Vérification et correction des spécifications B : application à l'assemblage de composants

Inès Mouakher Abdelmoula

Université Nancy 2 — Université de Tunis El-Manar
UFR mathématiques et informatique — Faculté des Sciences de Tunis

École doctorale IAEM Lorraine, Département de formation doctorale en informatique
École doctorale en informatique

Vérification et correction des spécifications B : application à l'assemblage de composants

THÈSE

présentée et soutenue publiquement le 27 Novembre 2010

pour l'obtention du

Doctorat de l'université Nancy 2
Doctorat de l'université de Tunis El-Manar
(spécialité informatique)

par

Inès MOUAKHER ABDELMOULA

Composition du jury

Président :	Yahia SLIMANI, Professeur, Université de Tunis El-Manar
Rapporteurs :	J. Christian ATTIOGBÉ, Professeur, Université de Nantes Samir BEN AHMED, Professeur, Université 7 novembre à Carthage
Examinateur :	Olga KOUCHNARENKO , Professeur, Université de de Franche-Comté
Directeurs de thèse :	Khaled BSAÏES, Professeur, Université de Tunis El-Manar Jeanine SOUQUIÈRES, Professeur, Université Nancy 2
Co-encadreur :	Francis ALEXANDRE, Maître de conférences, Université Henri-Poincaré

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Unité de Recherche en Programmation, Algorithmique et Heuristiques — Code 99/UR/14-04

Mis en page avec la classe thloria.

Table des matières

1 Introduction

1.1 Contexte . . . 1
1.1.1 Méthode B . . . 1
1.1.2 Approche CBSE . . . 3
1.2 Contributions . . . 4
1.2.1 Raffinement des systèmes de transitions étiquetés en B . . . 4
1.2.2 Aide à la construction de spécifications B à partir de l'échec de la preuve . 4
1.2.3 Application à une approche CBSE . . . 5
1.3 Plan de la thèse . . . 6

2 La méthode B

2.1 Introduction . . . 9
2.2 Fondement de la méthode B . . . 10
2.2.1 Principe du développement en B . . . 10
2.2.2 Les substitutions généralisées . . . 11
2.2.3 Machine abstraite . . . 14
2.2.4 Raffinement . . . 16
2.3 Modularité en B . . . 17
2.3.1 Les clauses **INCLUDES** et **IMPORTS** . . . 17
2.3.2 Les clauses **PROMOTES** et **EXTENDS** . . . 18
2.3.3 Les clauses **SEES** et **USES** . . . 18
2.4 Obligations de preuve . . . 18
2.4.1 Obligations de preuve associées à l'instanciation d'une machine . . . 20
2.4.2 Obligations de preuve associées aux assertions . . . 20
2.4.3 Obligations de preuve associées aux initialisations . . . 21
2.4.4 Obligations de preuve associées aux opérations . . . 21
2.5 Brève présentation du prouveur de l'atelier B . . . 22
2.5.1 Obligations de preuves élémentaires . . . 22

2.5.2 Le prouveur automatique . . . 23
2.5.3 Le prouveur interactif . . . 23
2.6 Détection, analyse et correction des erreurs dans une spécification B . . . 24
2.6.1 Détection des erreurs : échec de la preuve . . . 24
2.6.2 Analyse des erreurs . . . 25
2.6.3 Diagnostic et correction des spécifications B . . . 26
2.7 Conclusion . . . 32

3 Adaptation logicielle statique

3.1 Introduction . . . 33
3.2 Interopérabilité . . . 34
3.2.1 Objets, composants, services et interfaces . . . 34
3.2.2 Niveaux d'interopérabilité . . . 35
3.2.3 Incompatibilités . . . 38
3.3 Quelques approches d'adaptation . . . 38
3.3.1 Assemblage des composants . . . 39
3.3.2 Adaptation des services . . . 44
3.3.3 Recherche, composition et adaptation des composants . . . 47
3.3.4 Composition et adaptation des connecteurs . . . 50
3.3.5 Discussion . . . 51
3.4 Caractéristiques d'une approche d'adaptation . . . 51
3.4.1 Niveaux d'interopérabilité . . . 51
3.4.2 Spécification . . . 53
3.4.3 Spécification semi-automatique vs. manuelle . . . 55
3.4.4 Types d'adaptation . . . 56
3.4.5 Processus de construction de l'adaptateur . . . 57
3.5 Conclusion . . . 58

