionners understand that the most important thing to document in a framework is the interactions between classes* »[Jol99, p. 160].

	Notation graphique	Protocole d'entrée	Protocole de sortie	Concepts objets	Concepts algorithmiques	Représentation de l'état interne	Modèle Formel	Vérification statique	Modularité	Familiarité
Diagrammes de séquences	✓	★	★★	★	★	✓	✗	✗	✓	✓
Diagrammes de collaboration	✓	★	★	✗	★	✗	✗	✗	✓	✓
Diagrammes d'états transitions	✓	★★★	★★	★★	★★	✓✗	✓✗	✓✗	✗	✓
ADL	✗	★★★	★★★	★★	★★	✓✗	✓	✓	✓	✗
Expression de chemins	✗	★★	✗	✗	★★	✗	✓	✓	✗	✓
Regular types	✗	★★★	✗	★	★	✗	✓	✓	✓	✗
Yellin	✗	★★	★★★	★	★★	✗	✓	✓	✓	✗
Contrats Helm	✗	★	★★	★	★★	✓	✓	✓	✓	✓

✓ : Oui ★...★★★ : Évaluation subjective du critère
✗ : Non ✓✗ : Indique que selon les variations utilisées de la technique, le critère peut ou non être valide

TAB. 5.1 – Comparatif des techniques de représentation des dépendances comportementales.

Chapitre 6

Présentation du modèle synchrone

Afin de représenter le protocole des composants, nous avons choisi le modèle synchrone. Il s'agit d'une spécialisation de la théorie des automates. Les raisons de ce choix sont multiples :

- il repose sur une sémantique formelle permettant de faire des preuves automatiques ;
- il est fourni avec une plate-forme complète de développement fournissant des vérificateurs de modèles et des simulateurs ;
- il est supporté par de nombreuses notations à la fois graphiques et textuelles ;
- il a montré son applicabilité dans divers domaines tels les protocoles de communication, la synthèse électronique, des contrôleurs temps-réels, des interfaces graphiques, etc.

De plus, à notre connaissance, il n'a jamais été utilisé pour la spécification de composants logiciels malgré ses capacités à faire des preuves.

6.1 Systèmes réactifs

Les systèmes réactifs sont à la base de langages réactifs synchrones ou asynchrones. Un système réactif est un système qui ne réagit qu'aux stimuli de son environnement, et à un rythme imposé par celui-ci. C'est cela qui le différencie d'un système transformationnel ou d'un système interactif[1].

La conception d'un système réactif « doit au moins prendre en compte le parallélisme d'exécution du système et de son environnement. De plus, il est très souvent commode et naturel de concevoir un tel système comme un ensemble de composants s'exécutant en parallèle et coopérant à la réalisation du comportement souhaité. Il convient d'insister sur le fait que la décomposition d'un programme en activités pa-

[1] Un système réactif n'est actif que sur la réception d'un événement, alors qu'un système interactif peut faire des opérations entre les occurrences des événements

rallèles n'implique pas forcément une implémentation parallèle, et que, même dans ce cas, ce parallélisme d'implémentation ne correspond pas nécessairement à la décomposition initiale.

Les systèmes réactifs sont soumis à des contraintes temporelles strictes. Il s'agit en général de systèmes déterministes. Ils sont soumis à des contraintes de sûreté particulièrement sévères » [Hal91].

Les exigences liées à la sûreté de fonctionnement des systèmes réactifs ont nécessité la mise en place de procédures de validation des comportements des programmes écrits grâce à des langages dits réactifs. Ceci a été rendu possible essentiellement parce que la description formelle de la sémantique de ces langages permet de dériver de tout programme un modèle formel de son comportement sur lequel il est possible d'appliquer des algorithmes de vérification de modèle. Cependant il est à noter que les outils de vérification ne fonctionnent que sur un ensemble fini d'états, et ne peuvent ainsi donc pas gérer l'ajout et la suppression dynamique d'états.

6.2 Systèmes réactifs synchrones

Les systèmes réactifs synchrones ont été introduits pour faciliter la tâche du programmeur de systèmes réactifs en lui fournissant des primitives idéales (elles sont atomiques), permettant de raisonner comme si le programme réagissait instantanément aux éléments externes. Chaque événement du programme est précisément identifiable par rapport au flot des éléments externes, et le comportement du programme est complètement déterministe, tant du point de vue fonctionnel que temporel. Le temps physique est remplacé par un temps multiforme où tout événement est une base de temps potentielle. Il y a donc une relation d'ordre entre événements où les notions importantes sont celles de simultanéité et de précédence entre événements.

Quand nous parlerons d'instant, cette notion devra être comprise comme celle d'instant logique : l'histoire d'un système est une succession totalement ordonnée d'instants logiques, à chacun desquels 0,1 ou plusieurs événements surviennent. Les occurrences des éléments survenant au même instant logique seront considérées comme simultanées, celles survenant à des instants différents seront considérées comme survenant dans l'ordre de leur instant d'occurrence. La réaction à la réception d'un signal est atomique (elle s'exécute dans l'instant et aucun autre événement externe ne peut être pris en compte pendant la réaction). En dehors des instants logiques, rien ne se passe ni dans le système ni dans son environnement. En pratique, l'hypothèse de synchronisme revient à supposer que le programme réagit assez vite pour percevoir tous les éléments externes en bon ordre. L'émission d'un signal est aussi instantanée, ainsi plusieurs signaux peuvent être émis instantanément.

Ce principe d'émission et de réception instantanées de signaux est à l'origine des problèmes dit de causalité (on parle aussi de « cycles de causalité » ou de « paradoxe temporel »). Un cycle de causalité décrit l'absence d'une solution synchrone (par exemple un signal doit être émis si et seulement s'il est absent). Ces paradoxes sont difficilement évitables dans la programmation courante, et sont une nuisance pour le programmeur.

Les langages synchrones sont susceptibles d'être implémentés de manière particulièrement efficace et mesurable. Le code produit a la structure d'un automate fini, chaque transition correspondant à une réaction du programme.

6.3 Esterel, un langage synchrone

Le premier langage synchrone qui a été mis au point, et celui sur lequel nous baserons nos travaux est Esterel [BG92]. C'est un langage synchrone impératif textuel. Il communique avec son environnement au moyen de signaux et de capteurs. Les signaux servent en entrée / sortie, alors que les capteurs ne servent qu'en entrée. Les signaux / capteurs peuvent être valués.

L'unité de structuration d'un programme Esterel est le module. Il définit des signaux d'entrée et de sortie et bien sûr le comportement qui les lie les uns aux autres. Chaque module est autonome (il représente une unité d'exécution). L'exécution d'un module est démarrée par une instruction spécifique, l'instruction `run`. Pour décrire un module en cours d'exécution on parle encore de processus. Un processus s'arrête implicitement car il atteint un état final, ou explicitement par préemption.

La réalisation d'un programme Esterel fait souvent appel à des modules existants qui doivent collaborer. La connexion entre ces modules est faite automatiquement par correspondance entre les noms des signaux. S'il n'y a pas correspondance, les signaux du module appelé peuvent être renommés lors de son démarrage (figure 6.2). Ces modules peuvent être exécutés séquentiellement ou en parallèle. Le parallélisme est une construction du langage et la composition comportementale de modules est parfaitement définie.

Les signaux sont diffusés à tous les processus du programme (modulo le renommage et des règles de visibilité). Ainsi lorsqu'un signal est émis il est instantanément perçu par tous les processus. En Esterel, le contrôle est instantané. L'occurrence de signaux d'entrée peut provoquer instantanément l'émission d'autres signaux, c'est une conséquence directe du modèle synchrone idéal. Si le signal déclenche une action, celle-ci est effectuée, dans le cas contraire le signal est ignoré. Le langage intègre de manière orthogonale la concurrence et la préemption. Il introduit deux formes de préemptions, la préemption faible et la préemption forte. Une préemption faible sort de l'état courant sur réception d'un signal, mais la réaction associée à l'instant est tout de même exécutée, alors que la préemption forte empêche cette exécution.

La sémantique d'Esterel a été exprimée dans trois modèles différents : une sémantique mathématique, une sémantique constructiviste, et une sémantique de logique des circuits. Ces études sémantiques permettent entre autre de montrer que la composition parallèle de modules génère un automate contenant l'entrelacement optimal du comportement du programme.

Esterel est fourni avec de nombreux outils :

- un vérificateur de modèles (XEVE [BMDT96, Bou97]) qui permet de vérifier sur un programme des propriétés exprimées en logique temporelle ;
- un simulateur (XeS) qui permet de simuler le comportement d'un programme ;
- des générateurs de code pour OC, Java, C, BLIFF, etc. Nous portons un intérêt tout particulier au premier car ce fichier résultant de la compilation d'un pro-

gramme Esterel contient, dans une forme où le parallélisme initial est résolu, l'automate équivalent au programme Esterel. Ainsi il peut être, avec les traducteurs adéquats, utilisé en entrée de nombreux outils manipulant les automates.

Puisque dans nos travaux nous ne faisons qu'une utilisation indirecte de ce langage et utilisons une représentation graphique équivalente, nous ne donnons pas davantage de détails sur ce langage. Ainsi pour une plus ample description d'Esterel nous vous invitons à consulter « *The Esterel Language Primer* » [Ber00a].

6.4 SyncCharts, une représentation graphique et formelle

Les SyncCharts (Synchronous Charts) sont un modèle graphique synchrone dont le but est de faciliter la modélisation des systèmes réactifs. « *Ils intègrent les concepts de haut niveau des langages synchrones dans un formalisme graphique expressif* » [And96a], et permettent ainsi une représentation simple de la communication, du parallélisme et de la préemption au sein d'un système. Leurs représentations héritent graphiquement des Statecharts [Har87] et d'Argos [Mar90] mais offrent un meilleur moyen d'expliciter la préemption. Leur sémantique mathématique stricte [And96a] est basée sur celle du langage Esterel [Ber00b, BG92] dans lequel ils se traduisent directement[2]. Cette projection permet donc de bénéficier de l'ensemble des outils liés à Esterel. Il est à noter que cette notation graphique est utilisée comme moyen principal de saisie de programme dans l'environnement commercial Esterel Studio [Tec].

6.4.1 Exemple de SyncCharts

Afin de présenter la notation graphique, nous reprenons comme illustration l'exemple employé par les concepteurs des SyncCharts dans [And96a].

Ce système a trois entrées `A`,`B`,`R` et une sortie `O`. `O` doit être émis dès qu'il y a eu occurrences de `A` et `B`, depuis l'instant initial ou depuis l'occurrence précédente de `R`. `R` est un signal prioritaire qui remet le système dans son état initial (`R`eset). On trouve dans cet exemple un mélange de séquentialité, parallélisme et préemption :

- contrôle séquentiel : `O` doit être émis après la réception de `A` et `B` ;
- contrôle parallèle : on attend les occurrences de `A` et `B` séparément ;
- préemption : l'occurrence de `R` fait avorter (immédiatement) les attentes de `A` et `B`.

6.4.2 Description de la notation graphique

- Etat :
  - un cercle est un état ;
  - un rond plein noir est un état initial ;
  - un cercle doublé est un état final ;

[2]Afin de simplifier le discours, on considèrera qu'un SyncCharts se traduit en un module Esterel. En réalité un SyncCharts se traduit en autant de modules Esterel qu'il y a de macro-états constituant le SyncCharts [And96b].

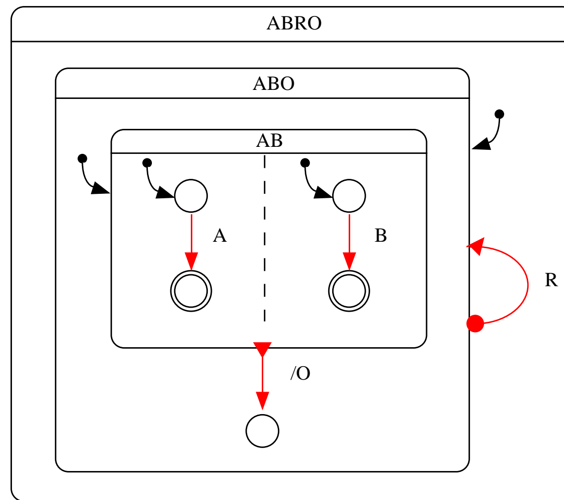


FIG. 6.1 – SyncCharts de l'exemple ABRO, extrait de [And96a].

```
module ABRO :
   input A,B,R;
   output O;
   loop
     abort
       run ABO; // dans ce cas la connexion
              //se fait par l'identité des noms de
signaux
     when R
   end loop
end module

module ABO
   input A,B;
   output O;
   await A
   ||
   await B;
   emit O
end module
```

FIG. 6.2 – Code Esterel correspondant au SyncCharts ABRO (cf figure 6.1).

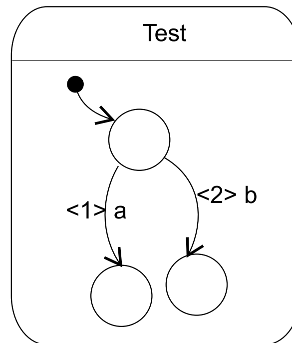
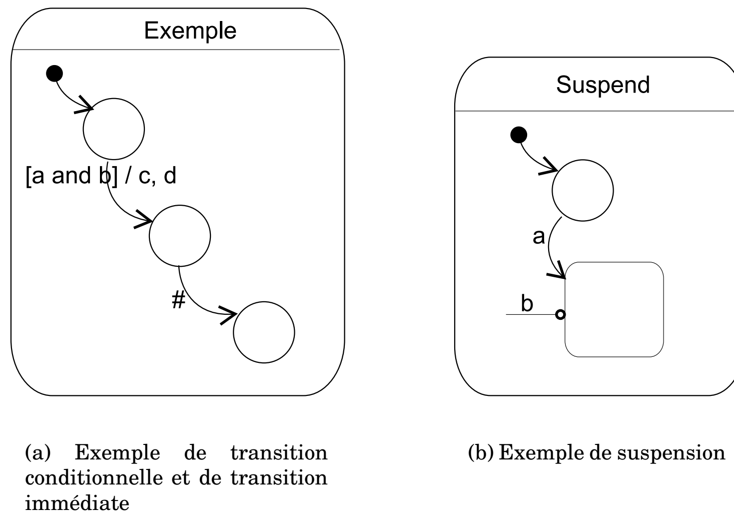


FIG. 6.3 – Transitions avec priorité dans un SyncCharts.

  - un macro-état est un état qui englobe d'autres états et des transitions ;
- Transition :
  - une transition dont la flèche n'a pas d'extrémité particulière est une préemption faible. On sort de l'état courant sur réception de ce signal, mais la réaction associée à l'instant est tout de même exécutée ;
  - une transition commençant par un rond noir est une préemption forte (on sort de l'état courant immédiatement à la réception du signal qui étiquette l'arc. La réaction associée à l'instant n'est pas exécutée) ;
  - une transition qui commence par un triangle est une terminaison normale. Elle est automatiquement empruntée. Elle est traversée spontanément lorsque l'état d'où elle débute est dans un état final ;
  - un entier sur l'extrémité d'une transition indique la priorité avec laquelle une transition doit être empruntée lorsque les signaux déclenchant les transitions arrivent simultanément (figure 6.3) ;
  - le label d'une transition décrit en partie gauche du « `/` » le signal attendu et en partie droite les signaux émis (ces signaux, séparés par une virgule sont émis simultanément). Une condition se présente entre crochets et peut porter sur la valeur des signaux. Si le label de la partie gauche est précédé d'un # la transition est empruntée immédiatement (dans l'instant) (figure 6.4(a)) ;
  - une transition ayant pour extrémité un rond vide et n'étant attachée qu'à une extrémité à un macro-état indique la suspension. Le comportement du macro-état est suspendu pendant tout le temps où l'expression portée par le label de la transition est vraie (exemple avec le signal `b` figure 6.4(b)).
- Une ligne pointillée sépare des états concurrents ;
- Une liste de déclarations en bas d'un macro état indique des signaux locaux.

La liste des signaux en entrée et sortie fait partie des informations disponibles pour chaque SyncCharts, quand bien même elle n'a pas de représentation graphique. Cette liste peut être étendue par des relations entre les signaux. Cela permet par exemple d'indiquer qu'un signal ne peut pas être présent au même instant qu'un autre.

Bien que la liste des éléments graphiques puisse paraître longue, sa mise en œuvre se révèle simple et intuitive. C'est ce point de simplicité, mais surtout sa sémantique



(a) Exemple de transition conditionnelle et de transition immédiate

(b) Exemple de suspension

FIG. 6.4 – Exemples d'utilisation des SyncCharts.

précise [And96a] et l'existence d'une représentation textuelle équivalente en Esterel [And96b] ainsi que les outils existants qui nous ont poussés à retenir les SyncCharts comme moyen d'expression de nos dépendances comportementales.

Cependant, l'utilisation de cette notation et de ce modèle pour la représentation du protocole de composants d'un framework n'est pas immédiate et requiert certaines adaptations qui font l'objet d'explications au chapitre 7.

Problématique des dépendances comportementales

Le modèle d'expression des dépendances que nous avons proposé jusqu'alors ne tient compte que des dépendances structurelles entre les éléments et est donc plus adapté à la résolution des problèmes liés à la personnalisation du framework. Cet aspect, certes fondamental, n'est pas suffisant pour l'utilisateur du framework qui a besoin d'informations sur le fonctionnement des composants surtout lorsqu'il est amené à les assembler dans une réutilisation de type instanciation. Il faut donc aussi s'intéresser aux dépendances comportementales entre les éléments.

Afin d'utiliser une classe, connaître le fonctionnement de chaque méthode indépendamment les unes des autres ne suffit pas. En effet bien qu'elles ne soient pas explicites, il existe souvent des dépendances entre les méthodes, formant le protocole de la classe. Lorsqu'à l'exécution ce dernier n'est pas respecté, il en résulte un comportement inattendu des instances.

Ce risque d'erreur est accru dans le contexte des frameworks : la mauvaise utilisation d'une instance pouvant avoir des effets néfastes à retardement. Ainsi, une instance mal utilisée peut lorsqu'elle est utilisée par une autre classe être responsable d'une erreur sur cette dernière. Nous utilisons le qualificatif « retardement » en contraste avec « immédiate ». Ce dernier décrit une situation où l'erreur est la conséquence immédiate de l'exécution d'une méthode.

Les erreurs « à retardement » sont rendues plus fréquentes dans les frameworks où le couplage entre les classes est fort. Ainsi l'écriture d'une classe qui doit venir s'intégrer par composition ou spécialisation à un framework est délicate puisqu'il faut prendre en compte la manière dont celle-ci est utilisée par des clients, mais aussi la manière dont la classe qu'elle remplace (ou sous-classe) communique avec les autres classes. Cela montre la difficulté de l'écriture d'une classe pour un framework, mais aussi le caractère insuffisant d'une liste de signatures de méthodes (i.e. une interface) pour décrire une classe. Pour s'en convaincre, il suffit d'implémenter toutes les méthodes d'une classe par des méthodes vides. La classe résultante se conforme bien à l'interface et ne pose aucun problème à la compilation, cependant son comportement et ses interactions avec les autres classes sont erronés. « Rien n'empêche, du point de

vue du typage de faire hériter la classe Machine-à-laver d'une classe Camion, même s'il faut être bien fatigué pour en arriver là. Et avoir la certitude qu'une routine existe, même avec la contrainte d'une signature conforme, ne garantit évidemment pas que l'effet obtenu sera celui souhaité » [Roger Rousseau [DEMN98]]

Ainsi on constate qu'il est important, pour utiliser sans erreur un framework (et à moins grande échelle une classe), de connaître :

– le protocole de la classe lorsqu'on est client de celle-ci ;
– les interactions d'une classe avec les autres lorsqu'on doit la remplacer ou la sous-classer.

Il faut noter que les problèmes liés au sous-classement tel que vérifier que la nouvelle sous-classe n'endommage pas le comportement de la super-classe ne sont pas traités dans le cadre de cette étude (ce problème est aussi connu sous le nom Fragile Base Class problem). Ainsi les travaux de Lamping [Lam93], Steyeart [SLMD96] ou Ruby [RL00] qui donnent des ensembles de lignes directrices et d'opérations à éviter lors du sous-classement d'une classe sont complémentaires aux nôtres. En effet, aucun d'entre eux ne prend en compte le protocole de la classe.

L'incapacité de l'interface d'une classe à pouvoir capturer l'essence du comportement de la classe qu'elle décrit vient, au-delà du manque de techniques simples pour intégrer cette information à la classe, du fait que l'information que l'on aimerait voir dans les interfaces ne peut pas être simplement générée et doit donc être fournie par le concepteur de la classe. Il est surprenant qu'en phase de spécification, le protocole ne soit pas clairement identifié et ne fasse pas l'objet d'un diagramme particulier, surtout dans les cycles de développement longs comme le RUP [JBR99] (ou ses prédécesseurs Booch, OMT...) où différents niveaux de spécification sont abordés. Il est tout aussi surprenant de constater que ces mêmes diagrammes ne sont utilisés qu'en phase de développement et rarement fournis à l'utilisateur final (spécialement dans le cas d'un framework), que ce soit sous forme de documentation ou intégrés au sein même des composants du framework. On peut supposer que cela vient du manque d'outil, de l'inadéquation de la documentation avec ce qu'est en droit d'attendre l'utilisateur final, ou encore de la désynchronisation qui existe entre la spécification et le code. Cela indique qu'il y a un problème dans le suivi de la documentation. Si l'utilisateur final pouvait accéder à cette documentation, elle lui serait d'un grand secours, et donnerait peut-être une motivation supplémentaire au développeur pour la garder en synchronisation.

Dans cette partie nous proposons une technique de documentation des dépendances entre composants appelée points de vue comportementaux*. Ils décrivent la coopération entre les composants d'un framework ainsi que leur protocole. Ils suivent ces composants de la phase de conception au développement et y sont ensuite intégrés pour servir de documentation mais surtout pour effectuer des vérifications statiques et dynamiques.

Le chapitre 7 présente tout d'abord le concept des points de vue comportementaux avant d'introduire les SyncClass, la notation graphique que nous avons mise au point pour les représenter. La projection de cette notation dans le modèle synchrone est détaillée donnant un fondement formel à notre notation.

Le chapitre 8 présente l'utilisation des points de vue comportementaux dans le cycle de vie d'un composant. Nous y décrivons aussi une proposition d'environnement (Co^2 - *Co*mposition et *Co*mposant) dédié au développement et à la manipulation de composants.

Chapitre 7

Points de vue comportementaux

« *So far, progress has been slow, perhaps because too much emphasis has been put on components and too little on how they are composed.* » [NM95]

Dans la suite de cette dissertation nous utiliserons indistinctement les termes de composants et classes. Pour nous, un composant est caractérisé par les propriétés suivantes :

1. c'est une entité de composition ;
2. il est encapsulé et promeut l'utilisation « boîte noire » ;
3. il fournit une interface d'entrée (une liste d'opérations et leur signature) ;
4. il fournit une interface de sortie (un ensemble d'opérations appelées par le composant) ;
5. il constitue une entité d'exécution, agissant comme une entité réactive [LM95] ;
6. il contient un unique flot de contrôle (*thread*) (la réentrance n'est pas autorisée) ;
7. Ses méthodes s'exécutent jusqu'à leur terminaison.

Les quatre premiers points sont en accord avec la définition de Szyperski qui indique qu'un composant est « *a unit of composition with contractually specified interfaces and explicit context dependencies* » [Szy98]. Les trois points suivants qui décrivent le modèle d'exécution constituent une limitation de notre modèle. En effet, les travaux que nous présentons dans la suite de ce document ne supportent pas un modèle général de concurrence. Ainsi notre modèle se restreint à un modèle purement séquentiel qui malgré tout est utile dans de nombreux cas [MRR01].

Ainsi cette définition englobe à la fois les modèles de composants industriels (Corba CCM, EJB ou Java Beans) et la simple classe qui est dans notre cas le constituant essentiel des frameworks.

Afin de pallier les manques évoqués précédemment nous proposons le concept de point de vue comportemental*. Un point de vue comportemental permet de décrire le comportement des instances d'une classe en mettant en avant leur protocole d'utilisation et leurs interactions *sans jamais recourir explicitement à leur état interne*. Contrairement aux diagrammes de séquences il décrit l'ensemble des scenarii possibles. Ce

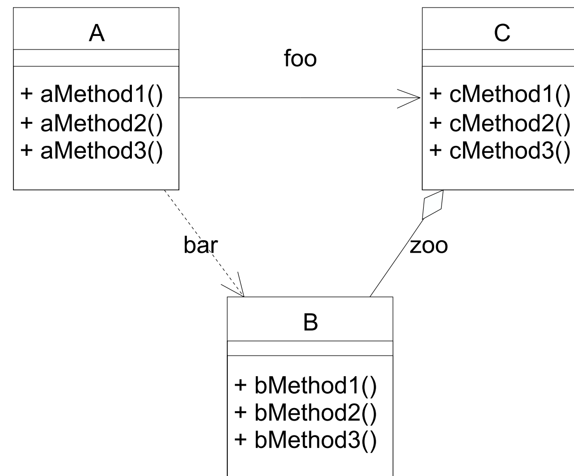


FIG. 7.1 – Modélisation UML d'un système constitué de 3 classes.

concept est graphiquement représenté par la notation de SyncClass. Les points de vue comportementaux n'ont pas pour vocation de remplacer les diagrammes UML existants lorsqu'ils sont utilisés pour décrire le fonctionnement interne des classes. Ils les complètent en offrant un moyen simple et synthétique de capturer le protocole d'un composant en liant les appels de méthodes en entrée avec ceux en sortie. Ainsi, ils sont donc plus adaptés pour remplacer les diagrammes d'états quand ils sont utilisés comme *protocol state machines* [OMG01b, p. 2-175].

Contrairement à l'étude menée par Sylvain Vauttier dans COBALT [Vau97], nous ne nous limitons pas à l'étude de la relation composant / composite, mais cherchons avec ces points de vue comportementaux à expliciter tout type de coopération entre composants.

Dans la présentation qui va suivre, nous prenons comme exemple celui décrit figure 7.1 de manière à pouvoir donner exhaustivement tous les points de vue.

7.1 Point de vue client

7.1.1 But

Le point de vue client* indique comment utiliser une classe en terme d'envoi de messages. Il décrit le protocole de la classe à laquelle il est associé et indique ainsi la séquence des messages autorisés par une instance afin qu'elle fonctionne correctement. On parle encore de protocole d'utilisation*. Ainsi sur l'exemple des trois classes (figure 7.1), le point de vue client pour `A` indique l'ordre dans lequel les trois méthodes `aMethod1`, `aMethod2`, `aMethod3` doivent être appelées. En plus de cette précision sur le protocole, cette vue présente les appels de méthodes effectués par chaque méthode de l'interface du composant que l'on décrit. On parle ici de protocole d'inter-dépendances*.

Ainsi dans l'exemple, cette deuxième propriété de la vue permet de savoir comment chaque méthode de la classe `A` interagit avec les classes `B` et `C`.

Le protocole d'utilisation est uniquement constitué de méthodes accessibles aux autres composants (publique, protégée, ami, etc.). De même, le protocole d'inter-dépendance n'accède qu'aux méthodes accessibles des composants appelés.

Sur l'exemple on peut recenser trois points de vue client décrivant chacun le protocole d'utilisation d'une classe : un pour la classe `A`, un pour la classe `B` et un pour la classe `C`.

7.1.2 Construction

La construction d'un tel point de vue ne peut pas être complètement automatisée puisqu'il semble actuellement encore très difficile d'extraire du code d'une classe la manière dont celle-ci doit être utilisée. La partie qui doit donc être effectuée manuellement est la création du squelette de l'enchaînement. Les informations relatives aux messages émis peuvent être extraites du code des méthodes quand celui-ci est disponible.

7.1.3 Fonctionnalités

Ce point de vue qui spécifie le protocole d'utilisation d'une classe et le lie avec celui d'inter-dépendance permet de visualiser et comprendre comment les classes interagissent, ainsi que les dépendances qui les lient. Puisqu'il indique la dépendance des classes, il peut aussi en indiquer l'indépendance. En effet, si le point de vue client ne fait état d'aucune dépendance avec d'autres classes alors la classe peut être considérée comme indépendante et être réutilisée « seule » (afin d'utiliser ladite classe aucune autre classe ne sera nécessaire).

Lors de l'écriture d'une classe, le point de vue client permet de visualiser les appels que doivent effectuer chaque méthode, ce qui le différencie de l'*ACtual behavior* des CHAMs dont il est très proche par le but (sous-section 5.4.3).

Ce point de vue permet, s'il est basé sur une représentation formelle, d'effectuer des vérifications statiques et dynamiques.

7.2 Point de vue de composition

7.2.1 But

Le point de vue de composition* est une vue synthétique qui décrit l'utilisation que fait une et une seule classe d'une autre. Cette vue synthétique se focalise sur une classe donnée du protocole d'inter-dépendance*. Elle est portée par la classe dont elle décrit les connexions. Nous avons fait ce choix car les informations qu'elle contient, bien qu'appartenant à une autre classe, lui sont propres (elles reflètent une partie de son comportement et ses pré-requis sur l'autre composant).