4 Raffinement des systèmes de transitions étiquetés

4.1 Introduction . . . 59
4.2 Définitions . . . 60
4.2.1 Système de transitions étiqueté (STE) . . . 60
4.2.2 Système de transitions étiqueté gardé . . . 61
4.2.3 Propriétés des systèmes de transitions étiquetés . . . 62
4.2.4 Composition d'étiquettes . . . 63
4.3 Traduction en B . . . 65
4.3.1 Système de transitions étiqueté . . . 65

4.3.2 Relation entre les états de deux STEs . . . 66
4.3.3 Composition d'étiquettes . . . 67
4.4 Relations entre STEs et raffinement B . . . 70
4.4.1 Relation entre deux STEs sur un même alphabet . . . 70
4.4.2 Relation entre deux STEs avec des étiquettes différentes . . . 76
4.4.3 Relation entre un STE et plusieurs STEs . . . 83
4.4.4 Raffinement entre deux STEs gardés . . . 84
4.5 Travaux connexes . . . 85
4.5.1 Les travaux de [BJK00] . . . 85
4.5.2 L'approche *GeneSyst* . . . 85
4.5.3 *ProB* . . . 86
4.6 Conclusion . . . 86

5 Assemblage de deux interfaces

5.1 Introduction . . . 89
5.2 Description des interfaces . . . 91
5.2.1 Diagrammes UML . . . 91
5.2.2 Formalisation . . . 93
5.2.3 Traduction en B . . . 93
5.3 Compatibilité entre deux interfaces . . . 94
5.3.1 Construction B . . . 94
5.3.2 Raffinement et compatibilité entre interfaces . . . 96
5.4 Adaptation . . . 100
5.4.1 Forme générale d'un adaptateur en B . . . 101
5.4.2 Correspondances entre opérations . . . 102
5.4.3 Compatibilité entre l'adaptateur et les deux interfaces . . . 111
5.5 Détection des incompatibilités, diagnostic et correction . . . 114
5.5.1 Typologie des erreurs . . . 114
5.5.2 Correction de l'adaptateur . . . 115
5.5.3 Stratégies de correction . . . 118
5.5.4 Étude de cas . . . 118
5.6 Travaux connexes . . . 124
5.7 Conclusion . . . 126

6 Assemblage de plusieurs composants

6.1 Introduction . . . 127
6.2 Présentation de l'approche . . . 128

6.3 Les trois premières étapes . . . 130
6.3.1 Spécification globale du système . . . 130
6.3.2 Description de l'architecture du système en termes de composants . . . 130
6.3.3 Spécification des fonctionnalités en termes des composants . . . 131
6.4 Formalisation . . . 132
6.4.1 Les STEs . . . 133
6.4.2 Diagrammes de séquences . . . 133
6.4.3 Comportement du système assemblé . . . 134
6.5 Vérification . . . 135
6.5.1 Traduction en B . . . 135
6.5.2 Vérifications en B . . . 137
6.6 Détection d'erreur, diagnostic et correction . . . 138
6.6.1 Typologie des erreurs . . . 138
6.6.2 Correction du médiateur . . . 138
6.6.3 Stratégies de correction . . . 141
6.6.4 Étude de cas . . . 141
6.7 Conclusion . . . 149

7 Conclusion

7.1 Bilan . . . 151
7.2 Perspectives . . . 152
7.2.1 Schémas B . . . 152
7.2.2 Détection des erreurs, analyse et correction des spécifications B . . . 152
7.2.3 Application à l'assemblage des composants . . . 153

Bibliographie **157**

Acronymes **163**

Annexes

A Prouveur de l'Atelier B

A.1 Quelques formes normales . . . 165
A.2 POs générées pour l'exemple 4.4.1 . . . 165

B Spécifications B des exemples du chapitre 5

B.1 Les spécifications B des lampes . . . 171
B.2 Les spécifications B de lecteur de CD . . . 179

C Études de cas d'assemblage de plusieurs composants

C.1 Un système de contrôle d'accès à un bâtiment 181
C.2 Les feux tricolores d'un carrefour 186
C.2.1 Spécification abstraite du système 187
C.2.2 Description de l'architecture du système en termes de composants 187
C.2.3 Spécification des services en termes des composants de l'architecture . . . 189
C.2.4 Spécification abstraite du composant *Feu_tricolore* 190
C.2.5 Description de l'architecture du composant Feu_tricolore 190
C.2.6 Spécification des services du composant Feu_tricolore 191

Table des figures

1.1 Vue d'ensemble de notre approche 5

2.1 Principe du développement par raffinement en B 11
2.2 Forme générale d'une machine abstraite B 14
2.3 Raffinement B associé à la machine M (Fig. 2.2) 16
2.4 Prouveur de l'atelier B 24

3.1 Classification des niveaux d'interopérabilté 35
3.2 Treillis de correspondances de spécification 38

4.1 Exemple de deux STEs 63
4.2 Machines B associées aux STEs T_1 et T_2 (Figure 4.1) 67
4.3 Relation entre les variables de M1 et M2 67
4.4 Construction B 70
4.5 Raffinement d'une étiquette *ee* 74
4.6 Assertion dans le raffinement M2 76
4.7 Construction B 77
4.8 Raffinement de chemins 82
4.9 Construction B 84

5.1 Vue d'ensemble de notre approche d'assemblage de deux interfaces 91
5.2 Interface $RI\_GRlight$ 92
5.3 La machine RI_GRlight 94
5.4 Compatibilité entre C_1 et C_2 95
5.5 Interface $PI\_GRlight$ 96
5.6 Extrait du raffinement B associé à l'interface $newPI\_GRlight$ 96
5.7 Adaptateur 100
5.8 Forme générale d'une spécification adaptateur 101
5.9 Interface $PI\_SMlight$ 104
5.10 Opérations onGreen() et onRed() 104
5.11 Interface PI_Mlight 106
5.12 Opération onGreen() 106
5.13 Opération onColor() 107
5.14 Interface RI_Blight 108
5.15 Opération BlinkRed() 109
5.16 Interface PI_Slight 110
5.17 Interface PI_Tlight 110
5.18 Interface RI_player 119

5.19 Interface PI_player . . . 121
5.20 Première version de l'adaptateur entre les interfaces PI_player et RI_player . . 122
5.21 Adaptateur entre les interfaces PI_player et RI_player . . . 125

6.1 Vue d'ensemble de notre approche . . . 129
6.2 Spécification globale du système de contrôle d'accès . . . 130
6.3 Architecture globale du système en termes des composants utilisés . . . 131
6.4 Composants de l'architecture du système de contrôle d'accès . . . 132
6.5 Un scénario pour la fonctionnalité *Leave* . . . 132
6.6 Architecture B du système en termes de composants . . . 137
6.7 Définition de la relation REL . . . 137
6.8 Renforcer une garde . . . 140
6.9 Fausse définition de la relation REL . . . 142
6.10 Diagramme erroné de la fonctionnalité Enter . . . 142
6.11 Diagramme corrigé de la fonctionnalité Enter . . . 148
6.12 Spécification corrigée de la fonctionnalité Enter . . . 149