Une classe peut avoir autant de points de vue de composition qu'elle a d'associations. Cette distinction est nécessaire car une classe peut utiliser plusieurs fois la même classe. On retrouve ici la notion de rôle utilisée en UML, où une classe peut être utilisée

selon différentes interfaces (une interface est la réalisation d'un rôle). A la différence du point de vue client qui décrit le protocole d'inter-dépendances entre un composant et tous les composants qu'il utilise, le point de vue de composition ne donne que le protocole d'inter-dépendance entre deux composants. Ainsi la classe `A` a un point de vue de composition sur `B` et un sur `C`, et la classe `C` a un point de vue de composition sur la classe `A` et `B`. Les points de vue de composition ne sont pas limités à l'orientation des associations. Ainsi `B` a un point de vue de composition sur `A` et un sur `C`. Les classes `A` et `B` ne présentent dans leurs points de vue de composition que la partie du protocole de `C` qu'elles utilisent. Le point de vue de composition de `A` sur `B` ne décrit que les appels de méthode effectués par `A` sur `B`.

Le point de vue de composition est un sous-protocole du protocole d'inter-dépendance décrit par les points de vue client puisqu'il ne représente que les appels de méthodes effectués par une classe sur une autre. Il peut aussi être vu comme étant une vue partielle du point de vue client d'une classe : par exemple le point de vue de composition de `A` sur `B` est partiellement le point de vue client de `B`.

Afin de représenter les informations sur la connexion entre deux composants, les points de vue de composition décrivent aussi les *callbacks* du composant dont le point de vue de composition fait l'objet. Cette information synthétique permet simplement à un utilisateur de mieux visualiser les connexions. Par exemple, lors de l'étude du point de vue de composition de `A` sur `B`, cela lui indique si `B` fait des appels sur A. Ces callbacks sont particulièrement intéressants pour les frameworks appelés.

7.2.2 Construction

Pour les raisons évoquées précédemment, les points de vue de composition sont une représentation partielle (de la partie inter-dépendance) du point de vue client. Aussi, ces points de vue de composition peuvent dans une certaine limite être construits à partir des points de vue client. En effet, les callbacks dépendant du composant qui fait l'objet du point de vue, cette information ne pourra pas être déduite du point de vue client et requerra le code source du composant.

7.2.3 Fonctionnalités

Cette notation offre un complément d'information, ainsi qu'une représentation plus synthétique de la connexion entre deux composants. Tout comme les points de vue client, les points de vue de composition permettent de faire de la vérification statique et dynamique et sont utiles pour la documentation. Bien qu'en théorie le nombre de points de vue de composition associé à chaque classe puisse être grand, l'utilisateur n'est pas obligé de tous les construire afin de bénéficier des outils de vérification.

Cette vue s'identifie à l'*Assumed behavior* de [IWY00] (sous-section 5.4.3), puisqu'elle décrit les suppositions faites par le composant sur le comportement des autres composants pour bien fonctionner. Les points de vue de composition contiennent en plus des *Assumed behavior* les callbacks.

7.3 L'encapsulation

Les points de vue client ou de composition représentant le protocole d'inter-dépendance, et donc une partie du contenu de certaines méthodes, on peut être amené à penser que cette description du comportement viole l'encapsulation qu'est justement censé respecter une classe.

Cependant, si l'on regarde précisément les informations requises à la description de la connexion entre les composants, on constate qu'il est absolument nécessaire pour la composition de composants de connaître ces interactions vers l'extérieur, en d'autres termes de connaître l'interface de sortie* des composants. Une interface de sortie décrit les méthodes utilisées et événements émis par un élément. Cette interface de sortie s'apparente au concept de ports ajouté à ROOM [SGW94] puisque, dans les systèmes réactifs, ces interfaces sont la base de toute connexion entre capsules. Ce concept est aussi présent dans les composants CORBA [OMG01a]. Une interface de sortie n'est pas contraire à la notion d'encapsulation. Pour étayer notre raisonnement nous utiliserons le classique parallèle entre composant logiciel et composant électronique. On peut constater que la spécification d'un composant électronique est faite d'un ensemble de pré-requis comme les tensions, les protocoles de communication attendus sur les entrées, mais aussi d'une description précise des sorties par rapport à ses entrées sans que cela ne nous indique ce qui se passe précisément dans le composant, et donc sans casser l'encapsulation du composant. Ainsi, les points de vue comportementaux ne faisant rien d'autre que décrire l'interface de sortie d'un composant logiciel en fonction de ses entrées, cela ne casse aucunement l'encapsulation.

Dans le contexte des diagrammes de classes UML ou des diagrammes de composants, les points de vue spécifient le protocole de l'interface de sortie des éléments. Cela revient à spécifier la connexion qui, dans les diagrammes de classes est représentée par une association, ou encore dans les diagrammes de composants à préciser la sémantique des dépendances entre les composants (montré dans un diagramme de composants par un lien entre un composant et une interface [OMG01b, p. 463]). En effet, ces associations sont des manifestations du comportement dans une représentation structurelle, et peuvent être représentées par des diagrammes de collaboration, de séquences, ou encore des points de vue comportemantaux. Par exemple, une dépendance `instantiate` indique que l'on va instancier l'élément pointé et sera représentée par l'envoi d'un message `new` dans un diagramme de séquences, une aggrégation va être à l'origine d'une variable qui devra elle aussi être utilisée selon le protocole de la classe dont elle est instance. Si l'on conçoit le lien comme un fil, les points de vue modélisent complètement le protocole de communication utilisé sur celui-ci.

Puisque nous désirons montrer toutes les méthodes appelées sur des composants il est alors nécessaire de représenter les appels à des composants qui pourraient être contenus dans les méthodes privées. Comme les méthodes privées ne prennent pas part au protocole, nous avons choisi d'intégrer les hypothétiques appels qu'elles font directement à l'emplacement de leur appel. Cela est comparable à des techniques d'*inlining*. Ainsi dans une situation où l'on a `foo() { bar(); tx.x() }` (et `bar` est une méthode privée appelant `v.z()`), alors le corps de `foo` dans la représentation des points de vue comportementaux indiquera l'appel à la méthode publique contenu dans la méthode privée.

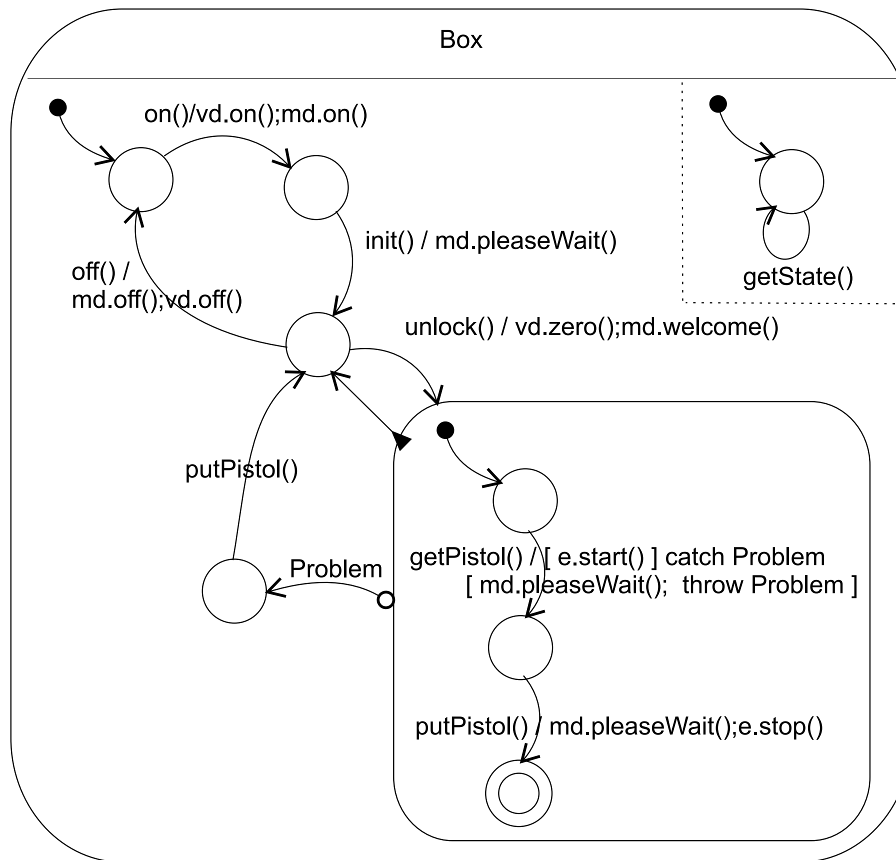


FIG. 7.2 – Point de vue client de la pompe à essence.

L'algorithme est alors le suivant : pour chaque méthode privée appelée dans le corps d'une méthode appartenant au protocole, on remplace l'appel à la méthode privée par son corps et cela de manière récursive.

7.4 Exemple de l'organisation des points de vue

Dans cette section nous montrons quels sont les différents points de vue pour l'exemple de la pompe à essence utilisé dans l'état de l'art. La notation graphique utilisée semblant intuitive, nous ne la présenterons formellement qu'après avoir mis l'accent sur le concept de point de vue comportemental.

Nous présentons successivement le point de vue client de la pompe (figure 7.2), celui du moteur (figure 7.3) ainsi que de l'afficheur de volume (figure 7.4). Nous montrons ensuite quelques points de vue de composition.

La figure 7.2 décrit le protocole de la pompe à essence. On peut constater que pour

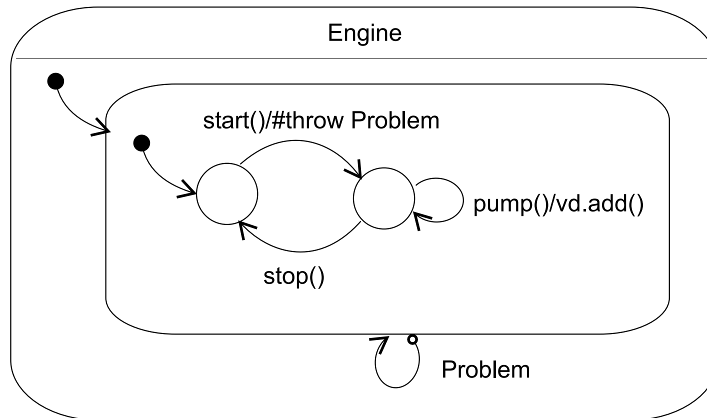


FIG. 7.3 – Point de vue client du moteur.

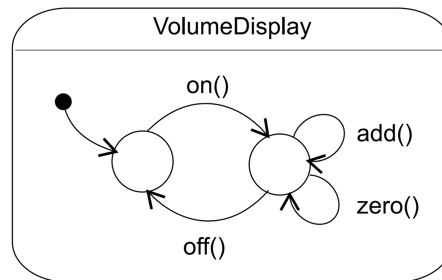


FIG. 7.4 – Point de vue client de l'afficheur de volume.

correctement utiliser la pompe il faut commencer par appeler la méthode `on` puis la méthode `init`... et que l'on peut aussi appeler à tout moment la méthode `getState`. L'imbrication des états permet simplement une décomposition hiérarchique du comportement à la manière des StateCharts [Har87]. Les transitions représentent des appels de méthodes. Les boucles ou les transitions quittant un macro-état sont des facilités de notation factorisant des transitions à partir de chacun des états contenus dans le macro-état. La notation pointée utilisée sur les transitions référence le composant à qui l'appel de la méthode est destiné.

La figure 7.3 montre qu'afin d'utiliser le moteur correctement, on doit utiliser la méthode `start` (qui peut lever une exception), puis la méthode `pump` qui appelle la méthode `add` sur l'afficheur de volume.

La vue cliente de l'afficheur de message (figure 7.4) permet de voir que ce composant est indépendant : il n'utilise les services d'aucun autre composant. Ainsi, si l'on désire le réutiliser on peut le faire sans se soucier d'autres composants.

Le point de vue de composition du moteur présenté figure 7.5 permet, en comparaison avec le point de vue client de l'afficheur de volume (figure 7.4), de voir que l'afficheur n'est pas seulement utilisé par le moteur. En effet le protocole complet de l'afficheur

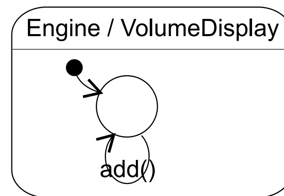


FIG. 7.5 – Point de vue de composition du moteur sur l'afficheur de volume.

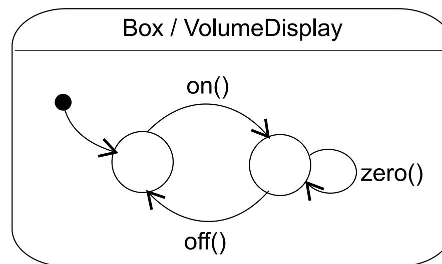


FIG. 7.6 – Point de vue de composition de la pompe sur l'afficheur de volume.

met en œuvre quatre méthodes alors que le moteur n'en utilise qu'une seule. En conséquence, quelqu'un voyant ce diagramme et voulant utiliser l'ensemble des composants doit se demander si les méthodes non présentes sont utilisées dans d'autres parties du système, ou si c'est à lui de les appeler. En voyant le point de vue de composition de la figure 7.6 il constate que l'ensemble des méthodes de l'afficheur sont utilisées par le composant puisque la pompe appelle `on`, `zero`, et `off`.

Parmi toutes les vues de composition du système, nous avons retenu celles qui portaient sur l'utilisation de l'afficheur de volume. Celles-ci ont l'avantage de montrer, pour chaque composant, la manière dont l'afficheur de volume est utilisé et elles permettent aussi de voir la dépendance qui existe entre les composants utilisant l'afficheur.

7.5 SyncClass, une représentation graphique des points de vue comportementaux

Afin de représenter les points de vue comportementaux, nous avons créé la notation des SyncClass. Les SyncClass sont à la fois une restriction et une extension syntaxique et sémantique des SyncCharts [And96a] (eux-mêmes une modification des StateCharts) intégrant en plus quelques notions de programmation objet ainsi que des structures de contrôle.

7.5.1 Notation graphique

La notation reprend aux SyncCharts les notions d'état initial (un rond plein noir), d'état terminal (un cercle doublé) et de macro-état (un rectangle aux angles arrondis). Il est à noter qu'un état représente seulement l'historique des appels qu'a subi l'objet. C'est d'ailleurs la sémantique des premiers diagrammes d'états-transitions d'Harel.

Une ligne pointillée est utilisée pour exprimer la concurrence logique entre des ensembles de méthodes qui peuvent être appelés indépendamment les uns des autres.

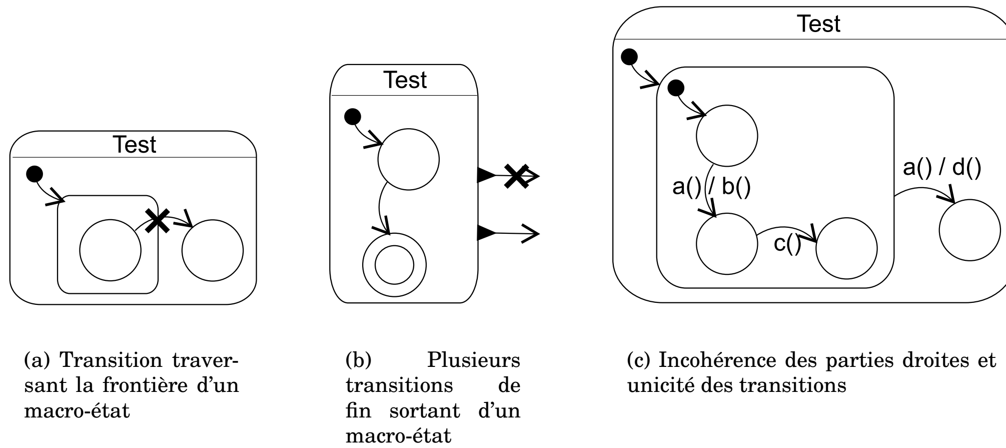
Il existe trois types de transitions :

1. transition de fin. Une transition commençant par un triangle marque une transition spontanée vers l'état sur lequel elle pointe. Elle est empruntée automatiquement lorsque l'état d'où elle sort est dans un état terminal. Cette transition ne peut être ni étiquetée, ni conditionnelle ;
2. transition exceptionnelle. Une transition commençant par un rond permet de représenter l'adaptation du protocole d'un composant lorsqu'une exception est levée. Cette transition est empruntée quand un élément contenu dans le macro-état d'où débute la transition lève une exception. L'étiquette portée par la transition indique la liste des exceptions pour laquelle elle sera empruntée (exemple : `E1, E2`). Il est important de noter qu'il ne s'agit pas d'un `catch` au sens des langages comme C++ ou Java et que l'exception n'est pas stoppée ;
3. transition par message. Une transition (sans particularité) représente la réception d'un appel de méthode. Sur la transition, la partie à gauche du « / » comporte uniquement la signature de la méthode appelée (on parle aussi de *trigger*), et la partie à droite représente le flot des messages émis vers d'autres composants (ce qui constitue un squelette des algorithmes des méthodes, on parle aussi de partie action). Comme de nombreux langages objets, les SyncClass utilisent la notation pointée pour indiquer le receveur du message (les appels de messages peuvent être synchrones ou asynchrones). Ainsi, tous les messages des parties actions des transitions référencent le composant auquel le message est envoyé. Le nom du receveur est celui porté par l'association liant les classes dans le diagramme de classes.

La signature des méthodes est constituée du nom de la méthode ainsi que du type de chacun de ses paramètres, et la liste des exceptions qu'elle lève (exemple : `foo(int, boolean) throws E1, E2`). La clause `throws` est obligatoire si la méthode peut lever une exception.

Les structures de contrôle que nous représentons dans les parties actions sont la condition, l'itération ainsi que les exceptions. La syntaxe associée à chacun de ces éléments est la suivante :

- `#...` est un appel de message conditionnel, exemple : `#a() #b()` (on exécute soit `a()` soit `b()`) ;
- `{...}` représente l'itération, on exécute un nombre indéterminé de fois ce qui est entre les accolades, exemple `{a(); b();}` ;
- `[...] catch [...]` représente la capture d'exception, exemple `[foo(); z()] catch E1, E2 [bar()]` si `foo()` ou `z()` lève une exception `E1` ou `E2` alors `bar()` sera exécuté ;



(a) Transition traversant la frontière d'un macro-état

(b) Plusieurs transitions de fin sortant d'un macro-état

(c) Incohérence des parties droites et unicité des transitions

FIG. 7.7 – Exemples de SyncClass incorrects.

– `throw` représente l'émission d'exceptions, exemple `throw E1` ;

La grammaire complète de ce langage est donnée annexe B.

7.5.2 Structuration d'un SyncClass, et contraintes d'utilisations

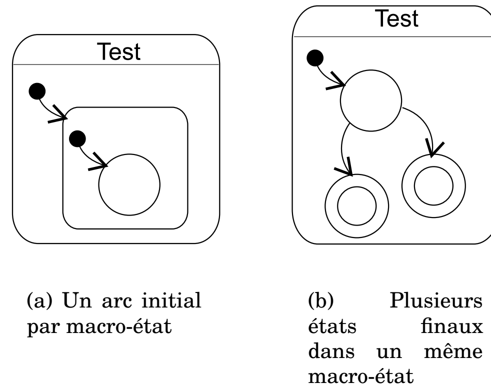
Un SyncClass contient implicitement un macro-état (celui-ci n'est pas représenté graphiquement). Un macro-état peut être divisé par des lignes pointillées qui indiquent le parallélisme entre les macro-états ainsi créés (on parle aussi de constellation). Tout comme un état, un macro-état a des transitions entrantes et sortantes.

Voici une liste des contraintes de construction d'un SyncClass :

– un arc initial est obligatoire dans toute constellation (figure 7.8(a)) ;
– aucune transition ne peut traverser la frontière des états (figure 7.7(a)) ;
– un macro-état peut contenir plusieurs états-finaux (figure 7.8(b)) ;
– une unique transition de fin peut quitter un état (figure 7.7(b)) ;
– deux transitions par message ayant la même signature doivent avoir la même partie action (figure 7.7(c)) ;
– à partir d'un même état ou d'un état englobant, toutes les étiquettes des transitions doivent être uniques (figure 7.7(c)) ;
– les méthodes appelées dans une partie action ne peuvent pas être destinées à la constellation émettrice.

7.5.3 Sémantique intuitive

L'exécution d'un SyncClass consiste en l'exécution de l'unique macro-état dont il est constitué. L'exécution d'un macro-état consiste en l'activation de chacune des constellations le constituant : les arcs initiaux sont empruntés.



(a) Un arc initial par macro-état

(b) Plusieurs états finaux dans un même macro-état

FIG. 7.8 – Exemples de SyncClass corrects.

Le passage d'un état à l'autre se fait lorsqu'une transition est activée. Cette activation se fait lorsque l'appel d'une méthode attendue est reçu. Dans un état donné, les méthodes attendues sont indiquées par les transitions qui sortent directement de l'état courant, ainsi que les transitions qui sortent de l'état englobant l'état courant, et cela de manière récursive jusqu'au niveau du SyncClass. Quand plusieurs états sont actifs (cas de concurrence), l'ensemble des méthodes attendues est l'ensemble des méthodes accessibles (calculé sur le modèle précédent) à partir de chacun des états actifs. L'ensemble des exceptions traitées sont calculées sur le même modèle. Si un appel non attendu est reçu, il est considéré comme une erreur.

Si dans deux constellations la même méthode ou la même exception peut être reçue, alors les deux constellations évoluent.

La traversée de la transition n'est effective qu'une fois toute la partie action exécutée. L'appel de message par défaut est synchrone, seul l'appel de message asynchrone n'est pas bloquant (il est représenté par un « ˆ » à la place du point dans les appels de méthode).

Quand un état final est activé, si une transition de fin sort de cet état, elle est alors automatiquement empruntée. Une transition sortant d'un macro-état n'est empruntée que lorsque toutes les constellations qui le composent sont dans un état final.

La sémantique formelle des SyncClass ne sera pas donnée puisqu'elle correspond à celle du langage synchrone Esterel. Cependant une comparaison entre ces deux modèles est présentée section 7.7.

7.5.4 Intégration des SyncClass à UML

A plusieurs reprises nous avons indiqué que nous utilisions des informations contenues dans des diagrammes de classes ou de composants dans nos SyncClass. Nous montrons donc maintenant l'intégration des SyncClass dans UML. Nous rappelons que nous n'avons pas retenu les StateCharts car ils n'étaient pas adaptés pour représenter le protocole d'inter-dépendance dans le point de vue client et car ils nous aurait fallu

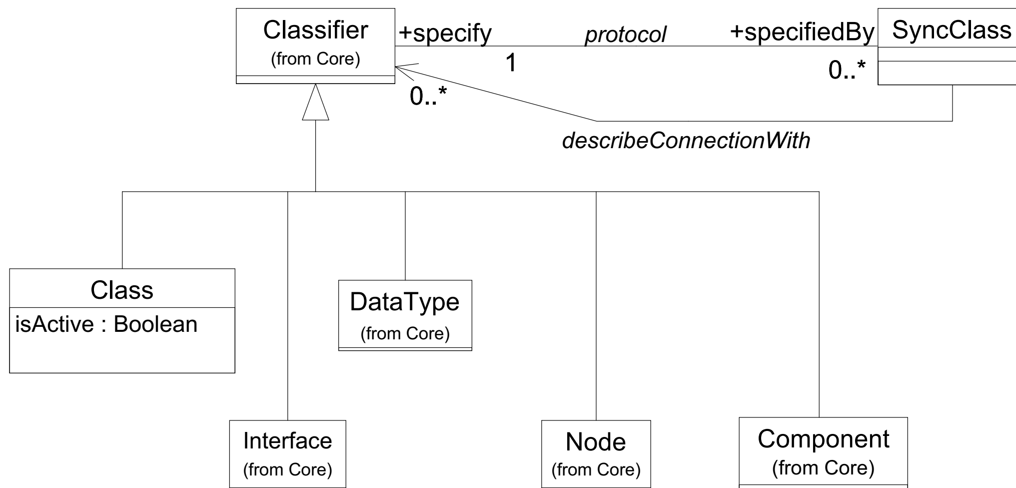


FIG. 7.9 – Liaison SyncClass / UML. À l'exception de la classe SyncClass, la hiérarchie présentée est extraite d'UML.

soit rédifinir une sémantique formelle des constructions que nous aurions retenue ou alors privilégier une sémantique existante (sous-section 5.3.3).

Malgré cela, étant donné que les SyncClass ressemblent graphiquement aux StateCharts, nous avons essayé de remplacer ceux-ci par les SyncClass. Cependant, devant la différence conceptuelle entre les deux modèles, ainsi que des différences fondamentales dans la syntaxe abstraite (qui auraient nécessité la refonte complète des StateCharts pour les transformer en SyncClass) nous avons préféré ajouter les SyncClass au méta-modèle UML sans toucher aux StateCharts. Un exemple de cette différence est l'encapsulation stricte des macros-états des SyncClass d'où aucune transition ne peut traverser les frontières entre états. Afin de ne pas surcharger le document, nous avons mis la description complète de la structure des SyncClass et de ses contraintes OCL dans l'annexe C. Nous ne décrivons ici que la connexion des SyncClass avec le méta-modèle UML existant.

Puisque le but de la notation est de décrire le protocole d'utilisation et de connexion entre des classes, des composants... et cela de manière complètement intégrée à UML, nous avons connecté les SyncClass au niveau de la classe `Classifier` qui est la super-classe directe de `Class` et `Component` (figure 7.9). Bien que `Classifier` soit aussi la super-classe de `UseCase`, `Interface` ou encore `DataType`, nous avons par souci de factorisation intégré les SyncClass à ce niveau puisqu'il n'existait pas dans la hiérarchie UML de super-classes communes aux éléments qui nous intéressaient.

Ainsi, l'association `protocol` relie un SyncClass et l'élément qu'il spécifie, alors que l'association `describeConnectionWith` référence les éléments dont les méthodes sont utilisées dans les parties actions d'un SyncClass. Ces relations reviennent à ajouter le concept d'une interface de sortie aux `Classifier` ainsi qu'à en spécifier le protocole.

Comme nous l'indiquions en introduction, les SyncClass sont inspirés des SyncCharts. Les sections suivantes comparent le modèle objet et le modèle synchrone et détaillent la transformation d'un syncClass en syncCharts.

7.6 Projection modèle objet / modèle synchrone

Le but de cette section n'est pas de décrire un système hybride mêlant langages synchrones et langages à objets comme cela peut être présenté dans [ABP+97], ou encore une approche où les systèmes réactifs sont utilisés pour générer toute la mécanique d'un programme. Ici, le but est de montrer la projection que nous faisons du modèle objet dans le modèle synchrone pour pouvoir tirer profit de ce dernier.

« Qu'un objet soit doué de capacités *réactives* de par son comportement est une propriété fondamentale, indispensable pour lui donner l'individualité, la personnalité, qui vont créer chez le programmeur l'illusion d'existence concrète qui est le ressort de toute l'approche objet. » [DEMN98].

« C'est en dotant nos entités informatiques d'une capacité de réaction que nous les faisons exister : il fonctionne donc il existe. » [DEMN98]

Ces citations extraites de [DEMN98, p. 23] soulignent l'approche considérée dans ce document : voir un objet comme une entité réactive [LM95]. Ici nous poussons simplement l'idée dans la phase de conception et de réutilisation afin de montrer certaines propriétés d'un système.

Avant de rentrer dans les détails de la projection, nous signalons que cette étude est limitée aux langages de classes. Nous avons choisi ceux-ci car ils sont les plus utilisés pour l'implémentation de frameworks.

7.6.1 Comparaison structurelle et comportementale des modèles

Pour commencer nous avons choisi d'établir qu'une classe se projette en un module (un SyncClass). Ce choix est motivé par le fait que ces deux constructions structurent un ensemble de code et ne représentent généralement une exécution qu'après une opération spécifique. Dans les langages à objets, cette opération qui crée une exécution d'une classe s'appelle instanciation. Une instance est créée. L'exécution d'un module s'opère par l'appel à l'instruction `run` qui engendre un processus (on parlera de démarrage). Il existe une légère différence entre le démarrage et l'instanciation : contrairement au démarrage du module qui « instancie » et active le module, une fois l'instance créée celle-ci ne sera active que sur réception d'un message. Cette différence devrait se réduire avec Esterel v7 qui intégrera le concept de module instanciable. De même qu'il est possible de créer plusieurs instances, plusieurs processus représentant le même module peuvent s'exécuter simultanément.

Une différence existe entre la durée de vie des instances et des modules. Par analogie, les modules se comportent plus comme des procédures : ils sont créés puis activés (de la mémoire est allouée et l'exécution se déroule) et ils restent actifs jusqu'à ce qu'une instruction les termine (fin du traitement et désallocation mémoire), alors que les instances sont créées (allocation de mémoire), activées sur un appel de méthode, et cela jusqu'à une éventuelle désallocation explicite ou implicite.

Structure	
Classe	Module
Méthode	Signal d'entrée Signal de sortie
Exécution	
Instance	Processus
Appel d'une méthode	Émission d'un signal
Réception appel de méthode	Réception d'un signal

TAB. 7.1 – Tableau de synthèse sur la correspondance modèle objet / modèle synchrone.