7.1 Construction B . . . 152

A.1 Rffinement d'une étiquette *aa* . . . 166

B.1 Raffinement B newPI_GRlight . . . 171
B.2 Machine B associée à l'interface PI_Mlight . . . 172
B.3 Machine B associée à l'interface $PI\_SMlight$ . . . 173
B.4 Machine B associée à l'interface RI_Blight . . . 173
B.5 Machine B associée à l'interface PI_Slight . . . 174
B.6 Machine B associée à l'interface PI_Tlight . . . 175
B.7 Adaptateur entre les interfaces $RI\_GRlight$ et PI_Mlight . . . 176
B.8 Adaptateur entre les interfaces $RI\_GRlight$ et $PI\_SMlight$ . . . 176
B.9 Adaptateur entre les interfaces $PI\_SMlight$ et $PI\_GRlight$ . . . 177
B.10 Adaptateur entre $RI\_GRlight$, $Red.PI\_Slight$ et $Green.PI\_Slight$ . . . 177
B.11 Adaptateur entre les interfaces RI_Blight et $PI\_GRlight$ . . . 178
B.12 Machine B associée à l'interface RI_player . . . 179
B.13 Machine B associée à l'interface PI_player . . . 180

C.1 Le composant *Turnstile* . . . 181
C.2 Le composant Card_reader . . . 182
C.3 Le composant Lamp . . . 183
C.4 Le composant *DB* . . . 184
C.5 Raffinement AccessControl erroné . . . 185
C.6 Raffinement AccessControl corrigé . . . 186
C.7 Le système abstrait du Carrefour . . . 187
C.8 Architecture du système Carrefour . . . 188
C.9 Le composant Feu_tricolore . . . 188
C.10 Description du service MiseEnService . . . 189
C.11 Description du service Change . . . 189
C.12 Invariant . . . 190
C.13 Architecture du système . . . 190
C.14 Architecture du composant Feu_tricolore . . . 191

C.15 Le composant Lampe 191
C.16 Description du service Succ 192
C.17 Invariant 192

Liste des tableaux

2.1 Substitutions primitives 12
2.2 Les $\mathcal{WP}$ des substitutions primitives 12
2.3 Quelques règles d'équivalence de substitutions 13
2.4 Substitutions dérivées 13
2.5 Prédicats définis sur les substitutions généralisées 14
2.6 Définition des prédicats 14
2.7 Les messages associés aux obligations de preuve élémentaires 25

3.1 comparaison des approches d'adaptation 52

4.1 Correspondance entre composition d'étiquettes et substitutions généralisées . . . 68

5.1 Différents cas de correspondances entre opérations 103

A.1 Certaines formes normales 165

1

Introduction

Sommaire

1.1 Contexte 1
1.1.1 Méthode B 1
1.1.2 Approche CBSE 3
1.2 Contributions 4
1.2.1 Raffinement des systèmes de transitions étiquetés en B 4
1.2.2 Aide à la construction de spécifications B à partir de l'échec de la preuve 4
1.2.3 Application à une approche CBSE 5
1.3 Plan de la thèse 6

Les méthodes formelles sont reconnues comme essentielles dans le cycle de vie du logiciel. Elles sont utilisées dans les différentes étapes du cycle de vie : les phases d'analyse, de conception et de validation et permettent d'améliorer la qualité des logiciels en éliminant les oublis, les incohérences et les bugs.

Les méthodes formelles déductives s'appuient sur des systèmes formels pour exprimer et vérifier les propriétés et les comportements attendus des systèmes. Elles sont constituées de langages formels munis de syntaxes bien définies et d'assistants de preuve ou de « model-checking » qui permettent de vérifier les propriétés.

Malgré tous les avantages qu'elles procurent, les méthodes formelles restent difficiles à utiliser et sont actuellement réservées aux logiciels les plus critiques. Nous sommes persuadés que l'élargissement des champs d'application des méthodes formelles à de nouveaux domaines doit se faire en liaison avec la proposition de nouvelles méthodologies de développement, c'est ce que nous tenons de montrer dans cette thèse, dans le domaine du développement par composant.

1.1 Contexte

Ce travail de thèse s'inscrit dans l'étude de la vérification et de la correction des spécifications B, dans le contexte de la construction des systèmes par assemblage des composants (CBSE, pour « Componant-Based Software Engineering »).

1.1.1 Méthode B

La méthode B a été introduite par J.R. Abrial au début des années 80 comme une méthode de développement de logiciels prouvés, elle est décrite dans le B-Book [Abr96a]. Ses caractéristiques

principales sont les suivantes : (i) des fondations mathématiques en logique du premier ordre avec *théorie des ensembles*, constituant un corpus formel connu et compris depuis longtemps, (ii) la *modularité* facilitant le développement de systèmes complexes, (iii) le *raffinement* permettant un développement incrémental depuis un niveau abstrait jusqu'à la génération de code et (iv) la possibilité de prouver chaque étape du développement.

La méthode B a été appliquée avec succès dans le développement d'applications réelles complexes, comme le projet METEOR [BBM99] ou le métro Val [BA05] et en recherche où l'on a proposé des extensions de la méthode et des combinaisons avec d'autres formalismes. Le succès de cette méthode provient notamment du fait qu'elle s'appuie sur des outils robustes qui permettent de spécifier et de valider des logiciels de grande taille. La plupart de ces outils intègrent des prouveurs de théorèmes : AtelierB [Cle08c], B4Free [Cle08d] avec Click'n'Prove [AC03] et BToolKit [B-C96]. Il existe aussi ProB [LB08] qui est basé sur le « model-checking ».

Nous nous intéressons principalement à trois thèmes : le raffinement B, l'aide à la correction de spécifications B à partir de l'échec de la preuve et l'utilisation conjointe de B et d'UML.

Raffinement B. Le raffinement est une notion clé de la méthode B, il est très utilisé pour le développement de logiciels. Le raffinement B établit une relation entre une spécification abstraite et une spécification plus concrète. Par une suite de raffinements, il est ainsi possible de transformer pas à pas une modélisation mathématique, qui correspond à une spécification initiale, en une représentation informatique (implantation). Les obligations de preuve correspondant à la vérification d'un raffinement sont automatiquement générées par les outils B. Le raffinement a fait l'objet de différentes études concernant sa formalisation, l'ajout de conditions supplémentaires pour renforcer cette notion, la construction d'outils de vérification, *etc.* Les propriétés exprimées par le raffinement et l'outillage associé ont également été la motivation d'autres travaux relatifs à la vérification [Tru06, CHS06, HHS06]. Nous nous intéressons plus particulièrement à ces derniers travaux.

Aide à la construction de spécifications B à partir de l'échec de la preuve. La notion d'obligation de preuve fait partie intégrante de la méthode B. Lors de l'écriture des spécifications (machines abstraites, raffinements, implantations), des formules, appelées obligation de preuve (PO), sont générées et doivent être ensuite prouvées pour valider les spécifications. Les outils implantant la méthode B proposent des prouveurs automatiques et interactifs dont les limites sont liées au caractère semi-décidable de la logique du premier ordre. Lorsqu'une obligation de preuve ne peut être prouvée, il se peut que le prouveur ou l'utilisateur ne trouvent pas la bonne stratégie de règles à appliquer ou que l'obligation de preuve soit fausse. C'est à ce deuxième cas que nous nous intéressons, puisque nous étudions la détection de l'échec, le diagnostic et la proposition d'aide à la correction des spécifications en utilisant les réponses du prouveur de l'Atelier B.