Si l'on regarde les détails des deux structures, on constate que les deux modèles sont munis d'entrées et de sorties. En effet la classe a des méthodes qui servent à indiquer le comportement que l'on veut activer de l'instance, et le module des signaux. La différence entre un module et une classe étant qu'une classe n'a pas d'interface de sortie indiquant les classes avec lesquelles elle communique.

Un module décrit le fonctionnement attendu d'une de ses exécutions ainsi que les signaux que celui-ci émet et reçoit. Quant à elle, la classe décrit le comportement et l'interface de ses instances. Les signaux d'entrée du module peuvent donc être associés à l'interface d'une classe, c'est-à-dire aux méthodes.

A l'exécution l'occurrence d'un signal dans le système correspond à l'appel d'une méthode. La différence fondamentale entre un signal et une méthode est la manière dont ils sont émis dans le système. Une méthode a un receveur désigné alors que le signal est envoyé dans tout le système, et tous les processus qui sont en mesure de l'utiliser réagissent. On a donc d'un côté un modèle à diffusion (*broadcast*) et de l'autre un modèle de communication point à point. Le passage d'un mode de communication à l'autre est effectué en préfixant les messages du nom de leur receveur.

7.6.2 Limites de la projection

La représentation complète d'un programme objet dans le modèle synchrone n'est pas envisageable si l'on ne peut pas en connaître statiquement le nombre d'instances. Cela rendrait l'utilisation des vérificateurs de modèles impossible, car ces derniers raisonnent sur un espace d'états finis, que la création et la suppression dynamique d'instances (et donc d'états) rend incalculable [LP99]. Ainsi dans un système où chaque composant est représenté il n'y a pas de problème puisque la configuration est statiquement connue. Cependant dans le contexte d'un framework, ou tout simplement d'une application écrite dans un langage de classes, il est alors nécessaire de décrire la configuration du système. Cette phase est connue dans les ADLs comme une phase d'instanciation (section 5.4). Les détails concernant la création de cette configuration sont donnés dans le chapitre suivant grâce à l'outil configurateur (sous-section 8.2.3).

En résumé, le protocole d'une classe est décrit par un SyncClass transformé en SyncCharts qui se compile en un module Esterel. La figure 7.10 rappelle brièvement la projection alors que le tableau 7.1 met en correspondance les concepts présentés. Nous

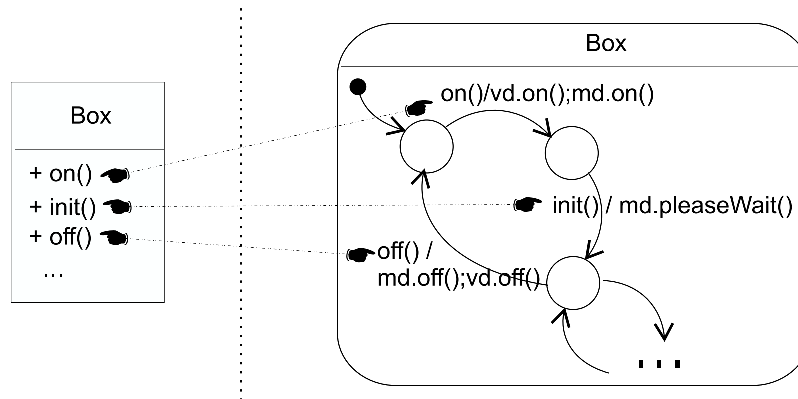


FIG. 7.10 – Correspondance entre le concept de classe et de SyncClass.

tenons à souligner que nous cherchons uniquement à trouver une représentation de la partie comportementale de la sémantique du modèle objet dans le modèle synchrone.

7.7 Transformation d'un syncClass en syncCharts

Les différences entre le modèle synchrone et le modèle objet identifiées il nous faut donc, en fonction de l'extension que constitue les SyncClass par rapport aux SyncCharts, projeter les SyncClass dans le modèle synchrone afin de pouvoir tirer profit de ce dernier. Dans un souci de lisibilité et de simplicité, ces projections sont effectuées vers les SyncCharts. En effet, nous rappelons qu'un SyncCharts est équivalent à un programme Esterel. Cette section décrit ces transformations. Il est à noter que le résultat des transformations évoquées ci-après ne sont jamais visibles de l'utilisateur.

Note : Afin de rapidement reconnaître un syncClass d'un syncCharts, on remarquera que les noms des étiquettes sont suffixés de paranthèses[1] dans les syncClass alors que ce n'est pas le cas dans les syncCharts.

7.7.1 Corps d'une méthode

Nous avons précédemment établi que l'appel d'une méthode correspond à l'émission d'un signal. Cependant, si l'on regarde en détail une différence existe entre l'émission des signaux et des méthodes. Les signaux émis en parties droites des SyncCharts sont émis simultanément (ce qui n'est pas le cas pour les appels fait par une méthode) car il n'y a pas moyen d'exprimer le séquencement des envois de signaux. Cela est une conséquence directe de l'hypothèse de synchronisme qui suppose la réaction instantanée à un signal. Cette hypothèse est en contradiction avec l'exécution du corps d'une méthode dont les instructions ne sont pas exécutées simultanément mais séquentiellement et où la durée d'exécution d'une méthode n'est pas nulle.

[1]Sauf les étiquettes représentant des exceptions.

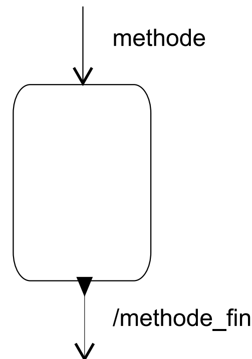


FIG. 7.11 – SyncCharts résultant de la transformation d'un corps de méthode.

Afin de pallier cette incompatibilité nous avons choisi de déplier le corps des méthodes en autant d'envois de messages que nécessaires et d'introduire des signaux de synchronisation. Ainsi le corps d'une méthode est matérialisé dans un SyncCharts, par un macro-état contenant le corps de celle-ci (la représentation de la partie action en SyncCharts) comme le montre la figure 7.11 pour la transformation de `methode() / corps` en un SyncCharts équivalent. La sortie de ce macro-état se fait lorsque son exécution est terminée. La transition vers un autre état se fera en utilisant une terminaison normale, et la fin de l'exécution de la méthode sera indiquée par l'émission d'un signal spécifiquement créé. Cette information est nécessaire pour pouvoir synchroniser les différents composants et éviter de perdre des signaux. L'exemple de la figure 7.12 montre le problème que l'on rencontrerait si l'on produisait des SyncCharts sans synchronisation ni dépliement. En effet, si l'on regarde précisément le système on voit qu'à l'instant où `b` est émis, `d` et `c` le sont aussi, entraînant la perte de `c` qui n'est pas prêt à être reçu. Bien sûr l'ajout de transitions vides ou la modification de la nature de certaines transitions permettraient de synchroniser le système, cependant cela altèrerait la possibilité de composer notre système car afin de calculer correctement ces transitions vides, il faudrait faire une analyse globale du système et de l'enchevêtrement des appels.

7.7.2 Envoi de message

Comme nous l'indiquions précédemment (sous-section 7.6.1) la communication est fondamentalement différente entre le modèle objet et le modèle synchrone. Ainsi, pour éviter les problèmes que pose le modèle synchrone avec sa communication par diffusion (*broadcast*) nous avons choisi de préfixer tous les envois de message du nom de leur receveur. Cette technique est celle utilisée dans les langages à objets où tout message est envoyé à un objet identifié. Afin de ne pas avoir à recréer un nom spécifique, la connexion des classes utilise le nom des relations dans les diagrammes de classes, ce qui facilite la lecture des points de vue.

Le modèle d'émission simultanée des signaux des SyncCharts ne convient pas à la description du corps d'une méthode dont l'exécution des différents constituants n'est

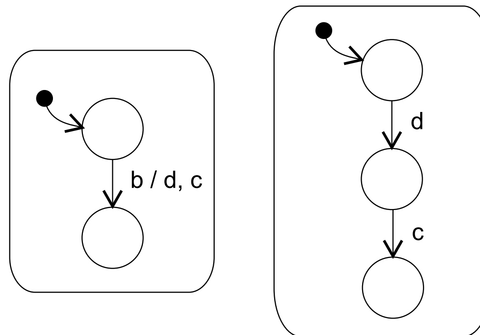


FIG. 7.12 – SyncCharts présentant le problème posé par la séquentialité sans signal de fin de méthode.

pas instantanée et où l'on doit attendre la terminaison d'un message avant d'exécuter le suivant. Bien que la plupart du temps il soit nécessaire d'avoir un tel système d'appels synchrones, il est aussi nécessaire de pouvoir indiquer des envois de messages asynchrones.

Messages synchrones

Les messages synchrones (en terme d'objet) sont utilisés pour indiquer un appel de méthode dont on attend la terminaison pour passer à la suite. Ainsi dans une transition `methode() / x.a()` l'appel à `methode` sera terminé une fois l'exécution de `a()` terminée.

Les appels de messages synchrones sont représentés en SyncCharts par un macro-état contenant l'envoi d'un signal représentant l'appel à la méthode, et la mise en attente de la réception du signal de fin de méthode (figure 7.13).

Messages asynchrones

Les messages asynchrones sont utilisés pour indiquer un appel de méthode dont on n'attend pas la terminaison pour continuer (« *a reply is optionnal and will be ignored* » [OMG]). Pour cela nous avons ajouté un nouvel élément de syntaxe : « ^ ». Il est utilisé en remplacement du « . » dans un appel de méthode. Ainsi dans une transition `methode() / x^a()` la `methode` sera terminée dès que l'appel `x^a()` aura été effectué.

La figure 7.14 montre un tel exemple pour l'appel asynchrone de la méthode `a`.

7.7.3 Séquence

Afin d'indiquer une séquence d'appels nous ne pouvons pas utiliser la virgule qui représente la simultanéité des signaux dans les SyncCharts, puisque celle-ci est utilisée comme séparateur des signaux. Nous avons donc introduit un opérateur de séquentialité, représenté par un « ; ». Pour indiquer qu'une méthode fait séquentiellement plusieurs appels de méthode on écrit `methode() / b(); c(); d();`. Cela signifie que

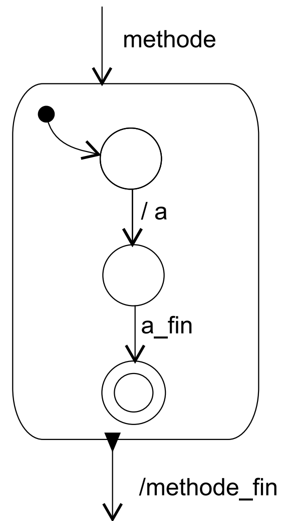


FIG. 7.13 – SyncChart représentant l'appel synchrone d'une méthode.

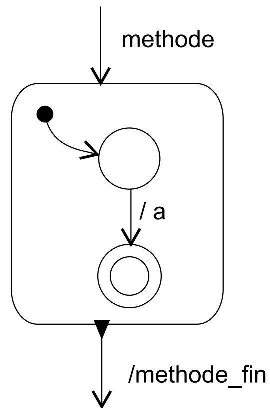


FIG. 7.14 – SyncChart représentant l'appel asynchrone d'une méthode.

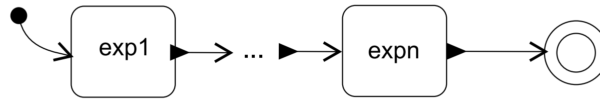


FIG. 7.15 – Transformation d'une expression séquentielle en un SyncCharts.

Java	SyncClass
if (x.test()) { x.a() ; x.b() ; } else { x.c()}	x.test() ; #(x.a() ; x.b()) #x.c()
if (a==0 && x.test()) {x.a()}	#x.test() ; #x.a()

TAB. 7.2 – Différents styles de conditionnels.

l'on ne pourra arriver dans l'état de fin de la méthode `a()` qu'une fois tous les messages exécutés jusqu'à leur fin respective. Dans l'exemple, la fin de `methode()` se produira une fois la fin de `b()` puis la fin de `c()` et la fin de `d()` survenue. Cet opérateur se compose avec les deux types d'envoi de messages.

La notion de séquentialité n'a pas de représentation immédiate dans un SyncCharts. Cependant elle n'ajoute pas de nouveaux macro-états mais joue un rôle de connexion entre les différents macro-états générés par les transformations (figure 7.15).

7.7.4 Structure de contrôle

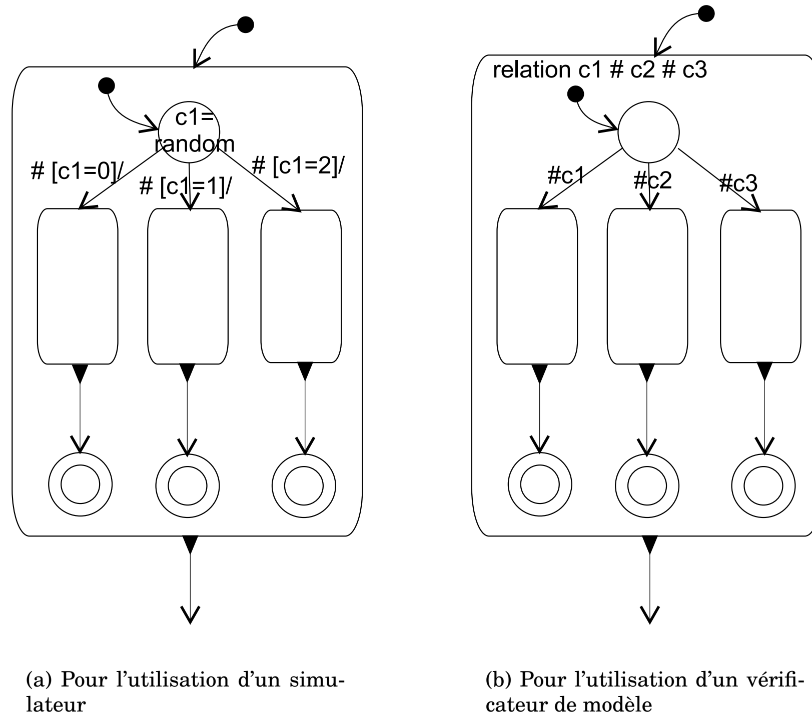
Comme nous cherchons à représenter l'utilisation d'une classe, il est donc nécessaire de pouvoir exprimer des conditions, des itérations ou des exceptions dans les parties actions des transitions.

Condition

Afin de représenter l'émission conditionnelle de messages, nous avons étendu la syntaxe des parties droites des transitions en préfixant les expressions conditionnées par un symbole `#`. On l'utilise par exemple de la manière suivante : `methode() / #a() #b() #c()` pour indiquer que l'appel à `a`, `b` ou `c` est conditionnel et exclusif.

Cette notation permet de représenter les conditions et les conditions / énumérations. Par exemple le code Java suivant : `methode() { if(...) {x.a(); x.b(); } else {x.c(); }}` se représente par `methode() / #(x.a(); x.b()) #(x.c())`. On notera que l'expression de la conditionnelle est ignorée. D'autres cas de conditionnelles peuvent être exprimés comme le montre la figure 7.2.

Le modèle synchrone et les SyncCharts n'autorisent pas l'émission conditionnelle de signaux. Cela rendrait le système non-déterministe, et donc le comportement impossible à modéliser dans les formalismes sous-jacents. Nous ne pourrions plus utiliser les vérificateurs de modèles ou les simulateurs. La transformation que nous faisons doit donc « déterminiser » l'expression de la conditionnelle selon l'utilisation que l'on va faire du SyncCharts résultant de la transformation.



(a) Pour l'utilisation d'un simulateur

(b) Pour l'utilisation d'un vérificateur de modèle

FIG. 7.16 – SyncCharts résultant de la transformation d'une conditionnelle.

Pour la simulation, un appel à une méthode effectuant un tirage aléatoire est utilisé. La valeur de retour de cette méthode est considérée comme un signal valué qui détermine la branche de la conditionnelle à emprunter. Cet exemple est montré figure 7.16(a).

Pour la vérification, afin que toutes les possibilités soient explorées, l'utilisation de signaux valués n'est pas possible. En effet, le vérificateur de modèles ne sait pas gérer les valeurs. Ainsi les signaux valués sont remplacés par des signaux non valués entre lesquels une relation d'exclusion est exprimée. Figure 7.16(b) la `relation` indique que `c1`, `c2` et `c3` sont exclusifs, ils ne peuvent pas être présents simultanément.

Lorsque la condition ne contient qu'un seul élément, la condition contient une transition vide vers un état final.

Itération

La notation retenue pour les itérations est le « `{ ... }` ». L'expression se situant entre les accolades est exécutée un nombre de fois indéterminé mais fini entre 0 et n. L'itération pose donc le même problème de non-déterminisme que la condition puisque nous ne voulons pas montrer le fonctionnement interne des composants. Là encore,

Java	SyncClass
while(a.b()) { x.c() }	a.b() ; { x.c() ; a.b() }
for (a.a() ; b.b() ; c.c()) { d.d() ; }	a.a() ;b.b() ;{d.d() ;c.c() ;b.b()}
do { a.a() ; } while (b.b()) ;	a.a() ;b.b() ;{a.a() ;b.b()}

TAB. 7.3 – Représentations de boucles en SyncClass.

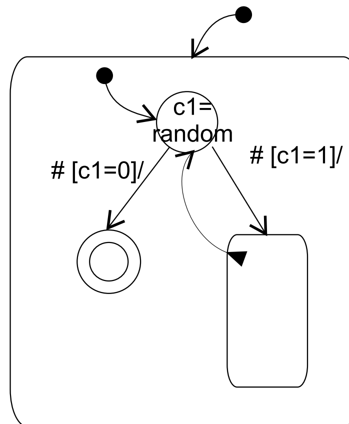


FIG. 7.17 – SyncCharts résultant de la transformation d'une itération.

la représentation en SyncCharts utilise un signal valué choisi aléatoirement ou une relation entre signaux.

Cette notation permet de représenter des boucles `while`. En préfixant la boucle du comportement désiré, on peut reproduire un comportement équivalent aux boucles `for`, et `repeat-until` comme le montre le tableau 7.3.

Comme nous l'avons déjà remarqué les expressions booléennes utilisées au sein des boucles et conditionnelles n'apparaissent jamais. Nous avons fait ce choix pour ne pas casser l'encapsulation. En effet, montrer ces expressions reviendrait à faire apparaître une partie du corps de la méthode. Or lorsqu'on utilise une méthode, on l'utilise pour la globalité de son comportement, et peu importe que l'on passe dans telle ou telle partie du code de celle-ci. Indiquer l'expression booléenne n'apporterait rien puisqu'elle n'est pas directement liée au contexte d'utilisation du composant (les valeurs mises en jeu dans la classe font appel à l'état interne de l'objet, et non pas à l'état de l'objet qui l'a appelé) sauf si bien sûr cette expression contient des appels à d'autres composants.

Exceptions

Devant l'importance qu'ont les exceptions dans la description du fonctionnement des systèmes et leur présence explicite en sortie des méthodes, nous avons ajouté le support des exceptions dans les parties actions. La syntaxe est la suivante : `methode() / [ expr1 ] catch Exception [expr2]`. L'expression précédente équivaut au code Java présenté figure 7.18.

```
methode() {
   try {
     expr1
   } catch(EFoo) {
     expr2
   }
}
```

FIG. 7.18 – Code java d'une exception.

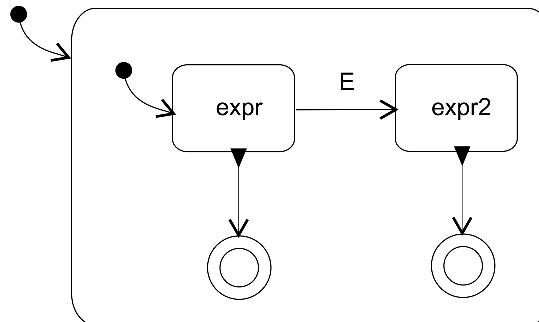


FIG. 7.19 – SyncCharts résultant de la transformation d'une exception.

L'idée sous-jacente à la transformation associée est que le bloc protégé par le `try - catch` peut être interrompu à tout instant durant son exécution. Ce bloc est donc placé dans un macro-état et l'exception sera modélisée par une préemption faible. Le résultat de la transformation est montré figure 7.19. Nous avons retenu la préemption faible car le signal est levé de l'intérieur du bloc préempté et peut ainsi être attrapé. L'utilisation d'une préemption forte aurait causé un cycle de causalité.

Les exceptions ont deux aspects : un premier qui est la gestion de l'exception (ce que nous venons de voir) et un second qui est l'émission d'une exception. L'émission d'une exception est indiquée par le mot-clef `throw`. Elle a pour effet d'arrêter l'exécution de la méthode à l'endroit où elle se situe. Il est donc inutile que celle-ci soit suivie d'une autre instruction puisqu'elle ne pourra jamais être exécutée (c'est un problème d'analyse de flot). L'émission de l'exception ne pouvant pas être effectuée autrement que par un envoi de message, il faut donc que la transformation gère l'arrêt de l'exécution de la méthode. Pour cela, le corps de la méthode est mis dans un macro-état dont on sort sur réception de l'exception (figure 7.20). Le `throw` est transformé en une émission de signal.

Les exceptions sont aussi présentées dans la signature des méthodes (partie gauche de la transition). En effet, à la manière de Java nous pensons qu'il est important d'indiquer dans la signature des méthodes si une exception peut en émaner, c'est-à-dire qu'elle n'est pas gérée par l'utilisateur. Ainsi dans la signature des méthodes elle est représentée par une clause `throws`. Le SyncCharts équivalent est présenté figure 7.20.

Toutes les structures présentées précédemment peuvent être composées. La grammaire du langage des parties droites de transitions est donnée annexe B.

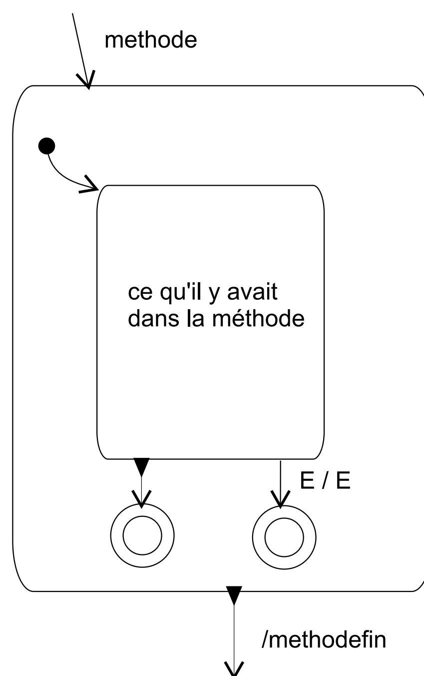


FIG. 7.20 – SyncCharts résultant de la transformation d'un throw.

7.7.5 Appel de méthode levant des exceptions

Si l'on examine en détail les exceptions, on constate que celles-ci, contrairement aux appels de méthodes ne sont pas destinées à un composant particulier. Le destinataire n'est connu qu'à l'exécution grâce au flot des appels de messages. C'est ce modèle d'exception qui est généralement utilisé dans les langages objets, et que nous utilisons dans les SyncClass. Cependant, la transformation d'un SyncClass en SyncCharts telle que nous l'avons définie jusqu'alors n'est pas en accord avec cette sémantique. Ainsi, si deux composants utilisent une même instance, alors si cette instance lève une exception les deux composants sont avertis de la présence de l'exception (le signal représentant l'exception est diffusé). Pour éviter ce problème on effectue une transformation supplémentaire sur tous les appels de méthode invoquant une méthode pouvant lever une exception. Cette transformation qui s'apparente à une analyse de flot, permet d'éviter les problèmes précédents et surtout que les exceptions soient retournées à la bonne méthode.

Le principe de la transformation est de répertorier pour un SyncClass donné, toutes les méthodes qui sont appelées et qui peuvent lever une exception, et de remplacer dans les contextes d'appel la capture de l'exception initiale par une exception spécifique à ce contexte. Cette exception spécifique est levée une fois l'exception initiale attrapée et le contexte cible déterminé. Le contexte cible est obtenu en gardant trace du contexte d'appel (simplement appelé contexte par la suite). Ce contexte représente l'appel à une méthode levant une exception dans un niveau d'imbrication donné (ce niveau d'imbrication est calculé sur l'imbrication des blocs de gestion d'exceptions (blocs `try catch`). Il permet ainsi de connaître le `try catch` dans lequel l'exception doit être prise en compte. Chaque contexte est identifié de manière unique selon deux informations, la méthode qui a effectué l'appel et le bloc `try catch` de cette méthode dans lequel a été effectué cet appel. Ce contexte est connu en émettant un signal spécifique au même instant que le signal d'appel de méthode. Cette transformation est présentée figure 7.21 dans le cadre de l'appel d'une méthode `cMethod1` qui lève l'exception `E`. Le signal de contexte est le signal `foo_cMethod1_ctx1`. L'exception `E` est changée en l'exception `E_ctx1`, une fois capturée cette exception est transmise à son contexte englobant par le signal `E_ctx0`.

7.7.6 Appel à super

L'appel des méthodes d'une super-classe est fréquemment utilisé afin de réutiliser le comportement d'une classe dont on hérite. Cependant ce concept et son interprétation dans la représentation d'un protocole n'est pas aussi claire que son comportement dans le contexte d'un langage.

Ainsi, si l'on considère l'appel à super comme un appel de méthode normal (un appel sur un autre composant), il faut donc que la super-classe soit prête à recevoir cet appel de message (c'est-à-dire que son protocole soit au bon endroit). Pour cela, il faut que le protocole d'une classe et de ses super-classes évoluent simultanément. Dans ce cas, l'appel à la super-méthode sera autorisé par le protocole de la super-classe. La super-méthode est exécutée dans son contexte. Cette technique nécessite pour être mise en

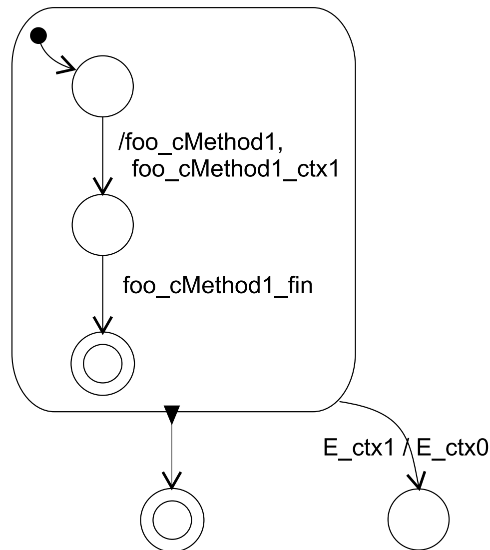


FIG. 7.21 – SyncCharts montrant le marquage du contexte dans les appels de méthodes.

œuvre une instrumentation particulière des SyncClass, mais de plus requiert aussi que le protocole de la sous-classe soit substituable à celui de la super-classe.

Une autre possibilité, qui est celle que nous retiendrons pour la suite de ce manuscript, consiste à remplacer l'appel à super par le corps de la méthode appelée. Cela évite le problème lié à la connaissance du protocole de la super-classe.

7.7.7 Résultat d'un SyncClass transformé

La transformation d'un SyncClass en SyncCharts engendre de nombreux signaux dont nous dressons un rapide bilan :

- une méthode appartenant à l'interface d'entrée d'un SyncClass donne un signal en entrée et un en sortie. Exemple : `methode()` donne `methode` et `methode_fin` ;
- un appel de méthode (interface de sortie) donne une émission de signal (signal de sortie), et l'attente du signal de fin (sauf cas d'appel asynchrone). Cet appel est préfixé du composant à qui il est destiné : `foo.cMethod1()` donne `foo_cMethod1` et `foo_cMethod1_fin` ;
- un appel à une méthode levant une exception génère un signal de contexte par appel (signal en sortie) ainsi qu'un signal spécifique à chaque exception levée dans un contexte donné (un signal d'entrée). Par exemple, pour un appel à une méthode `foo.cMethod1()` qui lève une exception `E`, on génère `foo_cMethod1_ctx1`, et `E_ctx1` ;
- une itération génère un signal qui permet d'indiquer si l'on décide de sortir ou non de l'itération ;
- une condition génère autant de signaux qu'il y a de branchements possibles.

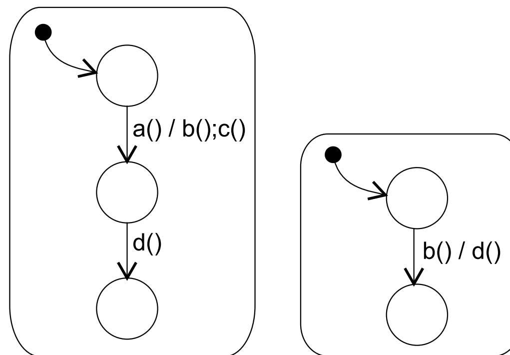


FIG. 7.22 – SyncClass montrant un cas de réentrance avec blocage.

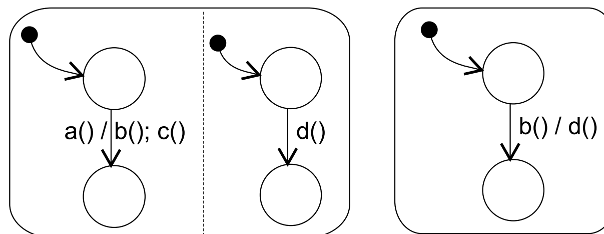


FIG. 7.23 – SyncClass montrant un cas de réentrance sans blocage.

Il est à noter que la transformation d'un SyncClass en SyncCharts se fait en un seul parcours du SyncClass.

7.7.8 Réentrance

Dans notre contexte, on identifie par réentrance le fait qu'une méthode de l'interface publique d'un composant fasse appel à une autre méthode de l'interface publique du même composant de manière directe ou indirecte. Selon les cas, la réentrance peut être autorisée ou non. En effet la première condition pour que la réentrance soit possible est que le protocole décrit par le SyncClass soit respecté. Ainsi, l'exemple de la figure 7.22 pose problème car l'appel à `d()` se produit avant que la méthode `a()` ne soit terminée, et donc le message n'est pas prêt à être reçu. Si l'on désire garder la même séquence d'appels il faudrait alors que `d()` soit dans un état exécuté en parallèle (figure 7.23).

Par ces deux exemples on voit les problèmes que pose la réentrance dans un contexte fini (connu statiquement) de réentrées. La récursion étant aussi une réentrée en nombre fini mais non connu statiquement, nous ne sommes pas en mesure de la représenter car en effet il faudrait prévoir autant d'états en parallèle qu'il y a d'appels récursifs possibles à l'exécution. Les cas de réentrance causant des blocages sont détectés par les vérificateurs de modèle. Ces problèmes viennent de la sémantique naturelle de l'envoi de message que nous avons choisi, et qui indique qu'un envoi de message n'est pas terminé tant que l'exécution de son corps ne l'est pas.

7.7.9 Résumé des différences SyncCharts / SyncClass

Les deux notations et la transformation d'un SyncClass en SyncCharts ayant été présentées, cette sous-section dresse un bilan des extensions et restrictions syntaxiques et sémantiques des SyncClass par rapport aux SyncCharts.

Comparaison syntaxique

Syntaxiquement, les SyncClass gardent les principaux éléments de représentation graphique des SyncCharts tels que les transitions, les états, les macro-états et les états finaux. Le seul élément supprimé est la suspension.

La principale différence se situe au niveau des transitions où les étiquettes associées ont été modifiées ou supprimées. Ainsi la partie gauche des transitions des SyncClass ne peut pas être conditionnée, ni marquée d'une priorité, et elle ne référence qu'un seul signal. De plus, la terminaison normale n'est pas étiquetée et la partie droite de la préemption forte, qui est utilisée pour représenter les exceptions, n'est pas étiquetée. On ne retrouve pas l'opérateur de simultanéité « , ». Cependant il est toujours utilisé comme résultat de certaines transformations.

Nous ne permettons pas non plus l'expression de relations entre les signaux des méthodes, ni la déclaration de signaux locaux.

Comparaison sémantique

Sémantiquement, ce sont aussi les transitions qui font l'objet d'une modification puisqu'elles ne sont plus traversées instantanément (chaque transition est développée en un ensemble de transitions).

Bien que les SyncClass utilisent la notation des préemptions fortes, il ne s'agit là que d'une utilisation purement graphique puisque ces préemptions sont transformées en des préemptions faibles dans le syncCharts. Cela évite les problèmes de cycle de causalité.

Une autre différence liée aux transitions est le traitement des signaux non attendus. Les SyncClass ne décrivant que le protocole « correct » du composant, tout message dont l'occurrence se produit à un moment où il n'est pas attendu est considéré comme une erreur. Cela n'est pas le cas dans les SyncCharts où l'occurrence d'un signal inattendu ne cause aucune réaction dans le système, le signal est ignoré. Implémenter cette sémantique dans les SyncCharts revient à ajouter un état puits (état poubelle / état d'erreur) dans lequel toutes les transitions non autorisées (qui doivent causer une erreur dans les SyncClass) aboutissent (figure 7.24).

7.8 Conclusion sur les points de vue comportementaux

Ce chapitre a présenté le concept de point de vue comportemental ainsi qu'une notation adaptée à leur représentation : les SyncClass. Un point de vue comportemental est une manière d'envisager le protocole d'utilisation et d'inter-dépendance d'un composant : le point de vue client décrit comment utiliser l'interface d'entrée d'un composant

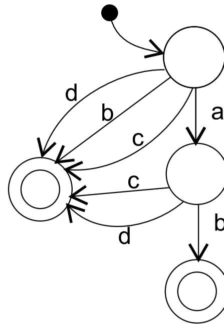


FIG. 7.24 – Etats puits dans un SyncCharts.

et comment se comporte son interface de sortie ; le point de vue de composition décrit le comportement d'une partie de l'interface de sortie d'un composant.

Cette approche, qui consiste à montrer à partir du protocole d'un composant ses interactions avec les autres, s'inscrit dans la direction de recherche énoncée par David Harel à UML 2000 [Har00] : « *It is also a prerequisite to a thorough investigation of what might be the problem in object-oriented specification : relating inter-object specification to intra-object specification* ». Certes notre approche ne correspond pas exactement à la vision d'Harel qui évoquait l'étude de la correspondance entre les diagrammes de séquences et les diagrammes d'états, cependant nos résultats vont dans le sens d'un rapprochement entre le comportement inter-composant et intra-composant. En effet, en spécifiant la relation inter-composant nous montrons obligatoirement une partie du comportement interne d'un composant puisque le comportement entre les composants est créé par le comportement des composants eux-mêmes. Ici nous agrégeons comportement inter- et intra-objets.

Les SyncClass étant basés sur les SyncCharts et donc le modèle synchrone, nous avons montré le rapprochement entre ce modèle et le modèle objet. Ce rapprochement et les règles de transformations d'un SyncClass en un SyncCharts permettent d'utiliser toute la puissance du modèle synchrone et des outils associés. Cette utilisation est montrée dans le chapitre suivant qui s'attache à présenter l'ensemble des outils relatifs aux SyncClass, ainsi que leur intégration dans le cycle de vie du framework.

Chapitre 8

Utilisation des points de vue comportementaux

Bien que les activités constituant le développement d'un framework soient les mêmes que celles du développement d'applications, on a vu que la documentation qui était fournie à l'utilisateur final requérait une plus grande rigueur.

Ainsi, ce chapitre montre comment cette documentation qui se veut, accessible à un utilisateur, s'intègre tout au long du cycle de vie pour produire un framework documenté.

Ce chapitre se conclut par la présentation d'un environnement dédié aux SyncClass. Cet environnement appelé Co^2 (*Co*mposant et *Co*mposition) fournit un ensemble d'outils d'aide à la conception, spécification et développement de composants.

8.1 Intégration des SyncClass dans le cycle de vie

Bien qu'il ne soit pas nécessaire de documenter l'intégralité des constituants du framework ni même de représenter l'ensemble des points de vue, nous pensons qu'un minimum est requis. Ce minimum est constitué des classes qui seront par exemple utilisées dans les points de variation.

Alors que de nombreuses documentations sont créées une fois le développement terminé, nous proposons que les SyncClass soient intégrés tout au long du cycle de vie du framework. De cela résulte un framework partiellement documenté qui permettra à l'utilisateur de comprendre les relations entre les composants. Ainsi nous décrivons dans cette section le rôle des SyncClass dans les activités du développement du framework et indiquons les différents outils qui pourraient être utilisés.

Nous rappelons que nous ne ciblons aucun cycle de développement logiciel particulier. Nous retenons juste que lors du développement logiciel, que ce soit dans un cycle en cascade, en V, en spirale ou autre, on retrouve les mêmes activités.

8.1.1 Analyse et conception

La spécification d'un composant doit être faite très tôt. Une fois les grandes classes du système déduites des cas d'utilisation (*Use Cases*) nous pensons que les SyncClass pourraient d'emblée être utilisés pour en spécifier le protocole.

Lors de ces activités, les SyncClass viennent en complément des diagrammes de séquences et des diagrammes d'états-transitions. Malgré les informations que ces derniers diagrammes partagent, aucun ne peut remplacer l'autre. Un diagramme de séquences décrit les interactions d'un ensemble de classes dans un scénario donné, un diagramme d'états-transitions l'évolution interne d'une classe, et un SyncClass le protocole d'une classe. En fait, la seule concurrence que l'on peut noter se fait dans le cas particulier de l'utilisation des diagrammes d'états-transitions comme *protocol state machine* (sous-section 5.3.3).

Lors de l'activité de conception détaillée, les SyncClass sont utilisés pour détailler le protocole des classes et ainsi compléter les interfaces de celles-ci. Les SyncClass conçus en phase de conception sont alors mis à jour.

Il est important de noter qu'à partir de cette phase, on pourra commencer à faire des analyses statiques du système pour détecter d'éventuelles inadéquations entre les composants spécifiés à l'aide de SyncClass. Ces analyses font l'objet d'une étude particulière à la section 8.4.

8.1.2 Implémentation

Lors de l'implémentation les SyncClass permettent de générer un squelette des méthodes à partir du comportement décrit dans les parties droites des transitions. Cependant quand bien même la conception a été très détaillée, des différences entre la conception et l'implémentation ne sont pas rares. Généralement des différences mineures entre la conception et l'implémentation ne posent pas de problème, si ce n'est lors de la maintenance. Cependant, comme nous avons décidé de fournir les SyncClass à l'utilisateur du framework, il est donc obligatoire que ces derniers soient cohérents avec l'implémentation.

Dualement, afin d'éviter la corvée de synchronisation des SyncClass avec l'implémentation, nous avons mis au point un outil qui permet de mettre à jour les parties actions des transitions à partir du code (sous-section 8.3.1). Si le protocole du composant a été changé, la mise à jour ne peut pas être automatique et doit être effectuée par le développeur.

8.1.3 Test unitaire

Comme nous l'avons vu section 1.5, le test unitaire est une activité d'autant plus importante que le développeur pourra difficilement effectuer des tests d'intégration couvrant toutes les utilisations possibles. Pour aider dans cette activité, les SyncClass peuvent être utilisés pour générer des jeux de tests fonctionnels et unitaires par composant.

Ces tests sont d'autant plus importants que du point de vue de l'utilisateur, le protocole fourni apparaît comme un gage de bon fonctionnement du composant quand il est respecté.

Cette vision du test peut sembler réductrice quand on la compare à tous les tests possibles [XRK00] associés aux différentes activités du cycle de vie. Cependant nous n'excluons pas les autres techniques de tests. Nous nous focalisons sur celle présentée ici à cause de son importance pour le développeur comme pour l'utilisateur. En outre les SyncClass proposent un moyen simple de les générer.

8.1.4 Test d'intégration

Les SyncClass n'apportent pas une aide spécifique aux tests d'intégration. Cependant, puisque l'intégration consiste à mettre les différents constituants du framework ensemble, on peut utiliser les SyncClass afin de tester l'assemblage de composants ou encore tester l'adéquation de composants deux à deux.

8.1.5 *Packaging* du composant

Une fois le composant réalisé et testé, il ne reste plus qu'à le distribuer ; cependant comme nous défendons l'idée qu'un composant n'est pas seulement un code mais un code *et* sa documentation (incluant le protocole), nous pensons qu'il est nécessaire d'y intégrer la documentation sous une forme appropriée. La finalisation tente de résoudre ce problème de localisation d'une documentation (est-elle dans un fichier, dans quel répertoire, dans quelle armoire et quel volume ? — support inadapté et jamais au bon endroit) en mêlant documentation et composant dans le format le plus approprié à la fois à la forme physique du composant et à sa documentation.

Cette documentation embarquée constitue une méta-donnée du composant accessible à tout moment.

8.1.6 Réutilisation (validation, exploitation)

Une fois le composant livré, son utilisateur pourra se référer à la documentation intégrée afin d'apprendre à l'utiliser (connaître le protocole du composant et son protocole de connexion) et mettre au point une application l'utilisant. Il pourra bénéficier des outils de vérifications dynamique et statique afin de s'assurer que son code fait un usage correct du composant.

Nous avons donc vu que les SyncClass s'intègrent tout au long du cycle de développement des composants d'un framework et constituent, à moindre coût, un moyen simple de documenter le protocole d'utilisation et d'inter-connexion d'un composant. La figure 8.1 résume graphiquement les différentes activités.

8.2 Outils généraux

L'ensemble des outils relatifs aux SyncClass est présenté figure 8.2. Les outils (représentés dans des rectangles) s'organisent autour des SyncClass et des autres données

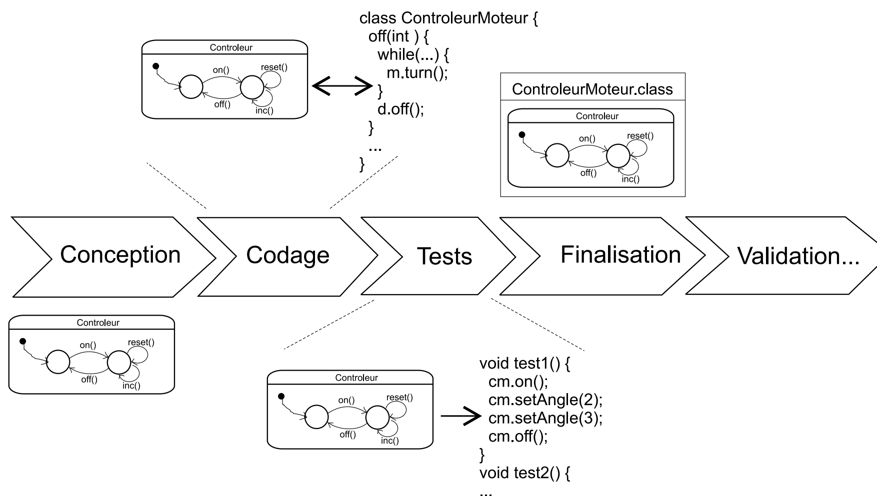


FIG. 8.1 – Les SyncClass dans le cycle de vie.

(représentées dans des ellipses). Ils font l'objet d'une présentation dans cette section et les sections suivantes.

8.2.1 Editeur / navigateur

Le but de cet outil est de permettre aux utilisateurs, qu'ils soient développeurs ou utilisateurs, de manipuler à loisir les SyncClass. Cet outil constitue l'interface principale de Co^2 puisqu'il permet à la fois d'éditer les SyncClass et d'appeler les outils comme les vérificateurs, le simulateur, etc. L'interface, intégrée à Eclipse [OTI01] présente dans un arbre les composants, leur point de vue et le code Java correspondant (figure 8.3).

A l'heure actuelle, à cause d'un retard dans la livraison du framework graphique (GEF) utilisé dans Eclipse, nous n'avons pas encore créé d'éditeur graphique. Malgré tout, les SyncClass peuvent être saisis et sauvegardés dans leur format externe qui est XML.

8.2.2 Finaliseur

Cet outil permet de rassembler en une seule entité le code du composant et sa documentation. Il est utilisé lors du *packaging*. Ainsi, étant donné un code binaire ou source et une documentation à base de SyncClass, cet outil intègre l'un et l'autre dans le format le plus adapté aux deux. Le tableau 8.2.2 montre le format et le lieu d'intégration de la documentation en fonction du type de composant.

Dans l'implémentation existante, seule l'intégration dans les fichiers .class est disponible.

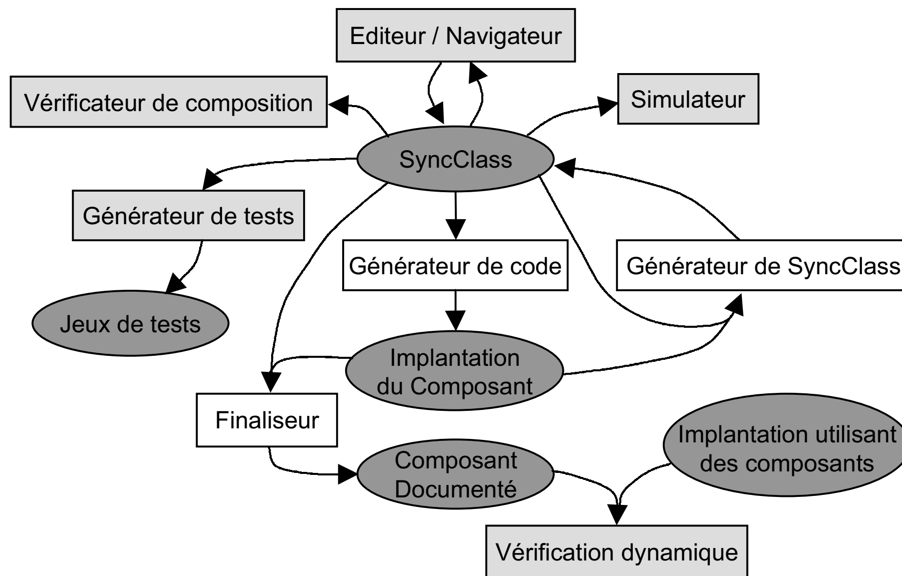


FIG. 8.2 – SyncClass et outils. Les outils dont le rectangle est blanc ne sont utilisés que par le développeur.

Type de composant	Endroit d'intégration	Format d'intégration
.jar	Fichier séparé intégré dans le .jar	XML du SyncClass + format compilé (BLIF)
.class	Dans le fichier .class, partie attributs utilisateurs	Format compilé, les versions graphiques sont dans un fichier séparé
EJB / CCM	Fichier de description	XML et format compilé

TAB. 8.1 – Intégration de SyncClass dans différents types de composants.

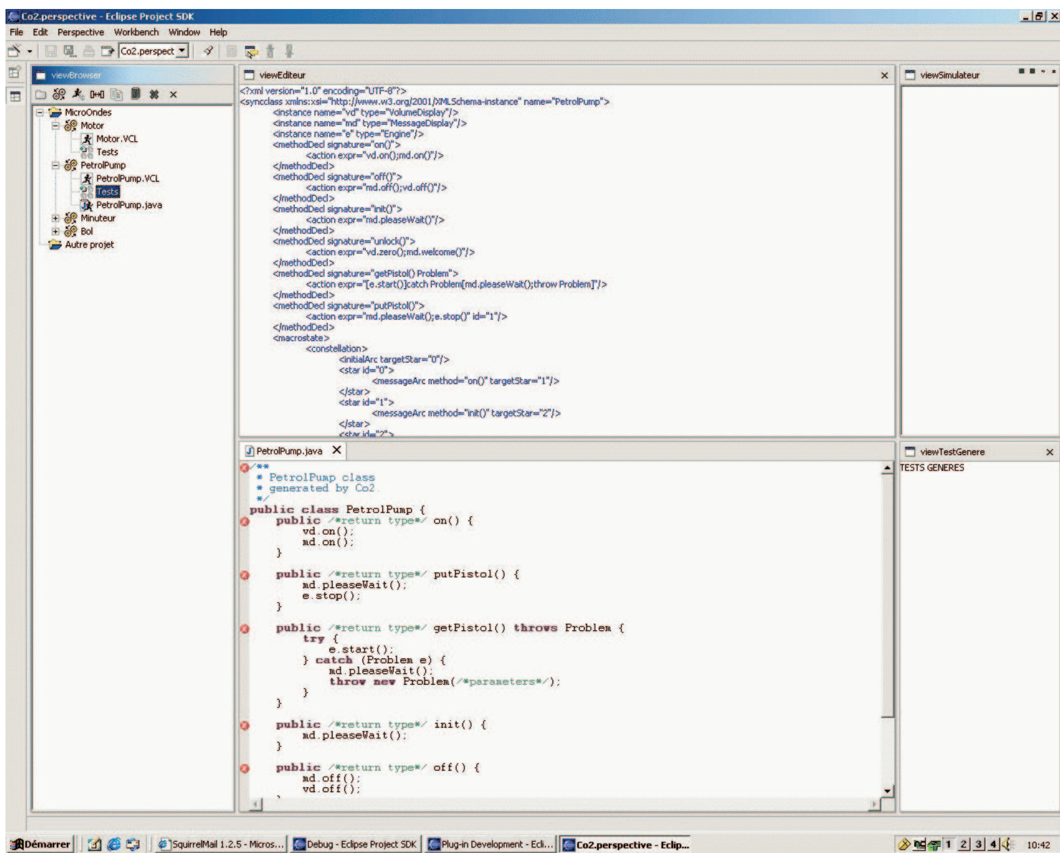


FIG. 8.3 – Éditeur de SyncClass.

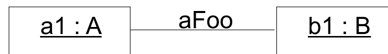


FIG. 8.4 – Exemple de diagramme objets utilisé pour une configuration.

8.2.3 Configurateur

Jusqu'alors nous avons raisonné sur un ensemble de composants dont les connexions étaient connues statiquement et décrites dans les SyncClass par la notation pointée. Ainsi sur l'exemple de la pompe à essence, nous avons un ensemble de composants dont l'organisation est parfaitement connue puisque nous ne gérons qu'une seule instance de chaque élément. Cependant cette unique représentation ne permet pas de gérer l'instanciation multiple de composants comme cela peut se produire dans un framework.

Nous introduisons donc la notion de configuration* (decription de l'organisation d'un système). Une configuration décrit un ensemble d'instances de composants et leurs connexions. Un outil permettant la saisie d'une configuration a été mis en place. Il utilise les diagrammes d'objets d'UML[1] qui représentent l'organisation des instances d'une application. Une configuration connectant deux instances est montré figure 8.4[2].

L'utilisation d'une configuration pose un problème en terme de signaux. En effet, si nous nous contentons de mettre en présence (en parallèle) les SyncCharts résultant de la transformation des SyncClass, aucune communication ne s'effectuera malgré le modèle de diffusion (*broadcast*) d'Esterel. Cela vient de l'inadéquation entre le nom des signaux sortants et des signaux entrants (section 7.7). La figure 8.5 exemplifie une telle situation en présentant les SyncCharts de deux classes connectées par une association nommée `foo`. On voit que le nom des signaux sortants est préfixé du nom de la relation, et que le nom en entrée est simplement le nom de la méthode. Afin de faire interagir ces deux classes, une phase de connexion est nécessaire.

Cette connexion, qui s'effectue à partir de la configuration, consiste à mettre en relation les signaux entrant et sortant des différents SyncCharts. Pour cela des connecteurs comme ceux présentés figure 8.6(a) et 8.6(b) sont produits. Afin de ne pas avoir à modifier les syncCharts générés, le renommage des signaux est effectué au démarrage du module Esterel par l'instruction `run` (section 6.3). La manière dont les signaux sont renommés, et l'algorithme de construction des connecteurs est présenté figure 28. Pour minimiser les problèmes lors de la connexion nous posons quelques contraintes supplémentaires sur les diagrammes objets. Ainsi toutes les instances et les relations doivent être nommées, le nom des instances et des relations doit être unique et le type de l'instance doit être indiqué.

8.2.4 Simulateur de SyncClass

Le simulateur a pour but non seulement de permettre la simulation d'un comportement décrit par un SyncClass, mais encore le comportement produit par un ensemble

[1]Les diagrammes d'objets jouent ici le rôle des diagrammes de structure rencontrés dans ROOM [SGW94].
[2]Le format externe de ces diagrammes sera XMI [Gro98]

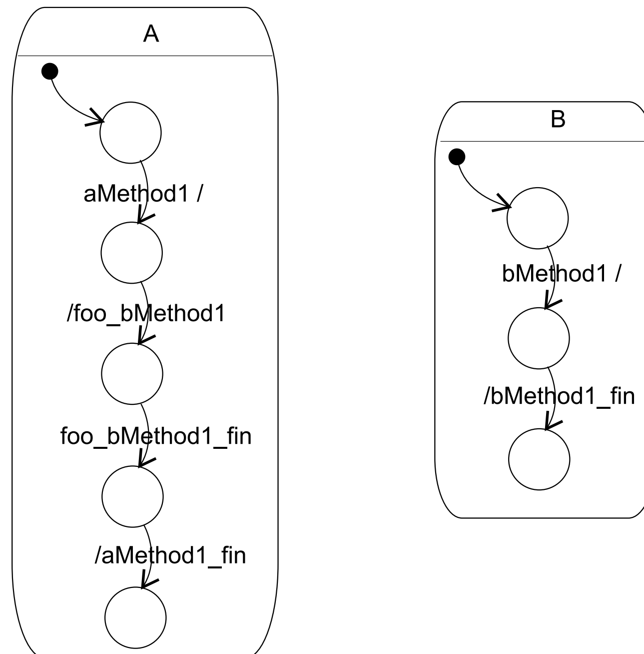
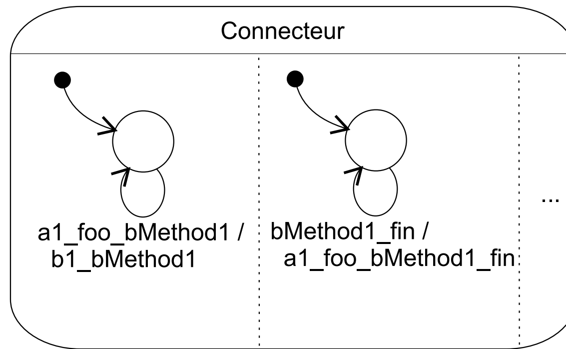


FIG. 8.5 – SyncCharts montrant le problème de communication de deux composants.

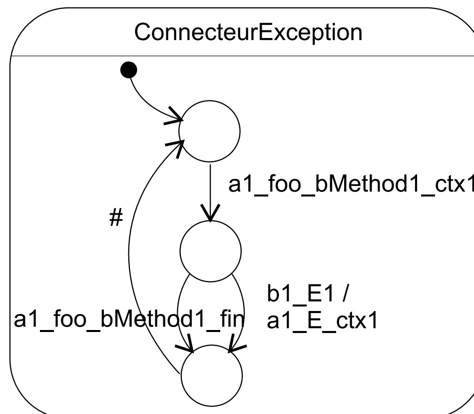
de composants. Grâce à cet outil on peut étudier le comportement d'un système sans avoir à écrire de code.

A l'heure actuelle, nous utilisons XES qui est le simulateur fourni avec la plate-forme Esterel. Une capture d'écran présentant ce simulateur en fonctionnement est donnée figure 8.8. La fenêtre située à gauche fournit l'interface de contrôle. Elle permet les interactions avec le système. Ainsi, pour simuler l'envoi de messages, il suffit de sélectionner l'entrée désirée puis de valider celle-ci en appuyant sur le bouton `tick`. La fenêtre de droite au premier plan présente l'organisation des macro-états du programme Esterel qui est simulé, et la fenêtre en arrière plan le code de ce programme.

Bien que ce simulateur soit pleinement fonctionnel, il n'est pas réellement adapté à l'utilisateur des syncClass. En effet, ce simulateur ne présente pas le syncClass, mais le code Esterel de la projection de ce syncClass en un syncCharts. Ce code, qui n'est lisible que lorsqu'on connaît les détails des projections, présente les signaux dans un format interne et montre en plus des signaux à usage domestique qui devraient être ignorés de l'utilisateur. Ainsi, afin d'éviter ces problèmes nous envisageons de mettre au point un simulateur dédié.

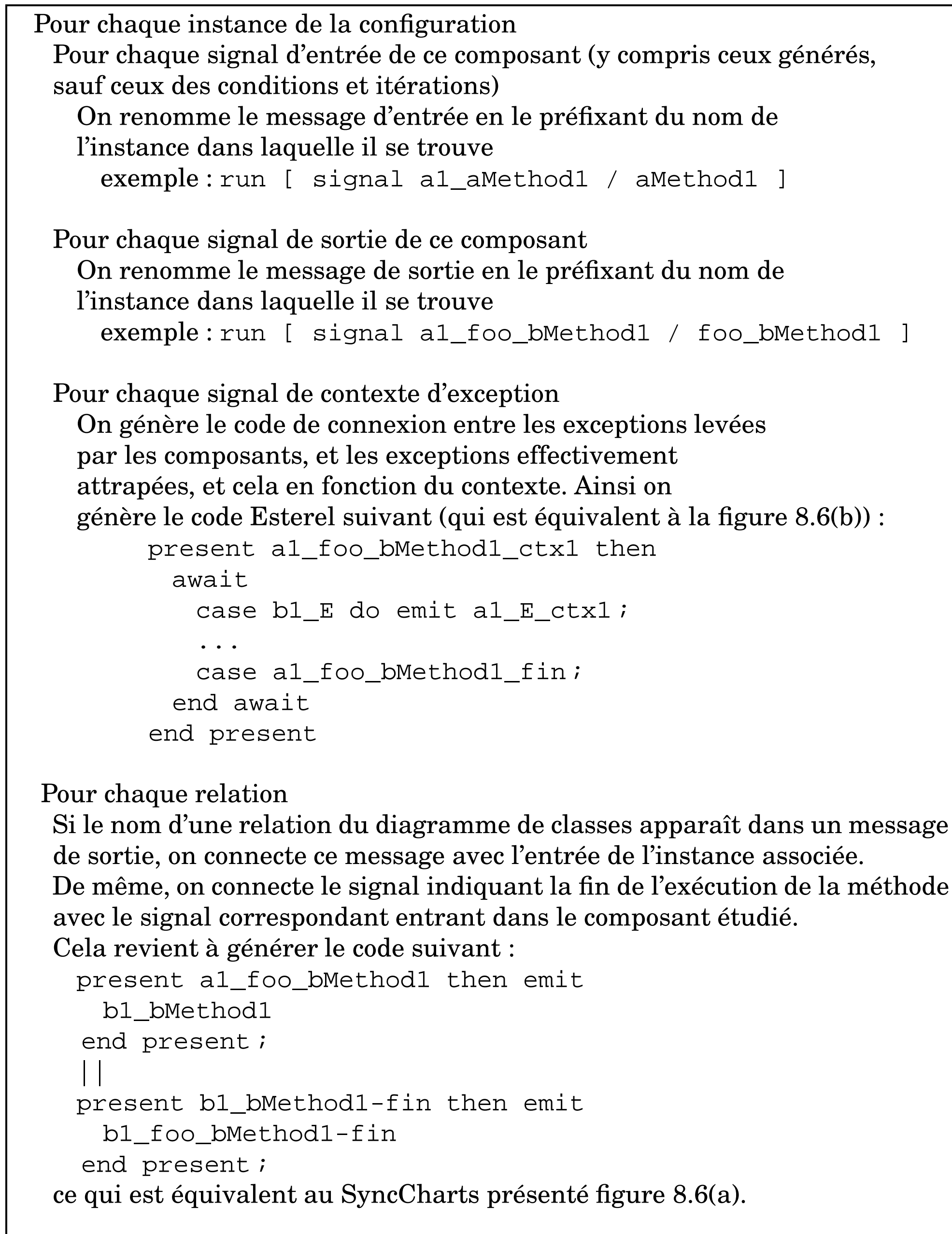


(a) Connexion des composants.



(b) Connexion des exceptions.

FIG. 8.6 – Connecteur généré par la configuration.



Pour chaque instance de la configuration
  Pour chaque signal d'entrée de ce composant (y compris ceux générés, sauf ceux des conditions et itérations)
    On renomme le message d'entrée en le préfixant du nom de l'instance dans laquelle il se trouve
      exemple : `run [ signal a1_aMethod1 / aMethod1 ]`

  Pour chaque signal de sortie de ce composant
    On renomme le message de sortie en le préfixant du nom de l'instance dans laquelle il se trouve
      exemple : `run [ signal a1_foo_bMethod1 / foo_bMethod1 ]`

  Pour chaque signal de contexte d'exception
    On génère le code de connexion entre les exceptions levées par les composants, et les exceptions effectivement attrapées, et cela en fonction du contexte. Ainsi on génère le code Esterel suivant (qui est équivalent à la figure 8.6(b)) :

```
present a1_foo_bMethod1_ctx1 then
  await
    case b1_E do emit a1_E_ctx1;
    ...
    case a1_foo_bMethod1_fin;
  end await
end present
```

Pour chaque relation
  Si le nom d'une relation du diagramme de classes apparaît dans un message de sortie, on connecte ce message avec l'entrée de l'instance associée.
  De même, on connecte le signal indiquant la fin de l'exécution de la méthode avec le signal correspondant entrant dans le composant étudié.
  Cela revient à générer le code suivant :

```
present a1_foo_bMethod1 then emit
  b1_bMethod1
end present;
||
present b1_bMethod1-fin then emit
  b1_foo_bMethod1-fin
end present;
```

  ce qui est équivalent au SyncCharts présenté figure 8.6(a).

FIG. 8.7 – Algorithme de connexion des composants utilisé par le configurateur.

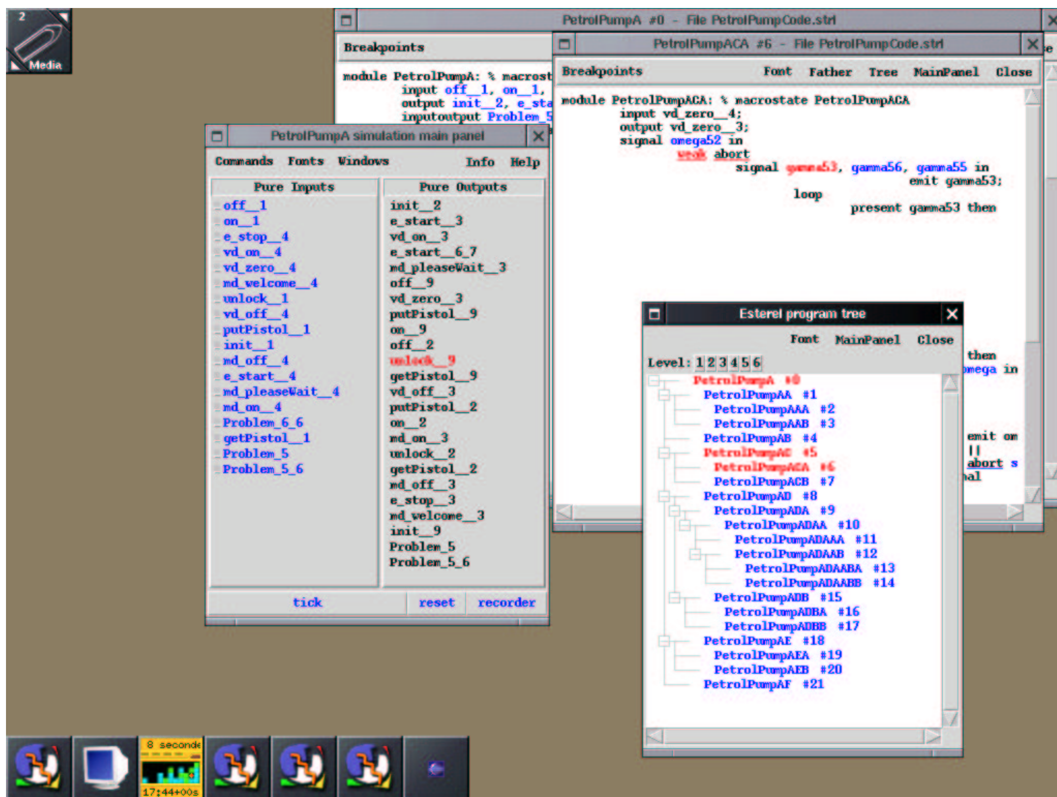


FIG. 8.8 – Capture d'écran du simulateur.

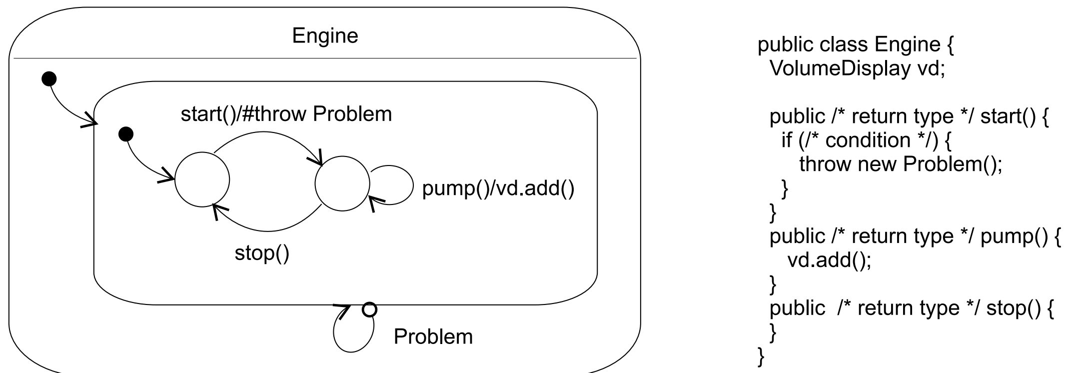


FIG. 8.9 – Point de vue client du moteur et son code source généré.

8.3 Générateurs

8.3.1 Générateur de code / générateur de SyncClass

Le rôle de ces deux outils est de fournir un système de *round trip engineering* entre les SyncClass et le code des composants qu'ils représentent. Ils permettent aussi de vérifier l'adéquation entre un composant et son SyncClass.

Générateur de code

Le générateur de code se focalise principalement sur la génération de squelette de méthode Java à partir d'un SyncClass comme cela est montré figure 8.9. Une fois la phase de conception terminée le développeur peut grâce à cet outil générer les squelettes des méthodes ainsi que celui de la classe. Étant donné que toutes les informations ne sont pas disponibles pour générer le code, des commentaires indiquent au développeur le type d'information qu'il faudra fournir.

Générateur de SyncClass

Cet outil génère les parties actions des SyncClass à partir du code des méthodes. Ainsi, il permet de compléter des syncClass qui n'auraient été que partiellement écrits.

La génération de ces parties est faite en parcourant le corps des méthodes : on ne conserve de la méthode qu'un squelette constitué de la structure de contrôle et des appels de méthodes. Si une méthode privée est appelée dans les parties actions, celle-ci est analysée et sa représentation intégrée en place et lieu de l'appel (ceci est comparable à une technique d'*inlining*).

Grâce au générateur de code et de syncClass, on peut aussi vérifier l'adéquation entre une action et le corps de la méthode qu'elle représente.

Implémentation

Étant donné que nous manipulons deux langages (le langage des parties actions et le langage dans lequel le composant est écrit, ici Java) nous utilisons deux compilateurs. Le premier est le compilateur Javac de Sun, dont nous avons arrêté le processus de compilation avant la génération de code. Nous avons fait ce choix car nous avons besoin que le typage soit résolu pour connaître le type exact des objets sur lesquels les méthodes sont appelées. Ce premier compilateur nous donne donc un arbre de syntaxe abstraite (AST) typé. Le second compilateur (simplement un analyseur syntaxique), que nous avons écrit, retourne simplement l'AST des actions.

Afin de générer le code Java, nous traversons l'AST retourné par le second compilateur.

Afin de générer les actions des SyncClass, nous transformons l'AST de la classe Java en un AST plus léger réduit aux structures de contrôle et appels de méthodes. Ce second AST est transformé en un AST des parties actions afin de générer les parties droites de transitions.

La comparaison d'une action et d'un code de méthode utilise une technique similaire à la précédente à la différence que la génération de code est remplacée par la comparaison des ASTs.

8.3.2 Générateur de tests

Faute de temps le générateur de tests n'a pas été mis en place, cependant nous pensons qu'il faut générer des tests fonctionnels. Ces tests n'ont pas pour objectif la couverture complète du code constituant le composant, mais une couverture complète du protocole. Cette technique de tests est habituellement très utilisée en phase de spécification ou de conception. Cependant, nous pensons qu'elle peut être utilisée afin de tester un composant une fois qu'il a été écrit.

Pour générer ces jeux de tests, nous envisageons d'utiliser une technique à base de graphes fonctionnels [XRK00]. Selon la littérature, la construction de ce graphe est habituellement la plus grande difficulté. Dans notre cas cette technique ne doit pas poser de problème puisque nous disposons du graphe fonctionnel sous la forme du point de vue client. Bien que nous soyons conscients de l'importance des données de tests (les paramètres passés aux appels de méthodes) dans cette activité, nous ne sommes pas en mesure de les fournir. Cela est la conséquence de la vue externe que les SyncClass donnent du composant, et qui donc n'apporte pas d'information sur le format des données attendues (à part sous la forme des paramètres dans la signature des méthodes).

Nous présentons rapidement les deux techniques de génération de tests que nous avons envisagées. La première utilise un vérificateur de modèles et la seconde des techniques de parcours de graphes.

Génération de tests en utilisant XEVE

Avant d'expliquer le principe de cette génération de tests, nous rappelons le fonctionnement de XEVE [BMDT96, Bou97]. Lorsque l'on cherche à prouver une propriété

(un signal est possiblement émis, toujours émis, jamais émis...) l'automate équivalent au programme Esterel est parcouru. Si lors de ce parcours la propriété est enfreinte, l'outil fournit une séquence de signaux entraînant la violation (contre-exemple de la propriété).

Tout comme les travaux de Gargantini [GH99], la génération de tests à partir de XEVE, utilise cette dernière propriété. Pour cela on commence par instrumenter le SyncCharts. A chaque fois qu'un signal est reçu, un accusé de réception spécifique à ce signal est émis. Pour obtenir une séquence de tests comportant un appel de méthode donnée, il suffit de demander à XEVE de vérifier que l'accusé de réception correspondant n'est jamais émis. Ce contre-exemple fourni par le vérificateur est donc une des séquences de tests cherchée.

Nous voyons deux inconvénients à cette technique. Le premier est que le vérificateur de modèles ne génère pas un ensemble de tests mais une unique séquence. Le second est que le chemin étant créé à partir d'un SyncCharts, il contient des signaux « domestiques ».

Générateur de tests à partir du graphe

Par rapport à la technique précédente, celle-ci apparaît comme plus traditionnelle. À partir du point de vue client qui constitue le graphe fonctionnel, nous pensons appliquer des algorithmes de parcours de graphe de manière à générer des séquences de tests. L'intérêt de cette technique est qu'elle permet, en fonction des algorithmes choisis, de mettre l'accent sur un aspect des séquences tels leur couverture, ou leur longueur.

Dans une recherche d'automatisation des tests, il est nécessaire de vérifier si l'exécution d'un test s'est bien passée. Ce rôle est généralement joué par un oracle. Dans l'état actuel de nos recherches nous ne sommes pas en mesure d'indiquer si nous pourrons le construire automatiquement.

8.4 Vérificateurs statiques et dynamique

Dans cette section nous présentons les différents outils de vérification que nous avons mis en œuvre. Ceux-ci sont au nombre de quatre dont trois sont des vérificateurs statiques. Les fonctionnalités statiques proposées sont :

– la vérification de l'adéquation d'un ensemble de composants dans une configuration donnée ;
– la vérification d'un code source par rapport à un point de vue client ;
– la vérification de la substituabilité d'un composant par un autre (sous-typage et remplacement).

L'utilisation de ces fonctionnalités ne requiert aucune connaissance particulière sur les vérificateurs de modèles.

Le dernier outil est un vérificateur dynamique qui permet la vérification des protocoles lors de l'exécution d'un programme.

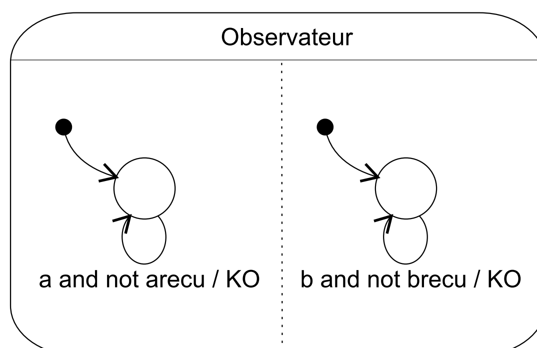


FIG. 8.10 – Observateur généré pour la vérification statique.

8.4.1 Vérificateur statique de l'adéquation d'un ensemble de composants

Le but de cette vérification est de s'assurer de l'adéquation des protocoles d'un ensemble de composants dans une configuration donnée. Cette vérification est effectuée avec comme hypothèse la correction du protocole d'entrée. Elle ne garantit le bon fonctionnement de l'assemblage que modulo cette hypothèse. Comparativement aux autres approches de l'étude bibliographique, notre outil prend en compte les exceptions et surtout les messages émis par chaque méthode.

Pour mettre en œuvre cette vérification on utilise le vérificateur de modèles XEVE. Le principe de la vérification consiste à observer le comportement du système. Cet observateur vérifie que tout appel de méthode envoyé est effectivement reçu par le composant à qui il est destiné. Dans ce cas « effectivement reçu » décrit, au-delà de la simple réception d'un message, l'adéquation de celui-ci avec le protocole : le message arrive au bon moment dans la séquence du composant. Pour cela, l'observateur émet le signal `KO` lorsqu'un message est émis sans être reçu. Comme le montre la figure 8.10, l'observateur est en fait un ensemble de petits observateurs mis en parallèle.

Ainsi lorsque le système et cet observateur sont fournis à XEVE, il ne nous reste plus qu'à vérifier comme propriété que `KO` n'est jamais émis.

Afin que l'observateur soit averti de la réception d'un message, une phase d'instrumentation complémentaire à la configuration est nécessaire. Cette phase d'instrumentation et la construction des observateurs sont décrites ci-après :

1. ajout d'un signal d'accusé de réception. Pour chaque signal entrant de chaque SyncClass du système, on ajoute l'émission d'un accusé de réception nominatif (par exemple `a` émet `arecu`) (figure 8.11) ;
2. création de l'observateur. Pour chaque signal entrant d'un SyncClass, on ajoute un observateur émettant le signal `KO` si un signal et son accusé de réception ne sont pas reçus dans le même instant (figure 8.10) ;
3. ajout d'une séquence d'entrée au système. Nous cherchons à prouver la validité de l'assemblage de composants dans le cas d'une utilisation normale (on rappelle que la vérification s'effectue modulo la correction de l'entrée). Ainsi si nous ne donnons

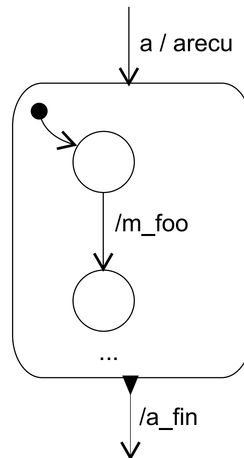


FIG. 8.11 – Instrumentation d'un syncCharts pour la vérification statique.

pas la séquence d'entrée, le vérificateur de modèles va chercher à explorer toutes les possibilités d'entrée sur le système et trouvera obligatoirement une erreur (KO) puisque rien ne l'empêchera d'appeler une méthode quelconque du protocole. La séquence a donc pour but de contraindre le vérificateur de modèles à n'envoyer que les signaux qui sont attendus.

Les erreurs que nous détectons sont celles de non-respect du protocole. Les propriétés de sûreté pourraient probablement être montrées ; cependant les critères d'observations sont différents, et les déterminer automatiquement est difficile.

Il est important de noter que dans les cas où tous les appels ne sont pas effectués sur le même composant, la séquence d'entrée devra être explicitement indiquée par l'utilisateur car nous ne pouvons pas encore la construire automatiquement. Ce cas se produit sur l'exemple de la pompe à essence où les messages peuvent être envoyés à la pompe ou au pistolet.

Nous avons vérifié l'adéquation des composants de la pompe à essence. Pour cela nous avons donné à XEVE la description instrumentée de notre système, et avons demandé de vérifier que le signal KO n'est jamais émis. Dans l'interface de XEVE qui est présentée figure 8.12, chercher à prouver cette propriété s'effectue en sélectionnant le carré rouge (encadré) puis cliquant sur le signal désiré dans la liste des signaux sortants.

Conformément à nos attentes, KO n'est jamais émis ce qui indique que dans la configuration donnée, nos composants s'utilisent correctement. Cela est indiqué par le message NEVER EMITTED (figure 8.13). Sur un Pentium III muni de 256Mo de mémoire vive, la vérification du système s'est effectuée en 23 secondes. Cependant la compilation du système est étrangement longue puisqu'elle a duré 17min. Cela est d'autant plus surprenant qu'Esterel a la capacité de manipuler des systèmes beaucoup plus conséquents. Bien que nous n'ayons pas encore réussi à réduire nettement ce temps et à identifier clairement la cause, nous pensons que cela est dû au compilateur Esterel

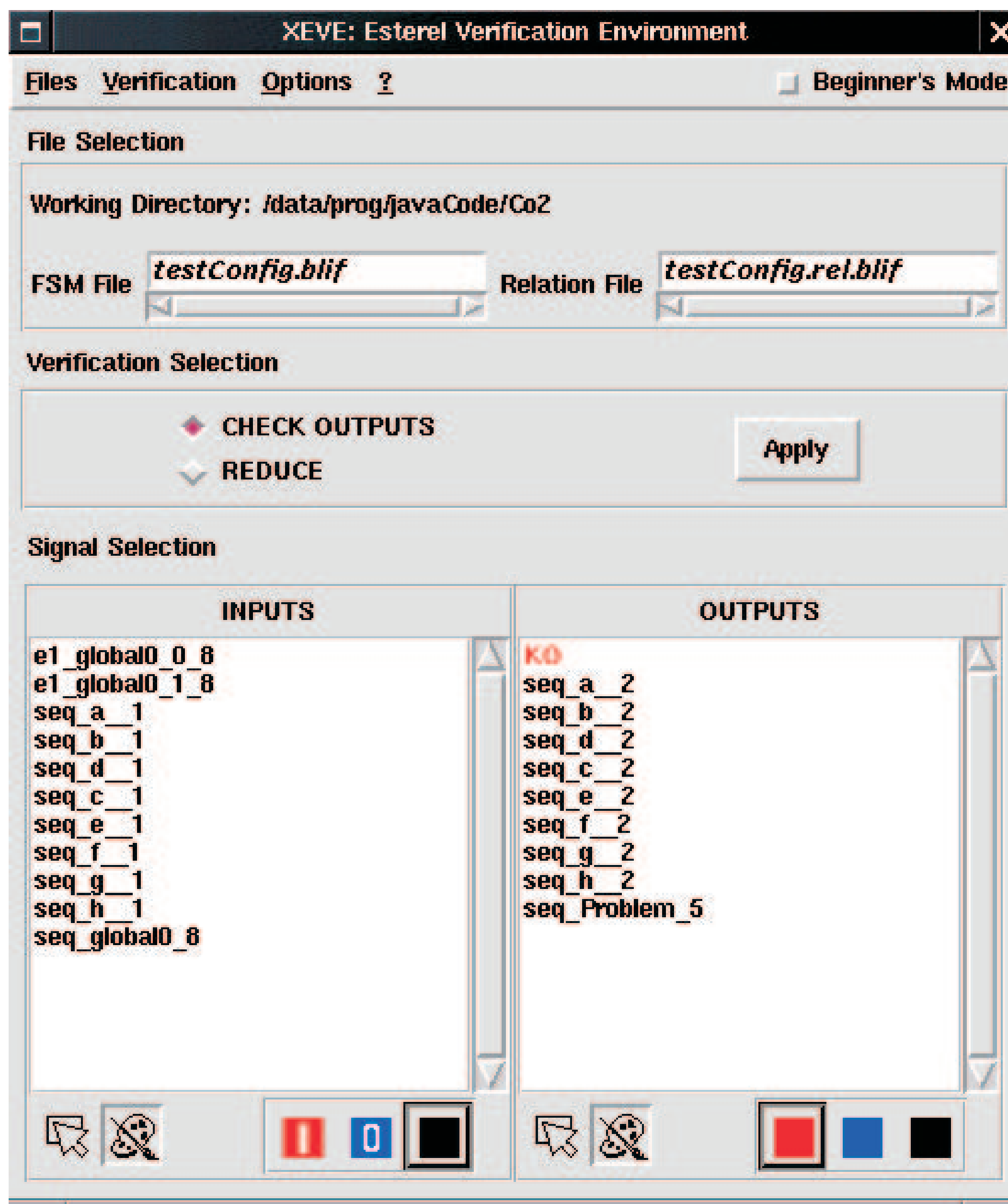


FIG. 8.12 – Utilisation de XEVE pour la recherche d'une potentielle émission de KO.

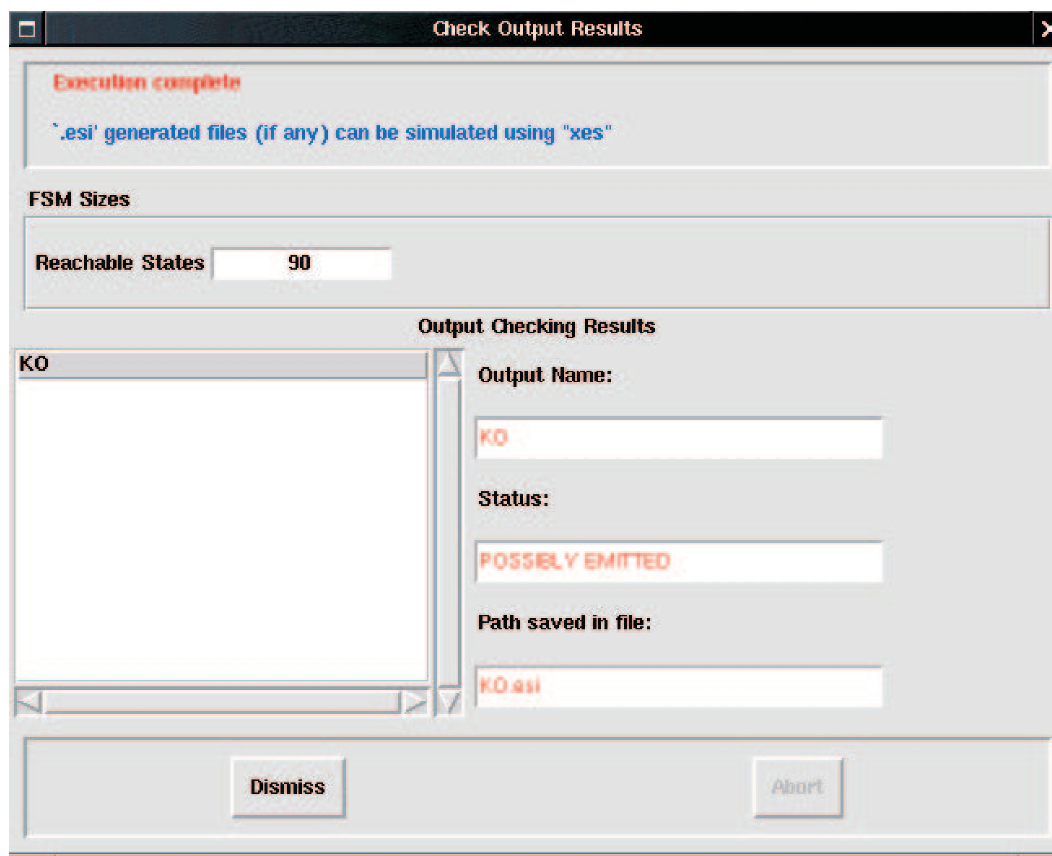


FIG. 8.13 – Fenêtre de résultat de XEVE indiquant que le signal KO n'est pas jamais émis.

(nos transformations ne prennent que quelques secondes). En effet, alors que le code que nous générons n'utilise que des préemptions faibles, le compilateur requiert, pour générer l'entrée à XEVE, qu'une vérification complète de causalité soit effectuée[3].

8.4.2 Vérificateur statique d'un code source par rapport à une configuration

Cette vérification a pour but de vérifier si un code source représentant l'utilisation d'un assemblage de composants fait une utilisation correcte de ce dernier. Elle constitue donc une variation de la vérification précédente qui testait la validité d'une configuration en supposant son utilisation correcte.

Afin d'être mise en place, cette vérification requiert les mêmes étapes que la phase d'instrumentation décrite dans la vérification précédente. La différence se trouve dans la troisième étape où la séquence d'entrée est remplacée par le code source dont on veut connaître la correction vis-à-vis d'une configuration.

Cette vérification n'est pas complètement implémentée. Il nous manque actuellement l'extraction du syncClass d'entrée à partir du code source.

8.4.3 Vérificateur statique de substituabilité comportementale

La vérification de substituabilité (ou *request substituability* [Nie95](section 5.5.3)) a pour but de vérifier qu'un composant peut être employé à la place d'un autre. Cette vérification revient à savoir si, dans la dimension comportementale, une classe est sous-classe d'une autre. On retrouve ici la notion de l'héritage de Wegner et Stanley dans [WZ88] : « *An instance of a subtype can always be used in any context in which an instance of a supertype was expected* » (encore appelé principe de Liskov).

Alors que les approches existantes se consacrent à montrer l'adéquation des protocoles d'utilisation, nous allons plus loin en testant l'adéquation du protocole d'inter-dépendance. En effet, la substituabilité ne doit pas seulement concerner le comportement à l'interface d'entrée mais aussi de celle de sortie. Par exemple sans ce test, on peut se demander quelle serait la pertinence d'une sous-classe qui respecterait l'adéquation du protocole d'entrée, mais dont l'implémentation serait vide.

Ainsi dans un premier temps nous présentons la technique que nous utilisons pour vérifier la substituabilité du protocole d'utilisation, puis regardons deux techniques pour vérifier la substituabilité du protocole d'inter-dépendance.

Vérification du protocole d'utilisation. Vérifier en terme de comportement qu'un type (par exemple `B`) est sous-type d'un autre (par exemple `A`) revient à montrer que le protocole d'utilisation de `B` se substitue au protocole d'utilisation de `A`. En terme d'automate, vérifier cette propriété revient à vérifier que toutes les séquences d'entrées reconnues par `A` le sont aussi par `B`.

[3]Cette opération qui est très longue fait une analyse complète du code à la recherche d'éventuels cycles de causalité.

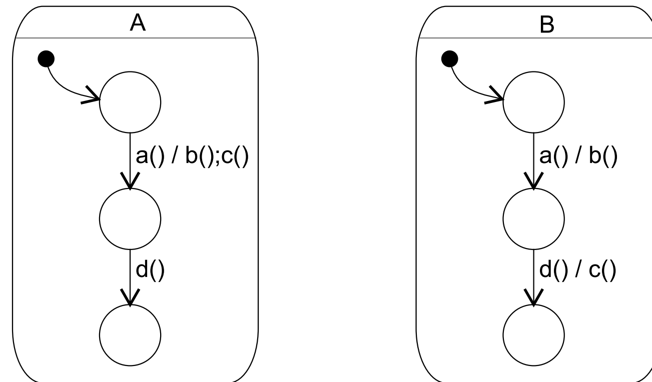


FIG. 8.14 – Exemple d'une sous-classe (B) ou un appel de méthode a été déplacé.

Afin de vérifier cette propriété, nous avons choisi d'utiliser XEVE sur un modèle similaire à ce que nous avions effectué pour les autres vérifications. Une phase d'instrumentation des syncCharts générés est nécessaire. Elle consiste à :

– ajouter un signal d'accusé de réception pour tous les signaux entrants du sous-type (`B`) à l'exception des signaux qui ne sont pas définis dans le type (`A`),
– ajouter un observateur pour tous les signaux qui émettent un accusé de réception,
– donner comme séquence d'entrée le syncCharts du super-type (`A`).

Si XEVE indique que `KO` n'est jamais émis les protocoles d'utilisation sont compatibles. Il est alors pertinent de tester la substituabilité des protocoles d'inter-dépendances.

Vérification du protocole d'inter-dépendance. Nous proposons une première technique basée sur l'analyse des parties actions des transitions. Cette analyse consiste à vérifier, pour chaque méthode, que les messages qui sont émis dans la méthode du type le sont aussi dans son éventuelle redéfinition du sous-type. Cette technique est cependant très restrictive car si un message a été déplacé d'une méthode à une autre, alors une erreur est indiquée (figure 8.14)

Afin d'offrir plus de souplesse, nous proposons une seconde technique de vérification basée sur la transformation des actions en un SyncClass équivalent. Cette transformation est faisable puisque le langage utilisé dans ces parties droites est une expression régulière qui peut donc être transformée en un automate [BS86]. Ainsi dans le SyncClass de `A`, on substitue à chaque transition l'automate équivalent, on fait la même chose pour `B`, puis on teste la compatibilité des deux automates ainsi créés. Pour cela nous utilisons XEVE et avons recours à une phase d'instrumentation similaire à celle de la vérification du protocole d'utilisation.

Alors que la première partie de la vérification s'intéressait à l'inclusion du protocole du super-type (`A`) dans celui du sous-type (`B`), la seconde partie vérifie l'inverse : les messages émis par le sous-type doivent être compatibles avec ceux du super-type. Cela est nécessaire car les méthodes des sous-types font potentiellement plus d'appels que celles des super-types, et il ne serait donc pas correct qu'une erreur soit indiquée.

Cette vérification a été utilisée pour vérifier les exemples présentés dans le chapitre suivant ainsi que sur des exemples jouets consistant à remplacer le moteur ou l'afficheur de la pompe.

8.4.4 Vérificateur dynamique

Par opposition aux vérifications statiques, nous proposons ici la vérification des protocoles à l'exécution. Elle permet de vérifier les points de vue client ou les points de vue de composition. La vérification des points de vue client vérifie que toutes les instances d'un composant sont correctement utilisées, et la vérification des points de vue de composition qu'un composant utilisent correctement les instances d'autres composants.

Lorsqu'une vérification dynamique est effectuée sur un composant donné, toutes les instances de ce composant sont vérifiées. Cela permet d'assurer que dans tous les cas d'utilisation rencontrés lors de l'exécution de l'application, le protocole n'a pas été violé. Si une erreur se produit dans l'utilisation du protocole, une erreur est indiquée à l'utilisateur du composant, et l'ensemble des méthodes qui aurait dû être appelé est indiqué.

Contrairement à des approches de vérifications statiques où il est nécessaire de préciser les connexions entre les composants, la vérification dynamique ne requiert pas de phase de configuration puisque cette configuration est automatiquement fournie par le programme utilisant le composant.

Mise en œuvre. La mise en œuvre de cette vérification requiert le contrôle sur l'envoi et la réception de messages afin de vérifier, avant l'exécution effective de la méthode, sa validité par rapport au protocole. Le protocole est représenté par un méta-objet qui contrôle les messages. A chaque instance d'un composant est associé une instance d'un méta-objet. Le nombre d'instances ne pouvant être déterminé qu'à l'exécution, la création et l'association d'une instance d'un point de vue sont faites à chaque instanciation du composant vérifié.

De nombreuses techniques à base de méta-objets nécessitent une préparation du code source. Dans notre contexte, une telle préparation n'est pas envisageable car elle altérerait les performances du composant, ou obligerait à gérer deux composants simultanément : un pour le déverminage, et l'autre pour l'exploitation. Afin d'éviter ce problème, puisque nous étions en Java, nous avons utilisé la réflexivité structurelle au chargement [Chi00], ce qui permet de modifier le composant uniquement lorsqu'il est utilisé. Ainsi, grâce à Javassist qui implémente cette technique, nous avons pu mettre en place un système de *wrappers* [BFJR98] ; nous avons aussi associé à la création de chaque instance d'un composant la création d'un méta-objet le contrôlant.

La figure 8.15 permet de comparer le comportement du système avec ou sans vérification. Elle montre l'appel d'une méthode sur un composant (on vérifie sur ce composant le point de vue client), ainsi que l'appel d'un composant sur un autre composant (on vérifie le point de vue de composition). Les indices apposés à côté des flèches indiquent l'ordre d'exécution : les numéros sans prime (1,2,3) représentent l'exécution normale du programme, les numéros avec primes viennent s'insérer dans l'exécution normale et représentent les appels faits au méta-objet. Ainsi on constate que pour la

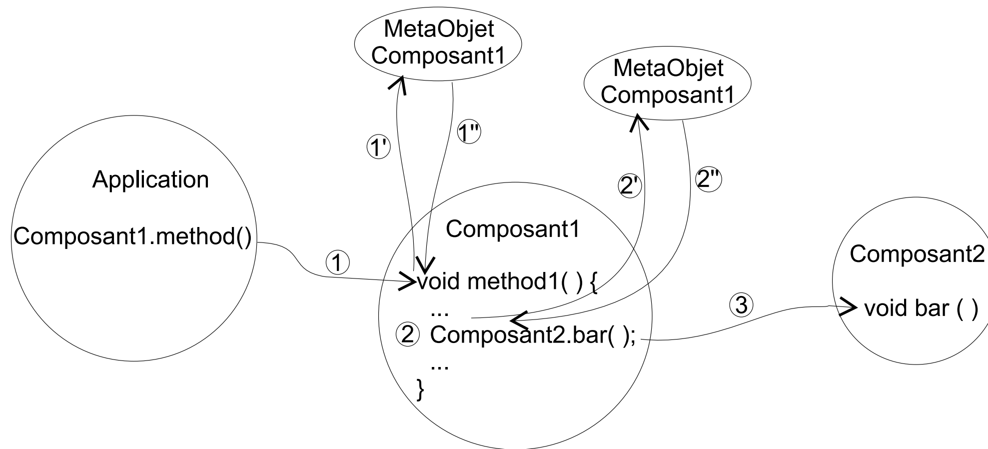


FIG. 8.15 – Vérification dynamique par méta-objet.

vérification de la vue cliente le méta-objet intervient à la réception de la méthode, alors que pour la vue de composition, il intervient juste avant l'appel à la méthode.

Lorsqu'un utilisateur décide de vérifier les points de vue comportementaux, il lui suffit de sélectionner les composants et les points de vue qu'il désire vérifier et de lancer l'exécution de son application.

8.5 Conclusion sur l'utilisation des syncClass

Nous avons montré comment les SyncClass, notation graphique et formelle, s'inséraient dans le cycle de vie du composant, et comment des outils associés aux activités de ce cycle venaient aider développeurs et utilisateurs de composants que ce soit dans le contexte d'un framework ou non. Le tableau 8.2 résume quels sont les outils utilisés dans les différentes activités.

Ce chapitre a aussi permis de montrer qu'en alliant graphisme et formalisme les techniques de vérification de modèles pouvaient être utilisées pour prouver certaines propriétés d'un framework.

8.6 Conclusion générale sur notre modèle d'expression des dépendances comportementales et travaux futurs

Dans cette partie, nous avons présenté un modèle d'expression des dépendances comportementales basé sur le concept de point de vue. La représentation de ces points de vue est faite dans un langage graphique et formel que nous avons créé et qui s'appelle SyncClass. Ce modèle formel fondé sur le modèle synchrone offre la possibilité de

Outils / Activités	Conception	Codage	Tests	Finalisation	Validation	Exploitation
Editeur, navigateur	✓					
Vérificateur de composition	✓				✓	✓
Simulateur	✓	✓			✓	
Générateur de tests			✓			
Générateur de code		✓				
Générateur de SyncClass		✓				
Finaliseur				✓		
Vérificateur dynamique		✓		✓		✓

TAB. 8.2 – Outils utilisés par activité.

vérifier statiquement et dynamiquement la cohérence des dépendances comportementales. De surcroît les SyncClass sont intégrés tout au long du cycle de vie du framework.

Les travaux futurs autour des SyncClass et de Co^2 peuvent être organisés selon quatre axes qui sont l'extension du modèle des composants, l'extension de la notation graphique, les outils, ainsi que les applications.

8.6.1 Modèle de composant

Comme nous l'indiquions en introduction au chapitre 7, notre modèle de classe est restreint à un modèle de composants réactifs, non concurrents (ou sous des conditions particulières). Afin de pouvoir modéliser le plus grand nombre de frameworks et d'applications, il nous paraît primordial de lever ces restrictions. Ainsi dans un premier temps nous pensons étendre notre modèle afin d'y supporter un modèle général de concurrence.

Puisqu'actuellement le langage vers lequel nous projettons les syncClass est Java, il serait intéressant d'étudier l'incidence de certaines constructions du langage comme la coercition de type, ou le clonage.

Afin de représenter ces nouvelles opérations il sera probablement nécessaire d'étendre la notation graphique.

8.6.2 Notation graphique

Au-delà des modifications qui pourraient être requises par l'extension du modèle, nous pensons regarder l'intérêt et l'impact de l'ajout de variables locales à un SyncClass. Un tel mécanisme permettrait d'obtenir pour certains cas une représentation plus précise du comportement d'un composant. Cependant, il faudrait alors regarder les capacités des vérificateurs de modèles et des autres outils à supporter une telle notation.

Une autre direction de recherche sur la notation graphique est la définition de règles de raffinement syntaxiques à la manière de celles proposées par Rumpe pour les automates [RK96] (sous-section 5.5.2). L'intérêt de telles règles, si elles étaient respectées, est de garantir la substituabilité par construction.

8.6.3 Outils

Les pistes de recherche pour d'autres outils tirant partie de ce modèle sont nombreuses. La première, qui nous tient particulièrement à cœur, est la création de composants de plus gros grain à partir d'une configuration. Ainsi sur l'exemple de la pompe à essence, au lieu d'avoir plusieurs composants, cet outil nous permettrait de créer un composant monolithique pompe à essence. Au-delà de l'aspect purement structurel du rapprochement, ce qui nous intéresse ici est la possibilité de déterminer le protocole de ce composant monolithique à partir du protocole des composants qu'il rassemble.

Un autre outil qui nous paraît essentiel est un outil permettant de vérifier l'adéquation partielle de deux composants. Un tel support semble particulièrement utile à l'utilisateur qui veut faire des vérifications mais dont tous les composants ne sont pas encore spécifiés (ce qui l'empêche de faire une vérification complète du système). Par exemple, étant donné le point de vue de composition de la pompe sur l'afficheur, et le point de vue client de l'afficheur, cet outil permettrait de vérifier que le point de vue de composition est bien compatible avec le point de vue client. Cette vérification est similaire à la notion de *request satisfiability* [Nie95], mais dans un contexte où le composant n'est pas complètement utilisé.

Dans le contexte d'un framework noir où la personnalisation privilégie la composition de composants, il nous semble primordial d'avoir un outil qui permette la génération d'un composant d'adaptation (d'interposition) à la manière de ce que propose Yellin [YS94] (sous-section 5.5.4).

Comme la compilation d'Esterel permet de cibler de nombreux langages, il est envisageable de se détacher des outils de la plate-forme. Ainsi, nous pensons qu'il pourrait être intéressant de regarder de nouveaux vérificateurs de modèles afin de tester leur performance et voir s'ils peuvent nous apporter de nouvelles fonctionnalités.

8.6.4 Applications

Les SyncClass ne sont pas limités à la documentation d'un framework et nous explorons quelques autres utilisations possibles.

Génie logiciel. Les syncClass n'ont pas fait l'objet d'études en vraie grandeur. Afin de valider ce concept de manière plus large, il est donc primordial de faire utiliser les syncClass et les concepts de points de vue comportementaux, et d'en analyser l'adéquation avec les besoins des utilisateurs.

L'autre point qui nous paraît primordial est l'étude d'une technique maintenant les syncClass en cohérence avec les autres diagrammes comportementaux et les diagrammes structurels que l'on peut rencontrer dans une modélisation faite en UML.

Grâce aux syncClass la visualisation du comportement inter-composants étant simplifiée, ils pourraient permettre la découverte de nouveaux schémas de conception comportementaux.

Langage. S'inscrivant dans la suite de ce qui est proposé dans la vérification dynamique (sous-section 8.4.4), il est envisageable d'étendre le modèle de réflexivité des langages de programmation de manière à rendre accessible le protocole. Il serait alors possible d'imaginer des programmes qui tireraient partie de cette information pour adapter leur comportement. Un tel mécanisme nous paraît intéressant pour la mise en place d'applications devant être reconfigurées dynamiquement, puisque la décision de reconfiguration pourrait alors être guidée par observation du protocole.

Une autre possibilité d'utilisation de ce protocole est la création de composants autocorrecteurs. Ceux-ci permettraient une plus grande souplesse d'utilisation des composants puisque lors de l'apparition d'un message indésirable (par rapport au protocole) on pourrait alors mettre en attente ledit message et ne l'exécuter qu'une fois le protocole prêt à l'accepter. Toutefois, la mise en place d'un tel système est non triviale puisqu'il faut prendre en compte les modifications éventuelles des contextes d'appels qui peuvent se produire entre le moment où le message est appelé et le moment où il est explicitement invoqué.

Troisième partie

Expression du modèle conceptuel des dépendances d'un framework

Chapitre 9

Mise en œuvre des dépendances dans BLOCKS

Chacune des techniques présentées précédemment résout un problème particulier lié à l'utilisation d'un framework. La première résout les problèmes liés à la personnalisation en exprimant les dépendances structurelles d'un framework, et la seconde les problèmes liés à l'instanciation en montrant les dépendances comportementales.

Cette partie a pour but de montrer la complémentarité des deux approches. Pour cela, nous utiliserons le framework BLOCKS développé dans l'équipe ORION de l'INRIA. Ce framework comme la plupart des frameworks comporte à la fois des dépendances structurelles et comportementales. Ce chapitre va donc s'attacher à montrer comment les deux modèles sont réunis et constituent le modèle conceptuel global des dépendances du framework.

9.1 Présentation de BLOCKS

BLOCKS est un framework dont l'objectif est de faciliter la création de moteurs de systèmes à base de connaissance. Pour cela il propose l'implémentation de concepts d'Intelligence Artificielle tel que les *frames* ou les règles. Ce framework d'une soixantaine de classes ne fournit pas de boucle de contrôle, son code source est accessible et son l'utilisation principale se fait par extension.

Dans ce chapitre nous ne détaillerons que la partie dédiée à la gestion de l'historique. Cette fonctionnalité d'un système à base de connaissance est essentielle, puisqu'elle permet de revoir l'évolution d'un raisonnement.

Dans le fonctionnement d'un historique, on peut isoler deux phases : la phase d'évolution (conséquence du fonctionnement du moteur) et la phase de consultation. Bien que dans la version actuelle de BLOCKS, ce gestionnaire soit utilisé par un moteur d'inférences fourni, il est envisageable qu'il soit utilisé seul.

Ainsi dans cette section nous regardons le fonctionnement de ce système et les contraintes liées à son instanciation. Dans une seconde section nous verrons les problèmes posés par sa personnalisation.

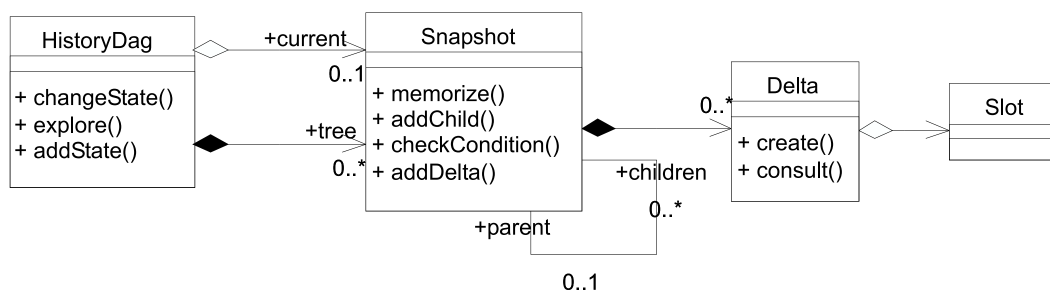


FIG. 9.1 – Diagramme de classes de la gestion d'historique dans Blocks.

9.1.1 Fonctionnement de l'historique

Un historique (`HistoryDag`) est un arbre de clichés (`Snapshot`). Ainsi, un cliché liste les valeurs qui ont changé depuis le dernier cliché. Ces valeurs sont appelées delta (`Delta`), terme qui décrit bien la notion présentée. Un delta encapsule la valeur d'un `Slot`. L'organisation des classes utilisées dans la gestion d'un historique est donnée figure 9.1.

Comme nous l'indiquions en introduction, l'historique est habituellement géré par un moteur d'inférences. Afin de faciliter la tâche de ce dernier la classe `HistoryDag` fournit un ensemble de services assurant que pendant l'évolution d'un raisonnement il y aura toujours un cliché ouvert. Ce composant interagit donc principalement avec les clichés puisqu'il en gère le cycle de vie.

Le cycle de vie d'un cliché (`Snapshot`) est le suivant : lorsqu'une nouvelle instance est créée, il est automatiquement ouvert, et est donc prêt à recevoir des deltas. La fermeture d'un cliché est une opération explicite. À partir de cet instant, le cliché ne peut plus être modifié, cependant il peut faire l'objet de vérifications (`checkCondition()`) et peut se voir ajouter des fils (`addChild()`). Le protocole reflétant ce cycle de vie est présenté dans un SyncClass figure 9.2.

En plus de gérer le cycle de vie des clichés, le gestionnaire d'historique (`HistoryDag`) est chargé à la fois de maintenir un cliché courant, et de gérer la construction de l'arbre des clichés.

Ainsi lorsqu'elle est instanciée cette classe crée donc un nouveau cliché qui sera le cliché courant. La fermeture du cliché courant est explicitement faite par la méthode `changeState()`. Cependant un nouvel état courant n'est pas automatiquement créé, permettant l'exploration de l'arbre. Le protocole d'`HistoryDag` est décrit figure 9.3. Il montre que la fermeture du cliché est effectuée lorsque le changement d'état est requis (`changeState()`), que l'exploration de l'arbre qui est conditionnelle est l'opération qui utilise la vérification des conditions (`explorate()`)[1] et que l'ajout d'un nouveau cliché qui deviendra alors l'état courant est fait explicitement (`addState()`).

Grâce à ce syncClass on voit que l'ajout de deltas n'est pas géré par l'`HistoryDag`. Cet ajout est fait par le `Slot` lui-même qui, lorsqu'il est accédé en écriture, crée un delta et l'ajoute au cliché courant (`memorize()` sur `Snapshot`).

[1]La notation utilisée en partie droite de cette méthode décrit l'équivalent d'un `forAll` pour tous les éléments contenus dans `tree`.

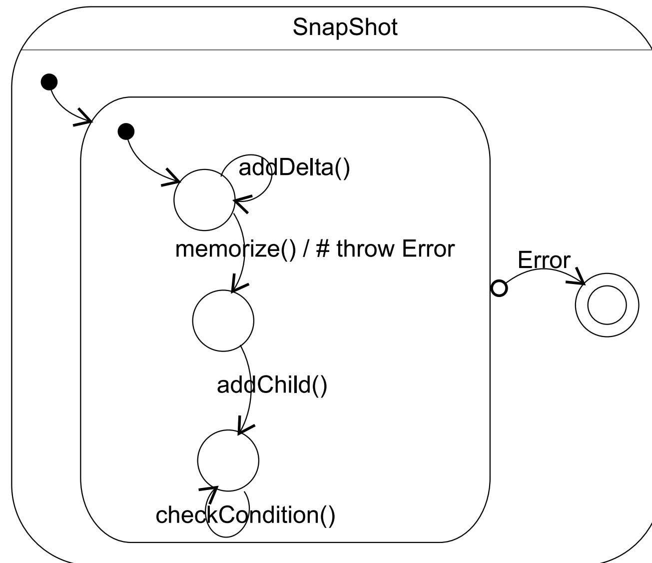


FIG. 9.2 – Point de vue client de la classe Snapshot.

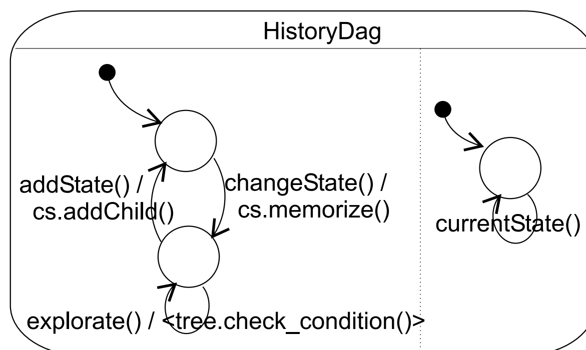


FIG. 9.3 – Point de vue client de la classe `HistoryDag`.

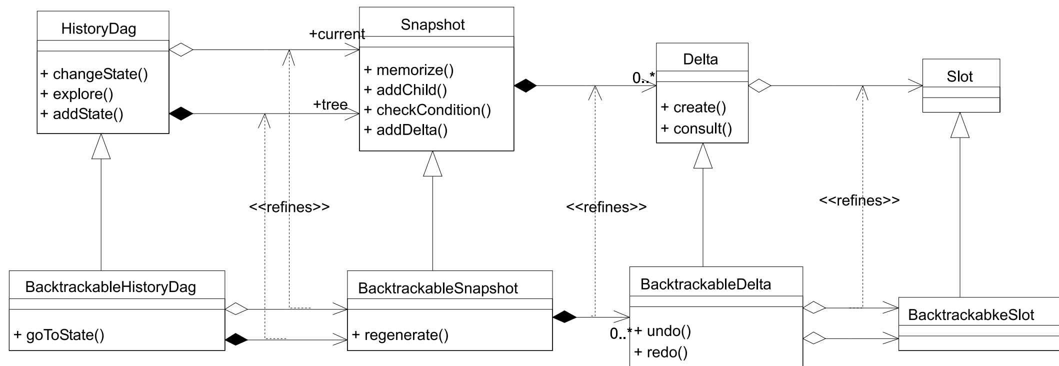


FIG. 9.4 – Extension de BLOCKS pour la gestion d'un historique navigable.

On notera ici que la vérification statique de l'assemblage d'un `HistoryDag` et de plusieurs `SnapShots` (il ne faut pas oublier que l'arbre est un ensemble de `SnapShot`) indique une erreur. Cette erreur qui se déclenche sur la méthode `changeState()` d'`HistoryDag` indique que le cliché courant `cs` n'est pas prêt à recevoir `memorize()` lorsque la séquence `changeState()`, `addState()`, `changeState()` se produit. En fait l'erreur ne vient pas d'une mauvaise utilisation du cliché mais de l'impossibilité d'exprimer le fait que l'appel à `addState()` a entraîné le changement de l'instance référencée par `cs`. Bien que nous n'ayons pas essayé, un tel problème ne se produira pas à l'exécution.

L'utilisation de ce framework se fait donc soit en utilisant directement les classes énoncées précédemment (instanciation), soit en les étendant puis les instanciant (personnalisation). Selon les auteurs du framework, l'extension de ces classes est la technique la plus délicate car il faut à la fois que les extensions produites soient sémantiquement acceptables : elles doivent correspondre aux hypothèses de conception du framework tout en respectant le protocole d'utilisation des classes.

9.1.2 Une extension de l'historique

L'exemple d'une telle extension est fourni avec le framework. Elle met en place un système de gestion d'historique *backtrackable* qui permet de remettre le système dans un état rencontré dans le passé et de recommencer le raisonnement à partir de celui-ci. Afin de changer l'état du système les valeurs sauvegardées dans les clichés sont rétablies les unes après les autres en remontant dans l'arbre.

Ce nouveau système de gestion d'historique introduit quatre nouvelles classes : `BacktrackableHistoryDag`, `BacktrackableSnapshot`, `BacktrackableDelta` et `BacktrackableSlot` réciproquement sous-classes d'`HistoryDag`, `Snapshot`, `Delta` et `Slot`. Cette extension est présentée figure 9.4.

Puisque l'historique n'est plus simplement analysable mais permet « le voyage dans le temps », l'interface de sa sous-classe (`BacktrackableHistoryDag`) est étendue de la méthode `goToState()`. Cette méthode appelera alors `regenerate()` sur le sous-

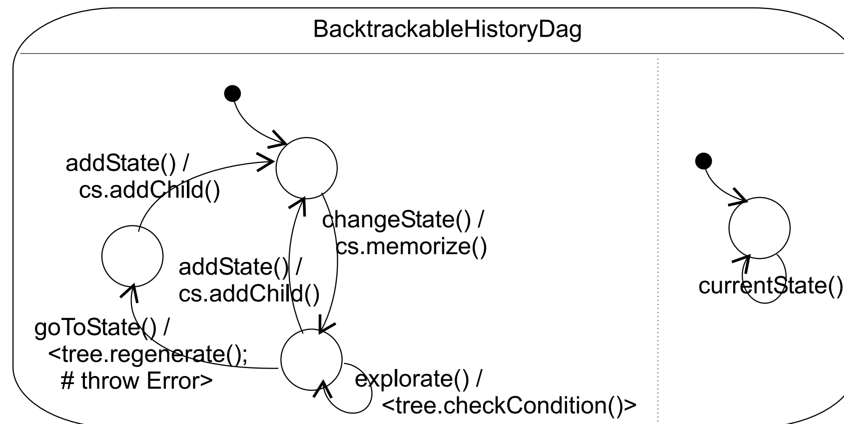


FIG. 9.5 – Point de vue client de `BacktrackableHistoryDag`.

ensemble des clichés nécessaires à la remise en état du système. Le nouveau point de vue client est montré figure 9.5.

Comme nous le laissions sous-entendre le protocole client de `BacktrackableSnapshot` est aussi modifié puisqu'il contient désormais la méthode `regenerate()`. On voit grâce au point de vue comportemental qu'une fois la regénération effectuée il est possible de créer une histoire parallèle puisqu'`addChild` peut être appelé.

Afin que les valeurs des slots puissent être remises à jour, les deltas sont munis d'une méthode `undo` et une méthode `redo`.

On notera que les extensions présentées de `Snapshot` et d'`HistoryDag` préservent la substituabilité comportementale. Pour s'en convaincre rapidement, il suffit de considérer le point de vue client de `BacktrackableSnapshot` (figure 9.6) en le restreignant aux entrées de `Snapshot`. On constate alors que le protocole est le même (figure 9.2). On appliquera le même raisonnement pour `HistoryDag` et `BacktrackableHistoryDag`.

Après discussion avec l'auteur il est apparu que toute extension de `Snapshot` suit un schéma similaire à celui requis pour l'extension de `BacktrackableSnapshot`. Ainsi toute création d'une sous-classe `Snapshot` entraîne la création d'une sous-classe de `Delta` et de `Slot`. De plus, une autre contrainte plus spécifique concernant les sous-classes est apparue : toute sous-classe d'un `BacktrackabkeSnapshot` doit référencer un `BacktrackableDelta` ou l'une de ses sous-classes. Les applications développées ne requièrent pas nécessairement la création d'une sous-classe d'`HistoryDag`.

9.2 Dépendances de BLOCKS

Comme nous l'avons vu l'utilisation de BLOCKS est régi à la fois par des dépendances structurelles et comportementales. Les dépendances comportementales ayant déjà été présentées, afin de simplifier la présentation du système, nous ne décrirons ici que les dépendances structurelles.

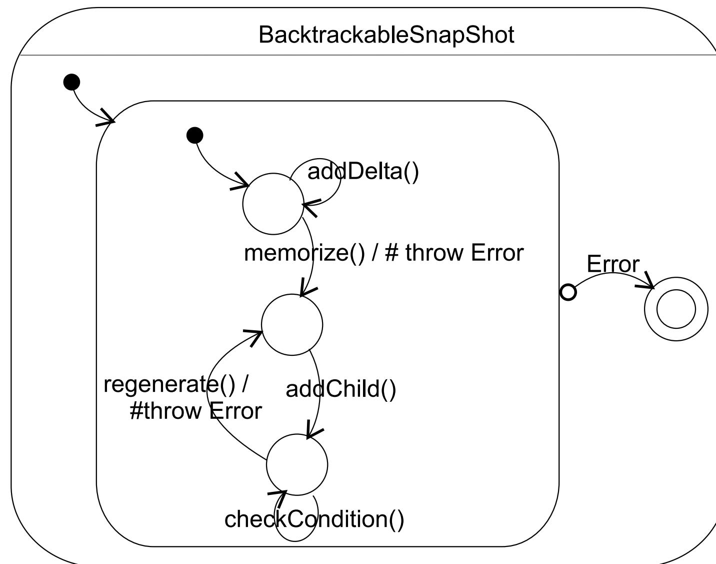


FIG. 9.6 – Point de vue client de la classe `BacktrackableSnapshot`.

9.2.1 Extension de Snapshot

La première dépendance structurelle mise en place est celle qui gère la création des sous-classes de `Snapshot`.

Comme nous l'avons expliqué au chapitre 4 l'une des premières étapes dans la description des dépendances structurelles est la création d'une réification des éléments pertinents. Ainsi, pour ce framework nous avons choisi d'avoir un élément conceptuel pour une classe et un autre pour sa sous-classe. Nous obtenons donc les six éléments conceptuels suivants : `SnapShot`, `SousSnapShot`, `Delta`, `SousDelta`, `Slot` et `SousSlot` que nous connectons deux à deux par une relation d'`heritageN`. La classe `HistoryDag` n'étant ni cause ni conséquence d'une dépendance, elle n'est pas réifiée.

Afin de représenter le fait que la création d'une sous-classe de `SnapShot` requiert la création d'une sous-classe de `Delta` nous avons réutilisé la dépendance existentielle présentée figure 4.12 que nous instancions entre les éléments conceptuels de `SousSnapshot` et `SousDelta`.

Nous avons donc une première expression des dépendances structurelles qui est présentée figure 9.7.

9.2.2 Extension de BackstrackableSnapshot

Afin de représenter la deuxième contrainte, deux possibilités s'offrent à nous : la création de nouveaux éléments conceptuels représentant `BacktrackableDelta` et `BacktrackableSnapshot` qui jouent des rôles particuliers dans notre framework, ou la création d'une dépendance spécifique posée entre les éléments conceptuels existants.

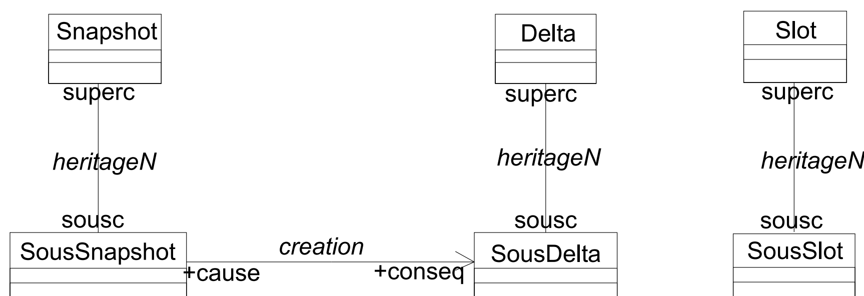


FIG. 9.7 – Dépendance pour la création de sous-classe de `Snapshot`, `Delta` et `Slot`.

Nous avons retenu la deuxième solution qui considère les éléments *backtrackable* comme des classes normales et avons choisi d'écrire une dépendance forçant toute nouvelle sous-classe de `BacktrackableDelta` à être associée à une sous-classe de `BacktrackableSlot`. Afin de retrouver la super-classe d'un élément, nous avons recours à une fonction embarquée nommée `isKindOf`. La dépendance est présentée figure 9.8.

Bien sûr lors de la création de sous-classes de `SnapShot`, il faudra vérifier que le protocole de celles-ci respecte la substituabilité comportementale.

9.3 Conclusion sur la mise en œuvre des dépendances dans BLOCKS

Grâce à l'exemple présenté, la dualité structure / comportement qui est propre à toute application est retrouvée, et la complémentarité des deux modèles est montrée. En effet, il apparaît clairement à travers l'exemple de la classe `Snapshot` ou `HistoryDag` qu'une unique dépendance structurelle ou comportementale aurait été insuffisante pour représenter à la fois l'utilisation et l'extension de ces classes. En effet les dépendances structurelles ne peuvent pas capturer des informations liées au protocole des composants du framework, et les dépendances structurelles ne peuvent pas non plus capturer des informations sur l'organisation structurelle des extensions du framework.

De plus cet exemple a permis de percevoir la nécessité d'une bibliothèque de dépendances, mais aussi de langages simples comme ISL et OCL pour expliciter les dépendances structurelles. En ce qui concerne le point de vue comportemental, cet exemple a permis de confirmer que malgré les restrictions liées au modèle d'exécution notre notation pouvait servir à représenter de vrais frameworks. Cependant, nous avons aussi vu les limites des vérifications statiques dans le cadre de notre modèle.

Nous avons donc montré comment les dépendances structurelles et comportementale permettaient d'exprimer le modèle conceptuel des dépendances du framework BLOCKS.

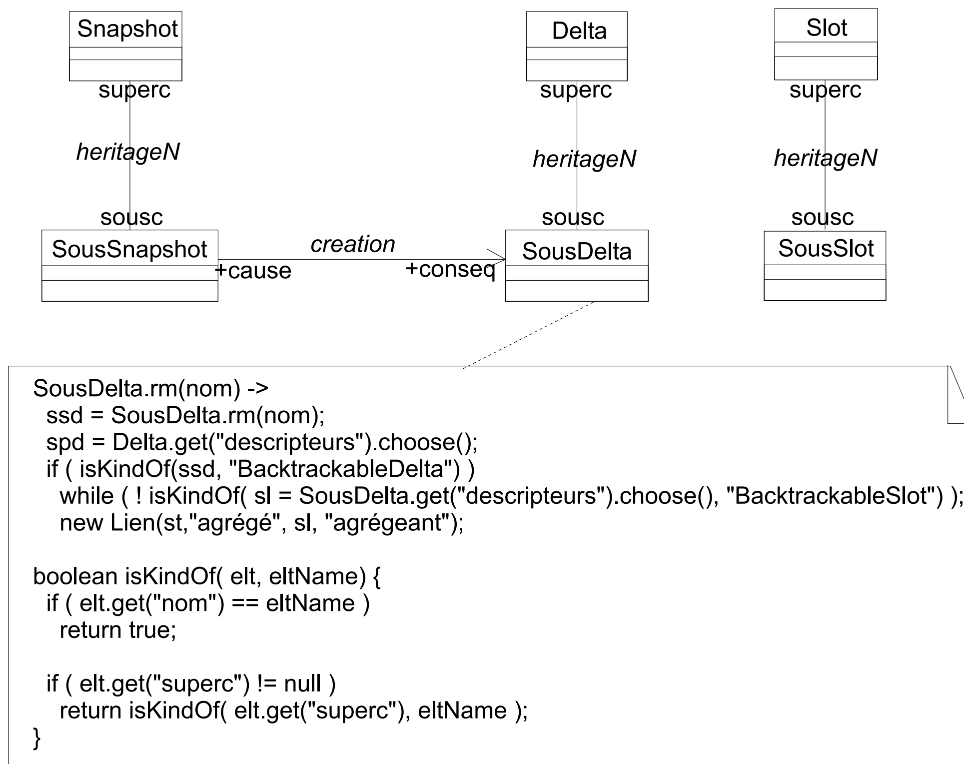


FIG. 9.8 – Représentation des dépendances, avec gestion du cas pour `BacktrackableSnapshot`.

Conclusion générale

Dans cette thèse nous avons eu pour objectif de mieux exprimer le modèle conceptuel des dépendances qui est sous-jacent à tout framework. Ce modèle conceptuel, dont l'étude a été menée en séparant son aspect structurel de son aspect comportemental a conduit à la création de deux modèles a priori indépendants dont nous rappelons les grands traits.

Bilan

Le premier modèle facilite l'utilisation d'un framework en en présentant explicitement les dépendances structurelles. Il véhicule des informations (capturées dès les phases de conceptions) permettant à l'utilisateur d'étendre le framework en respectant le couplage structurel existant entre les différents constituants de celui-ci.

Afin d'obtenir toute la souplesse requise pour l'expression de telles dépendances, la solution proposée opère sur une méta-description du framework. Faite sur demande lors du développement du framework, cette méta-description représente les dépendances sous une forme permettant vérification et aide à la construction d'une application. On regrettera seulement que celle-ci ne puissent pas être créée automatiquement, cependant on notera que l'expression des dépendances est faite à partir des bibliothèque de dépendance.

Le second modèle facilite la compréhension du couplage comportemental entre les composants d'un framework et cela sans en exposer le code source.

A ces fins, il introduit les concepts de point de vue client et de point de vue de composition qui décrivent le protocole de communication entre les composants. Il est à noter que nous avons émis certaines restrictions sur le modèle d'exécution des composants (introduction chapitre 7). Comme nous avons voulu notre modèle formel mais simple, la représentation des concepts est faite dans une notation graphique héritant du modèle synchrone. Grâce à ce formalisme, développeurs et utilisateurs du framework bénéficient d'un ensemble d'outils permettant de vérifier, entre autre, l'assemblage de classes ainsi que leur substituabilité. Cette vérification peut être statique ou dynamique.

Les bénéfices cumulés des deux modèles sont le respect de la structure et du comportement d'un framework, évitant des erreurs de compilation ou d'exécution difficiles à comprendre.

Grâce à eux est créé un continuum entre les documents produits par le développeur et ceux requis par l'utilisateur. Ces informations sont élaborées sans (trop) surcharger la tâche du développeur, et peuvent même l'aider.

Vers une réunion des modèles

Les travaux futurs concernant chacun de ces modèles ont été évoqués à la fin de leur présentation respective. Cependant, comme nous l'avons vu avec l'exemple de BLOCKS, les deux aspects que nous avons séparés sont inévitablement complémentaires. Ainsi, nous pensons qu'une piste complémentaire de recherche est la réunion des deux modèles. Nous avons imaginé trois rapprochements qui utilisent tous comme base le modèle structurel.

Rapprochement minimal, lien structure - SyncClass. Les points de vue comportementaux complètent naturellement la description structurelle du framework (niveau F0). Ainsi, afin d'être uniformément intégrés, les syncClass sont liés ontologiquement aux classes qu'ils décrivent par un lien `VueCliente` ou un lien `VueComposition`. Dans le contexte de BLOCKS, cela consisterait par exemple à attacher à la description de `Snapshot` son point de vue client.

Ce niveau d'intégration peut être considéré comme une simple documentation des classes. L'utilisateur pourra alors bénéficier de l'assistance offerte par le modèle des dépendances structurelles et aussi d'une aide simple lui indiquant certaines propriétés. Par exemple une dépendance structurelle comme l'héritage peut impliquer de vérifier la substituabilité comportementale.

Rapprochement intermédiaire, utilisation de la description du comportement pendant l'élaboration du framework. Alors que dans le rapprochement précédent les dépendances comportementales étaient utilisées dans la description, ce rapprochement les utilise à la fois au niveau de la description et de la méta-description du framework. Son utilisation principale serait à la fois de définir le comportement de certaines méthodes et de donner une méta-description du comportement des composants. Tout comme dans le rapprochement minimal cela permettrait alors de vérifier par exemple la substituabilité comportementale, et d'assurer qu'elle est toujours respectée lorsqu'on ajoute des méthodes.

Rapprochement complet, contrôle de l'instanciation. Les rapprochements précédents se limitaient au niveau de la description et de la méta-description du framework. Dans l'hypothèse où le système de dépendances structurelles serait utilisé pour contrôler la structure de l'instanciation, on pourrait si toutes les classes étaient munies de points de vue, vérifier l'adéquation des configurations ainsi créées sous réserve de la disponibilité de la description du point de vue de chaque classe.

Le premier rapprochement est le plus simple à obtenir, même si les deux autres sont souhaitables. Ainsi, quel que soit le résultat du rapprochement de ces modèles, le rapprochement minimal suffit à créer un outil offrant de nombreuses fonctionnalités supplémentaires par rapport aux outils existants.

Puisque ces modèles permettent la capture d'informations sur la structuration et le comportement des applications, il pourrait être intéressant de considérer l'utilisation de ces informations dans un système de rétro-conception. Elles indiqueraient ainsi la localisation des points de variation ainsi que le protocole des composants.

Notre modèle aboutit parfois à une représentation des dépendances qui peut sembler compliquée. Cependant, nous pensons que cette apparente complexité est la manifestation de la complexité intrinsèque des applications.

Enfin, malgré la restriction de notre étude au contexte des frameworks, nous pensons que cette expression des dépendances peut être intéressante hors de ce contexte. En effet bien qu'étant particulièrement présents dans les frameworks, les problèmes évoqués sont fréquemment rencontrés lors du développement classique d'applications.

Glossaire

Application dérivée : application créée à partir d'un framework*.

Bibliothèque (*library*) : ensemble de classes généralement prêtes à l'emploi, mais ne fournissant pas de boucle de contrôle.

Configuration : une configuration décrit l'ensemble d'instances et ses connexions utilisé pour faire des vérifications statiques dans le modèle des dépendances comportementales.

Component framework : framework* dont l'extension se fait par connexion de composants fournis (« s'oppose » à framework blanc*).

ConceptualElement : nom des éléments qui constituent la méta-description d'un framework* dans le modèle des dépendances structurelles*.

Dependance : nom de la classe désignant dans la méta-description du framework* les relations entre deux éléments.

Dépendance comportementale : décrit la dépendance qui résulte de l'entrelacement des appels de méthodes entre divers objets.

Dépendance existentielle : catégorie de dépendances structurelles* qui ne sont pas liées à la nature des éléments modélisés.

Dépendance ontologique : catégorie de dépendances structurelles* qui sont liées à la nature des éléments représentés.

Dépendance structurelle : dépendance existant entre les éléments constituant un framework* ou une application et dont le non-respect entraîne la non-validité du framework* ou de l'application.

Description du framework : décrit, dans le modèle structurel, la représentation d'un framework comme une sorte de graphe conceptuel.

Descripteur : élément constituant la description d'un framework*.

Élément conceptuel : élément constituant la méta-description du framework*.

F0 : nom donné à la description du framework*.

F1 : nom donné à la méta-description du framework*.

F2 : nom donné au modèle des dépendances structurelles*.

Factory : désigne l'objet qui fait la connexion entre la description du framework* et sa représentation physique.

Framework : *A framework is the skeleton of an application that can be customized by an application developer.* Système pré-fabriqué (Pierre Alain Müller).

Framework blanc : framework* dont l'utilisation se fait principalement par extension (i.e. dérivation). Son code source est accessible.

Framework gris : framework* qui pour être étendu « mixe » à la fois l'approche de framework noir* et blanc*.

Framework initial : désigne l'ensemble des classes qui constitue le framework sans application dérivée*.

Framework noir : framework* dont l'utilisation se fait principalement par composition et paramétrisation.

Glue : nom donné au code source qui connecte des composants.

Instanciation : désigne l'opération qui consiste en la création et la connexion d'instances à partir d'un framework*.

Interaction : concept que promeut le langage ISL*.

Interface de sortie : décrit l'ensemble des appels de messages sortant d'un composant.

ISL : Interaction Specification Language.

Lien : nom de la classe dont les instances résultent de la réalisation* d'une `Dependance`*.

Lien existentiel : élément résultant de la réalisation* d'une dépendance existentielle*.

Lien ontologique : élément résultant de la réalisation* d'une dépendance ontologique*.

Méta-description du framework : désigne la réification de la description d'un framework* (F1*).

Méta-framework : autre nom utilisé pour désigner la réification de la description d'un framework* (F1*).

OCL : Object Constraint Language. Langage de contraintes utilisé en UML.

Personnalisation : utilisée principalement pour les frameworks blancs*, cela désigne les opérations d'extension et de composition.

Point de variation (*hot-spot, axis of variation*) : nom désignant une classe ou un ensemble de classes à modifier afin de particulariser le framework*.

Point de vue client : vue du comportement qui décrit le protocole d'utilisation* et d'inter-dépendance* d'un composant.

Point de vue de composition : vue synthétique décrivant le protocole d'inter-dépendance* entre deux composants.

Point de vue comportemental : terme générique désignant à la fois les points de vue client* et de composition*.

Protocole d'inter-dépendance : désigne l'ordre dans lequel les messages de l'interface de sortie* d'un composant doivent être exécutés.

Protocole d'utilisation : désigne l'ordre dans lequel les messages de l'interface d'un composant doivent être appelés.

Réalisation conceptuelle : nom de la relation permettant de passer de la méta-description* à la description d'un framework* en respectant les dépendances structurelles*.

Règle de création : désigne la partie des dépendances structurelles* utilisée par l'opération de réalisation conceptuelle*.

Règle de vérification : désigne la partie des dépendances structurelles* utilisée par l'opération de vérification.

Schéma d'interactions : concept introduit par ISL* et réutilisé dans le modèle des dépendances structurelles* pour désigner un ensemble d'interactions.

Schéma de conception (*design pattern*) : un schéma de conception est la description nommée d'un problème de conception récurrent et de sa solution [GHJV95].

Schéma de vérifications : concept créé pour introduire une certaine forme de généricité dans les contraintes OCL utilisée dans le modèle des dépendances structurelles*.

SyncCharts : nom de la notation graphique dont dérivent les SyncClass. Ils sont une représentation graphique du langage Esterel.

SyncClass : nom de la notation graphique utilisée pour représenter les points de vue comportementaux*.

Annexe A

Présentation d'ISL

Le but de cette annexe est de présenter un peu plus en détail le fonctionnement du langage ISL. Le texte ci-dessous est extrait de l'article co-écrit par Anne-Marie Dery, Mireille Blay-Fornarino et Sabine Moisan et qui présente l'« apport des interactions pour la distribution de connaissances » [DBFM02].

« Nous proposons un modèle d'interactions qui permet de connecter des objets distant hétérogènes, sans modifier le code de leurs classes, en externalisant le code correspondant aux connexions.

Notre modèle est opérationnalisé par un langage de spécification de schémas d'interactions nommé ISL (Interaction Specification Language). Ce langage permet de décrire la connexion d'objets hétérogènes par des interactions n-aires, non-orientées. Il est basé sur la définition d'opérateurs (comme séquence, concurrence, etc.), dont la sémantique a été définie, et il s'inspire du langage d'interface IDL (Interface Défini-tion Language) défini par la norme CORBA.

La projection sur des langages cibles, comme C++ ou Java, est alors gérée par Dico*, un environnement de mise en oeuvre du concept d'interaction. Dico* réalise une réification des interactions entre objets et décharge l'utilisateur de la technologie réseau (déléguée à CORBA). Il est basé sur deux notions : schémas d'interaction et interactions. Au niveau des classes, les schémas d'interactions décrivent des connexions possibles entre classes. Au niveau des instances, les interactions instancient ces schémas pour représenter les connexions réelles entre objets. Le langage ISL offre une syntaxe assez simple pour décrire un schéma d'interaction sous forme de règles. Une règle ISL (voir exemple section 4.1) décrit comment le comportement d'un objet (c'est à dire la réception d'un message par cet objet) doit être modifié quand il interagit avec un autre. Dans un langage à objets, les comportements correspondent aux méthodes. Une règle modifie donc le comportement attendu d'une méthode. Une règle est composée d'une partie gauche qui décrit une réception de message (appelé message notifiant) et d'une partie droite, qui décrit le nouveau comportement à exécu-ter en utilisant le langage ISL. Par exemple, un attribut connecté à un objet graphique de trace devra à la réception du message setValue informer son représentant.

Grâce aux outils offerts par Dico*, il est possible de dynamiquement ajouter (on dira poser) ou retirer une interaction entre instances. De plus, dans le cadre du travail

collaboratif, quand plusieurs utilisateurs posent des interactions sur les mêmes objets, un mécanisme de fusion, assure la cohérence du comportement résultant.

Pour expliquer le concept et la mise en oeuvre d'interaction, prenons un exemple : un utilisateur veut connecter un attribut de type nombre flottant (comme l'attribut speed d'un objet MyCar instance d'une classe Car) avec une instance d'un objet graphique de type compteur (Speedometer). Il va alors poser une interaction entre les deux instances (l'attribut et le compteur). L'appel de la méthode setValue pour l'instance d'attribut va alors entraîner un comportement de trace (c'est à dire affecter la valeur et ensuite signaler la modification pour affichage au compteur).

Pour pouvoir poser cette interaction au niveau des instances, un schéma d'interaction doit avoir été défini auparavant. Dans notre exemple, le concepteur du système à base de connaissances a défini le schéma d'interactions suivant entre les classes Attribute<float> et Speedometer

```
Trace1(Attribute<float> attribut, Speedometer cpt) {
  attribut.setValue(val) -->
    attribut.setValue(val);cpt.update(val)
}
```

où `attribut.setValue(val)` est la partie gauche de la règle L'utilisateur peut instancier ce schéma, via l'environnement Dico*, pour poser l'interaction entre les instances `MyCar.speed` et `Cpt1` par appel à l'instruction : `instantiate("Trace1", MyCar.speed, Cpt1)`.

Ce même schéma peut être utilisé pour poser des interactions sur des attributs de différents objets (la vitesse d'autres voitures, aussi bien que, par exemple, l'attribut débit d'un compte en banque). »

Annexe B

Grammaire des parties actions

expression : := condition | sequence | block
condition : := "#" block [condition] | "#"
sequence : := expression " ;" expression
block : := trycatch | iteration | throw | method | parenth | forall
trycatch : := "[" expression "]" { catchBlock }
catchBlock : := "catch" exception "[" expression "]"
iteration : := "{" expression "}"
throw : := "throw" exception
forall : := "<" block ">"
method : := [name callKind] name "(" [param_list] ")"
callKind : := "." | "^"
param_list : := name ["," list]
name : := [a-zA-Z] [a-zA-Z0-9_]
parenth : := "(" expression ")"

Annexe C

Intégration des SyncClass à UML

Cette annexe présente l'intégration des SyncClass à UML. Elle est constituée de deux sections. La première présente le modèle des SyncClass et ses contraintes, et le seconde présente la connexion SyncClass / Classifier et ses contraintes.

C.1 Modèle UML des SyncClass

Comme nous l'indiquions sous-section 7.5.4 nous avons choisi d'ajouter les SyncClass au méta-modèle plutôt que d'essayer de remplacer les StateCharts. Ainsi, à la manière des spécifications d'UML [OMG01b][1] nous présentons tous les éléments, leur rôle et relation, avant de présenter les contraintes OCL.

SyncClass Un `SyncClass` représente les différents protocoles de l'élément auquel il est associé. Le protocole est modélisé par un graphe où les noeuds sont connectés par des transitions. Ces transitions sont traversées sur réception d'événements (appel de méthode ou levée d'exception). Un `SyncClass` contient un macrostate qui englobe les états et les transitions du SyncClass.
`relations` :
`top` : est un lien vers l'unique macro-état qui constitue un SyncClass au premier niveau.

MacroState Un `macrostate` est un état qui contient d'autres états et des transitions. Ces derniers sont contenus dans une `constellation`. Un `macrostate` est un artefact qui permet de représenter le parallélisme. En effet, s'il contient plus d'une constellation, alors elles s'exécutent de manière concurrente.
`relations` :
`constellations` : la liste des constellations qui constituent un macrostate.

[1]Contrairement à la spécification UML et dans un souci de compréhension, nous ne présentons pas nos éléments par ordre alphabétique.

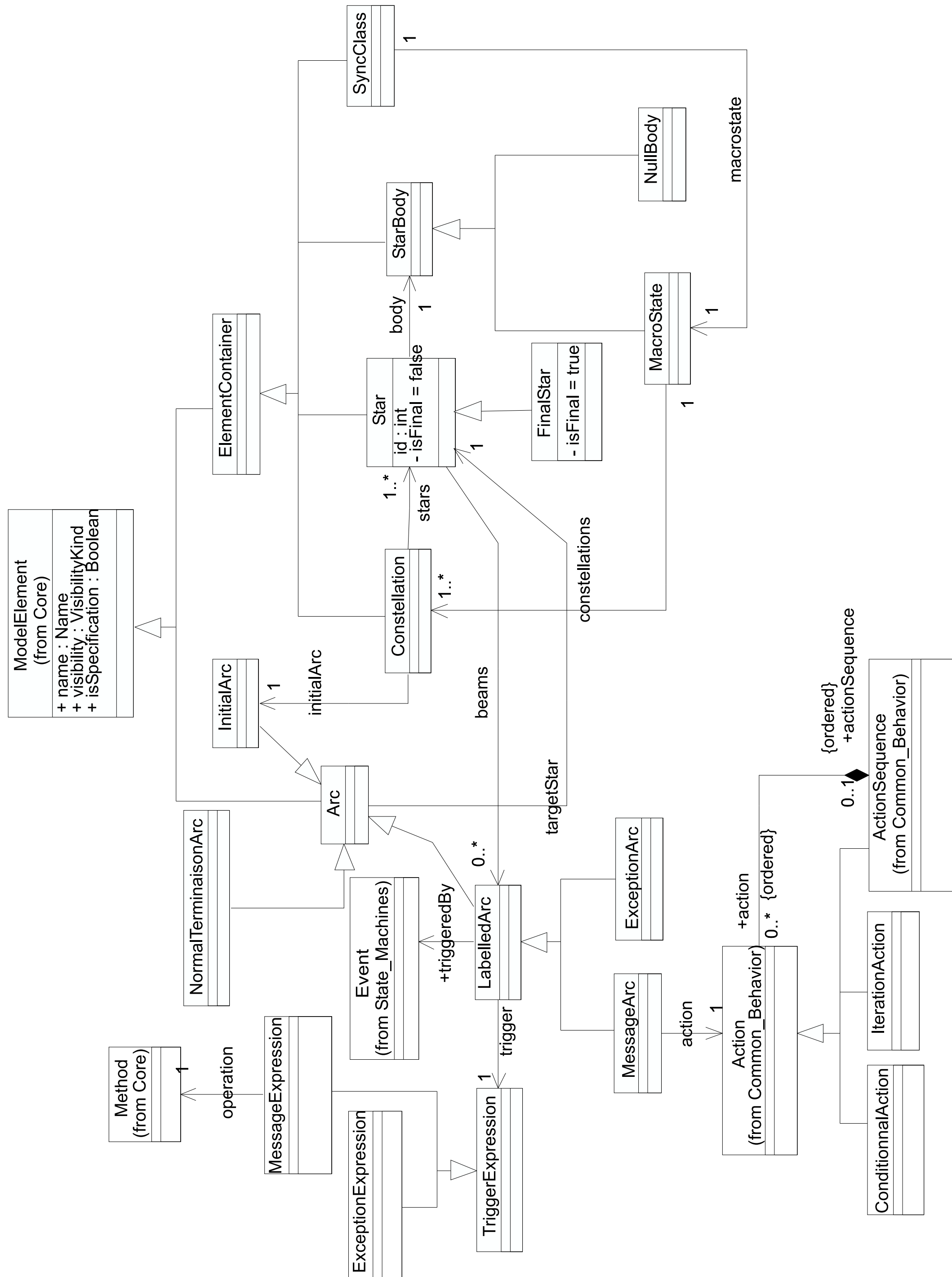


FIG. C.1 – Diagramme de classes des SyncClass.

Constellation Une `constellation` représente un ensemble d'états connectés par des transitions. Une `constellation` contient une unique transition initiale. Les états connectés sont représentés par une `Star`.
`relations` :
  `stars` : l'ensemble des étoiles de la constellation ;
  `initialArc` : l'arc initial de la constellation.

Star Une `Star` représente un état et ses transitions sortantes (« Une étoile rayonne, elle n'a que des rayons sortants » [And01]. Les transitions (rayons) sortant de la `Star` ne sont empruntables que lorsque le coeur de l'étoile (ou un de ses sous-états) sera l'état courant.
`relations` :
  `body` : référence le coeur de l'étoile ;
  `beams` : les transitions sortant de l'état.
`attributes` :
  `id` : le nom de l'étoile ;
  `isFinal` : indique si l'étoile est finale ou non.

FinalStar Une étoile finale représente un état final. Il signifie que la `constellation` est « terminée ». Attention des rayons peuvent malgré tout sortir de cet état.

StarBody Représente l'état lui-même. Cette classe abstraite est en fait la classe racine d'un schéma de conception composite (qui est composé de `MacroState`, `NullBody`, `Constellation` et `Star`).

NullBody Représente un état vide (graphiquement représenté par un rond vide), par opposition au macro-état qui contient d'autres états.

Arc Représente les transitions qui connectent les états. Un arc ne connaît que l'état vers lequel il mène (c'est d'ailleurs une Star).
`relations` :
  `targetStar` : l'étoile cible de l'arc.

InitialArc Représente un arc initial. Cet arc est spécial car il n'a pas d'extrémité de départ. Il est emprunté automatiquement lors de l'activation d'une constellation.

NormalTerminasonArc Représente les transitions de fin. Cette transition est empruntée automatiquement lorsque l'état d'où elle sort arrive dans un état final.

LabelledArc Représente une transition qui porte une étiquette. Cette étiquette indique l'action sur laquelle l'arc sera traversé.
`relations` :

`triggeredBy` : représente l'événement qui déclenchera la traversée de la transition ;
`trigger` : représente l'expression attendue pour déclencher la transition.

MessageArc Représente les transitions qui sont traversées sur l'occurrence d'un appel de méthode.
`relations` :
`action` : représente les actions qui sont exécutées lors de la traversée de la transition. Elle représente les parties actions des transitions. La classe `Action` ne sera pas détaillée car elle vient du méta-modèle UML. Les éléments fournis pas les sous-classes d'actions ne suffisant pas à nos besoins, et l'action semantics [OMG] n'étant pas encore adoptée par l'OMG et étant beaucoup trop lourde pour ce que nous voulions faire, nous avons préféré ajouter les classes `ConditionnalAction` et `IterationAction` afin de représenter ce qui nous manquait.

ExceptionArc Représente les transitions qui sont traversées sur la levée d'une exception.

TriggerExpression Représente l'expression de la partie gauche des transitions. C'est la partie qui indique quel sera l'événement attendu par la transition.

MessageExpression Représente la partie gauche de la transition lorsqu'il s'agit d'une méthode.
`relations` :
`operation` : une référence sur l'opération du `Classifier`.

ExceptionExpression Représente la partie gauche de la transition lorsqu'il s'agit d'une exception.

Les contraintes OCL

`ExceptionExpression` est forcément associé à un `ExceptionArc` :

```
context ExceptionExpression inv :
ExceptionArc.trigger.oclTypeOf(ExceptionArc)
```

`MessageExpression` :

```
context MessageExpression inv :
ExceptionArc.trigger.oclTypeOf(MessageArc)
```

Une seule normale terminaison ne sort d'une `Star` :

```
context Star inv :
Star.beams.select( b |
  b.oclKindOf(NormalTerminaisonArc) )->size() = 1
```

Une `Star` ne peut être connectée qu'à une autre `Star` du même niveau :

```
context Constellation inv :
 stars->forAll( s |
   stars.beams->forAll( b in stars.beams |
    stars.includes(b.targetStar) ) )
    and
    stars.includes(initialArc.targetStar)
```

Si dans une `Constellation` d'un macro-état il existe une étoile finale, alors toutes les constellations doivent avoir une étoile finale :

```
context MacroState inv :
 constellations->exists(c.stars->exists(s |
   isOclType(FinalStar) ) )
   implies
   constellations->forAll(c |
    c.stars->exists(s | isOclType(FinalStar) ) )
```

C.2 Modèle de la connexion SyncClass / Classifier

Cette section ne détaille pas la connexion SyncClass / Classifier puisqu'elle a déjà l'objet d'une discussion dans la sous-section 7.5.4. Nous nous focaliserons sur les contraintes de cette intégration.

Tous les éléments spécifiés par un SyncClass sont du même type.

```
context SyncClass inv :
self.describeConnectionWith->forAll( sc |
 sc.type.equals(self.specify.type) )
```

Les méthodes utilisées comme déclencheur dans un SyncClass doivent appartenir aux `Method` du `Classifier` que le SyncClass spécifie. La méthode `getMethods` retourne l'ensemble des méthodes (instance de la classe `Method`) qui apparaissent dans les `TriggerExpression` d'un SyncClass.

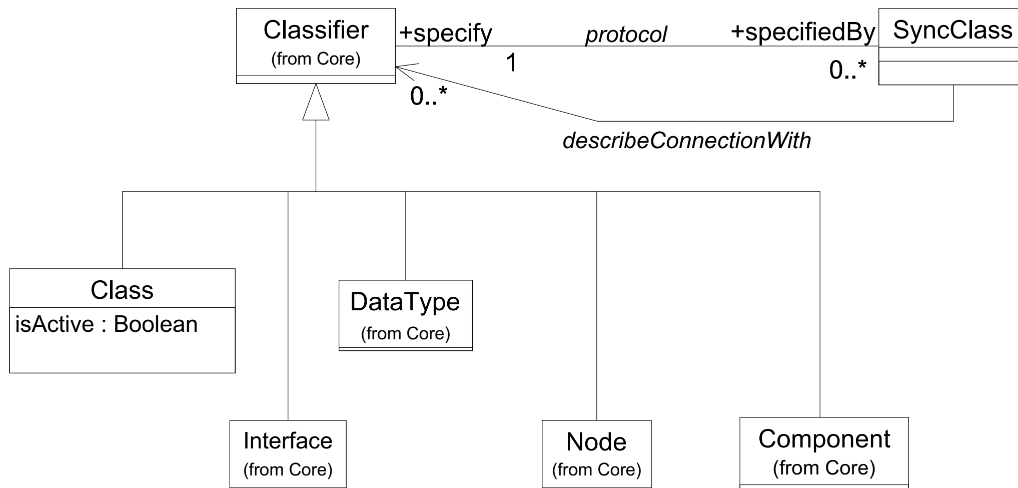


FIG. C.2 – Liaison SyncClass / UML.

```
context SyncClass inv :
  SyncClass.getMethods()->forAll( m |
   SyncClass.specify.features.includes(m))
```

Les méthodes qui constituent les actions des transitions doivent appartenir aux `Method` des `Classifier` dont le SyncClass représente la connexion. On suppose que l'on bénéficie d'une méthode `getSentMessages()` qui retourne tous les messages appelés.

```
context SyncClass inv :
    SyncClass.getSentMessages()->forAll( m |
      SyncClass.describeConnectionWith->exists( aClassifier |
       aClassifier.features.includes(m) ))
```

Il ne peut pas y avoir plus de points de vue que de classes avec lequel il y a une dépendance, une association ou une variable plus un (le point de vue client) :

```
context Classifier inv :
Classifier.protocol.size() <=
  Classifier.clientDependencies.select( c |
   c.oclTypeOf(Usage) )->size() +
  Classifier.participant->size() +
  Classifier.structuralFeature->size() + 1
```

Le `receiver` doit référencer un nom de relation ou de variable :

```
context MessageArc inv :
receiver.method.type.associationEnd->includes( association |
  association.name.equals(receiver) ) or
  receiver.method.type.clientDependency->includes( dep |
   dep.name.equals(receiver) ) or
  receiver.method.type.structuralFeatures->includes( sf |
   sf.name.equals(receiver) )
```

Bibliographie

[AB01] Gabriela Arevalo et Isabelle Borne. Architectural description of object oriented frameworks : An approach. Dans *Langages et Modèles à Objets - L'Objet*, volume 7, pages 183–198, 01 2001.

[ABP+97] Charles André, Frédéric Boulanger, Marie-Agnès Péraldi, Jean-Paul Rigault, et Guy Vidal-Naquet. Objects and synchronous programming. *European Journal of Automation*, 31(3) :417–432, 1997.

[AC98] Ellen Agerbo et Aino Cornils. How to preserve the benefits of design patterns. Dans *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 de *ACM SIGPLAN Notices*, pages 134–143, New York, Octobre 18–22 1998. ACM Press.

[AG97] Robert Allen et David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, Juillet 1997.

[Agu00] Ademar Aguiar. A minimalist approach to framework documentation. Dans *OOPSLA Companion - Doctoral Symposium*, October 2000.

[Alh98] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly, Sebastapol, CA, 1998.

[And96a] Charles André. Representation and analysis of reactive behaviors : a synchronous approach. Dans *CESA*, pages 19–29, july 1996.

[And96b] Charles André. Synccharts a visual representation of reactive behaviors. Rapport technique TR95-52, I3S - CNRS, Avril 1996.

[And01] Charles André. Communication personnelle, 2001.

[arg] Argo-uml. http ://argouml.tigris.org/.

[BB90] Gérard Berry et Gérard Boudol. The chemical abstract machine. Dans ACM, éditeur, *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA*, pages 81–94, New York, NY, USA, 1990. ACM Press.

[BD99] Greg Butler et Pierre Dénommée. Documenting frameworks. Dans *Building Application Frameworks*, chapitre 21, pages 495–503. Wiley, 1999.

[Bec99] Kent Beck. *Extreme Programming Explained : Embrace Change*. Addison-Wesley, 1999.

[Ber00a] Gérard Berry. The esterel language primer. Rapport technique, Ecole des Mines CMA / INRIA Sophia, July 2000.

[Ber00b] Gérard Berry. *Proof, Language and Interaction : Essays in Honour of Robin Milner*, chapitre The foundations of Esterel. MIT Press, 2000.

[Ber01] Laurent Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*. Thèse de doctorat, Université de Nice, Octobre 2001.

[BFJR98] John Brant, Brian Foote, Ralph E. Johnson, et Donald Roberts. Wrappers to the rescue. Dans Eric Jul, éditeur, *ECOOP '98—Object-Oriented Programming*, volume 1445 de *Lecture Notes in Computer Science*, pages 396–417. Springer, 1998.

[BG92] Gérard Berry et Georges Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, Novembre 1992.

[BG98] Kent Beck et Erich Gamma. Test infected : Programmers love writing tests. *Java Report*, 3(2) :37–50, 1998.

[BJ94] Kent Beck et Ralph Johnson. Patterns Generate Architectures. Dans M. Tokoro et R. Pareschi, éditeurs, *Proc. European Conf. on Object Oriented Programming (ECOOP)*, volume 821 de *Lecture Notes in Computer Science*, pages 139–149, Bologna, Italy, Juillet 1994. Springer-Verlag, Berlin.

[BMDT96] Amard Bouali, Jean-Paul Marmorat, Robert De Simone, et H. Toma. Verifying synchronous reactive systems programmed in ESTEREL. *Lecture Notes in Computer Science*, 1135 :463–483, 1996.

[Boh98] K. A. Bohrer. Architecture of the San Francisco frameworks. *IBM Systems Journal*, 37(2) :156–169, 1998.

[Bou97] Amard Bouali. Xeve : An esterel verification environment (version v1.3). Rapport technique RT-214, INRIA, October 1997.

[BR99] Isabelle Borne et Nicolas Revault. Comparaison d'outils de mise en oeuvre de design patterns. *L'objet*, 5(2), 1999.

[Bra95] John M. Brant. HotDraw. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[BRJ99] Grady Booch, James Rumbaugh, et Ivar Jacobson. *The Unified Modeling language User Guide*. Addison-Wesley, 1999.

[BS86] Gérard Berry et Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1) :117–126, 1986.

[Car90] John M. Carroll. *The Nurnberg Funnel : Designing Minimalist Instruction for Practical Computer Skill*. MIT Press, 1990.

[CE00] Krzysztof Czarnecki et Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[CH74] Roy. H. Campbell et Arie N. Habermann. The Specification of Process Synchronization by Path Expressions. Dans *Proc. Int. Symp. on Operating Systems*, volume 16 de *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, Berlin, 1974.

[Chi00] Shigeru Chiba. Load-time structural reflection in java. Dans Elisa Bertino, éditeur, *ECOOP 2000, European Conference on Object-Oriented Programming, Sophia Antipolis and Cannes, France*, volume 1850 de *Lecture Notes in Computer Science*, pages 313–336, New York, NY, 2000. Springer-Verlag.

[CLL99] Peter Coad, Eric LeFebvre, et Jeff De Luca. *Java Modeling in Color with UML*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1999.

[DBFM02] Anne-Marie Dery, Mireille Blay-Fornarino, et Sabine Moisan. Apport des interactions pour la distribution des connaissances. *L'OBJET*, 2002. A paraître.

[DEMN98] Roland Ducournau, Jérôme Euzenat, Gérald Masini, et Amedeo Napoli. *Langages et Modèles à Objets*. INRIA, 1998.

[DK76] Frank DeRemer et Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2) :80–6, Juin 1976.

[DMNS97] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, et Patrick Steyaert. Design guidelines for tailorable frameworks. *Communications of the ACM*, 40(10) :60–64, Octobre 1997.

[dmp] Declarative meta programming. progwww.vub.ac.be/poolresearch/dmp/.

[eAPSZ01] Fernando Brito e Abreu, Geert Poels, Houari A. Sahraoui, et Horst Zuse, éditeurs. *Quantitative approaches in object-oriented software engineering (L'Objet)*, volume 7. Hermes, 2001.

[EYG97] A. Eden, A. Yehudai, et J. Gil. Precise specification and automatic application of design patterns. Dans *1997 International Conference on Automated Software Engineering*, pages 143–152. IEEE Computer Society Press, 1997.

[Fay99] Mohamed Fayad. Future trends. Dans *Building Application Frameworks*, chapitre 26, pages 617–619. Wiley, 1999.

[FHLS97] Garry Froehlich, H. James Hoover, Ling Liu, et Paul Sorenson. Hooking into object-oriented application frameworks. Dans *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 491–502, NY, Mai 17–23 1997. ACM.

[FHLS98] Garry Froehlich, H. James Hoover, Luyuan Liu, et Paul Sorenson. Requirements for a hooks tool. http ://www.cs.ualberta.ca/ softeng/papers/ssr04.pdf, 1998.

[FMvW97] Gert Florijn, Marco Meijers, et Pieter van Winsen. Tool support for object-oriented patterns. Dans Mehmet Akşit et Satoshi Matsuoka, éditeurs, *ECOOP'97—Object-Oriented Programming*, volume 1241 de *Lecture Notes in Computer Science*, pages 472–495. Springer, 1997.

[Fow97] Martin Fowler. *UML Distilled*. Addison-Wesley, 1997.

[FPR00a] Marcus Fontoura, Wolfgang Pree, et Bernhard Rumpe. Framework development an adaptation with uml-f - the uml profile for framework architectures. Tutorial Notes, ECOOP 2000, 2000.

[FPR00b] Marcus Fontoura, Wolfgang Pree, et Bernhard Rumpe. UML-F : A modeling language for object-oriented frameworks. Dans E. Bertino, éditeur, *Proceedings of ECOOP 2000*, volume 1850 de *LNCS*, pages 63–82. Springer, 2000.

[FSJ99] Mohamed Fayad, Douglas Schmidt, et Ralph Johson. Application frameworks. Dans *Building Application Frameworks*, chapitre 1, pages 3–28. Wiley, 1999.

[Gar01] David Garlan. *Encyclopedia of Software Engineering*, chapitre Software Architecture. Wiley, 2001.

[GB] Erich Gamma et Kent Beck. Junit a cook's tour. Fournit avec JUnit.

[GH99] Angelo Gargantini et Constance Heitmeyer. Using model checking to generate tests from requirements specifications. Dans Oscar Nierstrasz et Michel Lemoine, éditeurs, *ESEC/FSE '99*, volume 1687 de *Lecture Notes in Computer Science*, pages 146–162. Springer-Verlag / ACM Press, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1995.

[GK98] Joseph (Yossi) Gil et Stuart Kent. Three dimensional software modelling. Dans *Proceedings of the 1998 International Conference on Software Engineering*, pages 105–114. IEEE Computer Society Press / ACM Press, 1998.

[GK00] David Garlan et Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. Dans Andy Evans, Stuart Kent, et Bran Selic, éditeurs, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 de *LNCS*, pages 498–512. Springer, 2000.

[GMW97] David Garlan, Robert T. Monroe, et David Wile. Acme : An architecture description interchange language. Dans *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[Gro98] Object Management Group. XML Metadata Interchange (XMI). Rapport technique ad/98-10-05, Object Management Group, Février 1998.

[Hal91] Nicolas Halbwachs. Conception de systèmes réactifs. les langages synchrones. Rapport technique TR95-52, IMAG, October 1991.

[Har87] David Harel. Statecharts : A visual formalism for complex system. *Science of Computer Programming*, 8(3) :231–274, 1987.

[Har00] David Harel. On the behavior of complex object-oriented systems. Dans *UML 2000*, pages 324–329. Springer-Verlag, 2000.

[HHG90] Richard Helm, Ian M. Holland, et Dipayan Gangopadhyay. Contracts : Specifying behavioural compositions in object-oriented systems. Dans *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, Octobre 1990. Published as Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, volume 25, number 10.

[HHK+01] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, et Jukka Viljamaa. Generating application development environments for java frameworks. Dans *Generative and Component-Based Software Engineering*, LNCS. Springer-Verlag, 2001.

[HHT+99] Markku Hakala, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, Jukka Viljamaa, Jukka Paakki, et Kai Koskimies. Managing object-oriented frameworks with specialization templates. Dans *ECOOP Workshop Reader - Workshop on Object technology for Product-line Architectures*, 1999.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, Aaout 1978.

[Hol92] Ian M. Holland. Specifying reusable components using contracts. Dans Ole Lehrmann Madsen, éditeur, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 de *Lecture Notes in Computer Science*, pages 287–308, New York, NY, 1992. Springer-Verlag.

[IT94] ITU-T. Z.120 message sequence chart (msc), 1994.

[IWY00] Paola Inverardi, Alexander L. Wolf, et Daniel Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3) :239–272, Juillet 2000.

[JBR99] Ivar Jacobson, Grady Booch, et James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[JMG97] Christophe Gransart et Philippe Merle Jean-Marc Geib. *CORBA : Des concepts à la pratique*. Masson Editeur, Masson, France, 1997.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. Dans *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, Octobre 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[Joh93] Ralph Johnson. How to design frameworks. Notes for OOPSLA 93, October 1993.

[Joh97a] Ralph E. Johnson. Components, frameworks, patterns. Dans *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[Joh97b] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10) :39–42, Octobre 1997.

[Jol99] Art Jolin. Usability and framework design. Dans *Building Application Frameworks*, chapitre 6, pages 153–162. Wiley, 1999.

[KKS96] Nils Klarlund, Jari Koistinen, et Michael I. Schwartzbach. Formal design constraints. Dans *OOPSLA '96 Conference Proceedings : Object-Oriented Programming Systems, Languages, and Applications*, pages 370–383. ACM Press, 1996.

[Kwo00] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. Dans Andy Evans, Stuart Kent, et Bran Selic,

éditeurs, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 de *LNCS*, pages 528–540. Springer, 2000.

[Lam93] John Lamping. Typing the specialization interface. Dans *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 201–214, 1993.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, et Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, Avril 1995.

[LM95] Doug Lea et Jos Marlowe. Interface-based protocol specification of open systems using PSL. Dans Walter G. Olthoff, éditeur, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 de *Lecture Notes in Computer Science*, pages 374–398, Åarhus, Denmark, 7–11 Aaout 1995. Springer.

[Lor93] Mark Lorenz. *Object-Oriented Software Development*. Prentice Hall, Englewood Cliffs, 1993.

[LP99] Johan Lilius et Ivan Porres Paltor. Formalising UML state machines for model checking. Dans Robert France et Bernhard Rumpe, éditeurs, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 de *LNCS*, pages 430–445. Springer, 1999.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12) :147–155, Décembre 1987.

[Mar90] Florence Maraninchi. *ARGOS : un langage pour la conception, la description et la validation des systèmes réactifs*. Thèse de doctorat, Université Joseph Fourier, Grenoble I, France, 1990.

[MB97] Michael Mattsson et Jan Bosch. Framework composition : Problems, causes and solutions. Dans *Proceedings of TOOLS USA '97*, Juillet 1997.

[McI68] D. McIlroy. Mass-produced software components. Dans *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[MCK97] Matthias Meusel, Krzysztof Czarnecki, et Wolfgang Köpf. A model for structuring user documentation of object-oriented frameworks using patterns and hypertext. Dans Mehmet Akşit et Satoshi Matsuoka, éditeurs, *ECOOP'97—Object-Oriented Programming*, volume 1241 de *Lecture Notes in Computer Science*, pages 496–510. Springer, 1997.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[Mic] Sun Microsystems. Enterprise javabeans. http ://java.sun.com.

[Mic97a] Sun MicroSystems. The abstract window toolkit, 1997. http ://java.sun.com/products/jdk/awt/.

[Mic97b] Sun Microsystems. Javabeans. http ://java.sun.com, July 1997.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, Berlin, 1 edition, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Min96] Naftaly H. Minsky. Law-governed regularities in object systems. part 1 : An abstract model. *Theory and Practice of Object Systems*, 2(4) :283–301, 1996.

[MN96] Simon Moser et Oscar Nierstrasz. The effect of object-oriented frameworks on developer productivity. *IEEE Computer*, 9 :45–51, Septembre 1996.

[MQR95] Mark Moriconi, Xiaolei Qian, et R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4) :356–372, Avril 1995.

[MRR01] Sabine Moisan, Annie Ressouche, et Jean-Paul Rigault. BLOCKS, a Component Framework with Checking Facilities for Knowledge-Based Systems. *Informatica, Special Issue on Component Based Software Development*, 25(4), 2001.

[MS99] Hafedh Mili et Houari Sahraoui. Describing and using object frameworks. Dans *Building Application Frameworks*, chapitre 23, pages 523–561. Wiley, 1999.

[Mul97] Pierre-Alain Muller. *Modelisation Object avec UML*. Eyrolles, 1997.

[Nie93] Oscar Nierstrasz. Regular types for active objects. Dans *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 1–15, Octobre 1993.

[Nie95] Oscar Nierstrasz. Regular types for active objects. Dans Oscar Nierstrasz et Dennis Tsichritzis, éditeurs, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[NM95] Oscar Nierstrasz et Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2) :262–264, Juin 1995.

[OCM00] Alvaro Ortigosa, Marcelo Campo, et Roberto Moriyõn. Towards agent-oriented assistance for framework instantiation. Dans *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, volume 35.10 de *ACM Sigplan Notices*, pages 253–263, N. Y., Octobre 15–19 2000. ACM Press.

[OMG] OMG. Uml action semantics consortium. http ://www.umlactionsemantics.org/.

[OMG97] OMG. Object constraint language specification, Sep 1997.

[OMG01a] OMG. Corba component model, 2001. http ://www.omg.org/cgi-bin/doc ?ptc/2001-11-03.

[OMG01b] Object Management Group OMG. Omg unified modeling language specification. URL : http ://www.omg.org/, February 2001. version 1.4 (draft).

[OTI01] OTI. Eclipse. http ://www.eclipse.org, 2001.

[PDN86] R. Prieto-Diaz et J. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4) :307–334, Novembre 1986.

[Pin95] Xavier Pintado. Gluons and the cooperation between software components. Dans Oscar Nierstrasz et Dennis Tsichritzis, éditeurs, *Object-Oriented Software Composition*, pages 321–349. Prentice-Hall, 1995.

[PPSS95] Wolfgang Pree, Gustav Pomberger, Albert Schappert, et Peter Sommerlad. Active guidance of framework development. *Software–Concepts and Tools*, 16 :94–103, 1995.

[Pre94] Wolfgang Pree. Meta Patterns - A Means to Capturing the Essentials of Reusable Object Oriented Design. Dans M. Tokoro et R. Pareschi, éditeurs, *Proc. European Conf. on Object Oriented Programming (ECOOP)*, volume 821 de *Lecture Notes in Computer Science*, pages 150–162, Bologna, Italy, Juillet 1994. Springer-Verlag, Berlin.

[Pre99] Wolfgang Pree. Hot-spot-driven development. Dans *Building Application Frameworks*, chapitre 16, pages 379–393. Wiley, 1999.

[Pre00] Wolgang Pree. A l'occasion du "workshop on methods and tools for object-oriented framework development" - oopsla. Communication personnelle, October 2000.

[Pro02] RAINBOW Project. Noah. http ://www.essi.fr/ rainbow, 2002.

[RF00] Pascal Rapicault et Mireille Fornarino. Instanciation et vérification de patterns de conception : un méta-protocole. Dans *Langages et Modèles à Objets*, 2000.

[RJ96] Don Roberts et Ralph Johnson. Evolving frameworks : A pattern language for developing object-oriented frameworks. Dans *Proceedings of Pattern Languages of Programs*, 1996.

[RK96] Bernhard Rumpe et Cornel Klein. Specification of behavioral semantics in object-oriented information modeling. chapitre Automata Describing Object Behavior, pages 265–286. Kluwer Academic Publishers, 1996.

[RL00] Clyde Ruby et Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. Dans *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) de *ACM SIGPLAN Notices*, pages 208–228, Octobre 2000.

[RN01] Pascal Rapicault et Amedeo Napoli. Evolution d'une hiérarchie de classes par interclassement. Dans *Langages et Modèles à Objets*, 2001.

[Rüp99] Andreas Rüping. Managing class dependencies. Dans *Building Application Frameworks*, chapitre 14, pages 325–344. Wiley, 1999.

[RR01] Pascal Rapicault et Jean-Paul Rigault. Open implementation of uml meta-model(s) making meta-modeling and meta-programming meet. Dans *Reflection - Poster*, 2001.

[SBF96] Steve Sparks, Kevin Benner, et Chris Faris. Managing object-oriented framework reuse. *Computer*, 29(9) :52–61, Septembre 1996.

[SG99] Joãn Pedro Sousa et David Garlan. Formal modeling of the enterprise JavaBeans component integration framework. Dans *Proceedings of FM'99*, LNCS 1709, Toulouse, France, Septembre 1999. Springer Verlag.

[SGW94] Bran Selic, Garth Gullekson, et Paul T. Ward. *Real-Time Object Oriented Modeling*. John Wiley & Sons, 1994.

[SLJ00] Gerson Sunyé, Alain Le Guennec, et Jean-Marc Jézéquel. Design patterns application in UML. Dans Elisa Bertino, éditeur, *Proc. 14th ECOOP : Object-Oriented Programming, Sophia Antipolis and Cannes, France*, volume 1850 de *LNCS*, pages 44–62. Springer-Verlag, 2000.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, et Theo D'Hondt. Reuse contracts : Managing the evolution of reusable assets. Dans *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 de *ACM SIGPLAN Notices*, pages 268–285, New York, Octobre 6–10 1996. ACM Press.

[Sou99] Neelam Soundarajan. Understanding frameworks. Dans *Building Application Frameworks*, chapitre 12, pages 289–308. Wiley, 1999.

[SP98] Ricardo Silva et Roberto Price. Tool support for helping the use of frameworks. Dans *XVIII International Conference of the Chilean Computer Science Society*. IEEE, 1998.

[SSP95] Albert Schappert, Peter Sommerlad, et Wolfgang Pree. Automated support for software development with frameworks. Dans M. H. Samadzadeh et Mansour K. Zand, éditeurs, *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, pages 123–127, 1995.

[Szy98] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[TAL95] TALIGENT. Leveraging object-oriented frameworks, 1995. White paper.

[Tec] Esterel Technologies. Esterel studio. http ://www.esterel-technologies.com.

[Vau97] Sylvain Vauttier. *Une étude de la modélisation du comportement des objets composites*. Thèse de doctorat, Université de Nimes, 1997.

[WGM88] Andre Weinand, Erich Gamma, et Rudolf Marty. ET++ an object oriented application framework in C++. *ACM SIGPLAN Notices*, 23(11) :46–57, Novembre 1988.

[WMB99] Axel Wienberg, Florian Matthes, et Marko Boger. Modeling dynamic software components in UML. Dans Robert France et Bernhard Rumpe, éditeurs, *UML'99 - The Unified Modeling Language. Proceedings of the Second International Conference. Fort Collins, Colorado, USA*, volume 1723 de *Lecture Notes in Computer Science*, pages 204–219. Springer-Verlag, Octobre 1999.

[Wuy01] Roel Wuyts. *A Logic Meta - Programming Approach to Support the Co - Evolution of Object - Oriented Design and Implementation*. Thèse de doctorat, Vrije Universiteit Brussel, 2001.

[WV01] Bart Wydaeghe et Wim Vanderperren. Visual component composition usiing composition patterns. Dans *TOOLS*, pages 120–129, 2001.

[WZ88] Peter Wegner et Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. Dans S. Gjessing et K. Nygaard, éditeurs, *ECOOP '88, European Conference on Object-Oriented Programming, Oslo, Norway*, volume 322 de *Lecture Notes in Computer Science*, pages 55–77, New York, NY, Aaout 1988. Springer-Verlag.

[XRK00] Spyros Xanthakis, Pascal Régnier, et Constantin Karapoulios. *Le test des logiciels*. Hermes Sciences, 01 2000.

[YS94] Daniel M. Yellin et Robert E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. Dans *OOPSLA '94 Conference Proceedings : Object-Oriented Programming Systems, Languages, and Applications*, pages 176–190. ACM Press, 1994.

[ZW97] Amy Moormann Zaremski et Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4) :333–369, Octobre 1997.

Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation

Résumé

L'utilisation d'un framework pose des problèmes liés au respect de la structure et du comportement de celui-ci. Ces problèmes sont la conséquence de la perte des informations de conception lors de l'implémentation, ce qui se traduit par l'absence d'une documentation pertinente lors de l'utilisation. Ainsi le but de cette thèse est de fournir un modèle d'expression des dépendances structurelles et comportementales. Ces modèles sont intégrés au cycle de vie du framework, de la spécification jusqu'à la finalisation.

Le modèle de dépendances structurelles propose une expression explicite des dépendances grâce à une réification partielle du framework. Ce modèle, indépendant de tout langage permet au développeur du framework aussi bien qu'à un utilisateur, de consulter les dépendances entre les éléments du framework, et de bénéficier d'une aide dynamique ainsi que d'un système de vérification.

Le modèle de dépendances comportementales étend l'interface statique des classes par une spécification dynamique (protocole) de celles-ci. Cette spécification définit les séquences valides d'enchaînement de messages entrant et sortant de la classe. Cette spécification, basée sur le modèle formel d'Esterel, permet des vérifications statiques et dynamiques.

Mots-clefs : Framework, méta-modélisation, dépendance, vérification

---

Models and techniques to specify, develop et use a framework : a meta-modeling approach

Abstract

The use of a framework poses problems related to the respect of its structure and behavior. These problems are the consequence of the loss of design time information during implementation, which results in the absence of a relevant documentation at re-use time. The goal of this thesis is to provide models to express behavioral and structural dependencies. These models are integrated into the framework life cycle from the specification to the finalization.

The model of structural dependencies proposes an explicit expression of dependencies through a partial reification of the framework. Independent of any language, it provides assistance and verification capabilities, and it is accessible by framework developpers and users.

The model of behavioral dependencies augments the static interface of classes with their dynamic specification (protocol). This specification expresses the valid sequences of the class incoming and outgoing messages. This specification, based on the formal model of Esterel, allows static and dynamic checks.

Keywords : Framework, meta-modeling, dependency, verification