Utilisation conjointe de B et UML. La méthode B, comme toute autre méthode formelle est une démarche basée sur des notations mathématiques et des preuves de validation formelle. Un des principaux avantages des méthodes formelles est la précision, la non ambiguïté et la détection des erreurs. Par contre, les méthodes formelles sont plus lourdes à utiliser et moins intuitives que les méthodes semi-formelles à base de diagrammes telle que UML, celles-ci ne permettent pas de vérifier la cohérence des systèmes et peuvent même avoir une sémantique ambiguë. L'utilisation conjointe des langages de spécification formels et semi-formels est une solution pour bénéficier des avantages des deux approches. Nous nous intéressons à l'intégration de UML et B en proposant des traductions de diagrammes UML en B, les travaux couvrent deux objectifs : le développement de spécifications prouvées et la vérification des propriétés.

1.1.2 Approche CBSE

Depuis plusieurs années, l'ingénierie du logiciel tend à s'inspirer du développement du « hardware » en utilisant l'assemblage de « composants logiciels », cette approche est une avancée majeure pour la construction de systèmes complexes. Elle vise la construction de systèmes de grande taille par l'assemblage de composants logiciels préfabriqués considérés comme des « boîtes noires » qui communiquent avec leur environnement par le biais d'interfaces. La construction basée sur l'approche CBSE présente des avantages en terme de modularité, de réutilisation, de coût et de sûreté.

Depuis son apparition l'approche CBSE est de plus en plus étudiée [Szy99, HC01] dans le domaine de la recherche et de plus en plus adoptée dans l'industrie. Cela se traduit par le nombre d'approches « composants » dans l'industrie : COM [Mic95], JavaBeans [Sun97, Sun01] .NET, *etc*, de langages de description d'architecture (ADLs, pour « Architecture Description Language ») : C2, WRIGHT, ACME, *etc*, de modèles de référence comme CORBA [Obj98] et UML 2.0 et par l'avènement des composants sur étagère COTS (« Components Off The Shelf »).

Les composants sont souvent développés de manière indépendante et conçus avec des objectifs et des conditions d'utilisation qui ne coïncident pas nécessairement avec ceux du système à réaliser. Le succès de l'application d'une approche CBSE dépend de l'interopérabilité entre les composants à assembler. L'interopérabilité est définie comme la capacité de deux ou plusieurs composants logiciels à coopérer indépendamment des différences concernant leurs langages d'implantation, leurs interfaces et leurs plate-formes et environnements d'exécution [Kon95, Weg96]. Dans le domaine, quatre niveaux d'interopérabilité sont couramment identifiés : syntaxique, sémantique, protocole et qualité de service (QoS). Le niveau syntaxique est bien supporté par les modèles des composants existants, mais l'étude des autres niveaux reste un problème complexe et délicat pour les concepteurs. L'interopérabilité d'un système dépend de plusieurs facteurs comme par exemple la description des interfaces, les conditions de compatibilité et l'adaptation.

Description des interfaces et de l'assemblage. L'expressivité du langage, utilisé pour la description des interfaces et de l'assemblage des composants, influe sur la qualité de développement des logiciels dans une approche CBSE. Dans ce contexte, le formalisme UML 2.0 est très utilisé étant donné qu'il offre des éléments permettant de représenter divers aspects d'un système. D'une part, il comporte des diagrammes structurels comme par exemple les diagrammes de classes et les diagrammes de structures composites. D'autre part, il est doté de diagrammes comportementaux assez expressifs et intuitifs tels que les diagrammes d'états-transitions du protocole, dédiés à la description des interfaces des composants, et les diagrammes de séquences décrivant des interactions entre plusieurs composants.

Conditions de compatibilité. La vérification de l'intéropérabilité se base sur un ensemble de conditions (ou de propriétés) à tester entre les spécifications. Parmi ces conditions, nous pouvons citer la compatibilité (*i.e.* les conditions à vérifier entre les composants à assembler), la substitution (*i.e.* les conditions permettant à une interface d'être substituée par une autre interface), la conformité (*e.g.* les conditions entre une spécification et son implantation), *etc.* Ces conditions sont définies pour chaque niveau d'interopérabilité différemment.

Adaptation. Elle fournit principalement des solutions pour résoudre les incompatibilités pouvant apparaître lors d'un assemblage de composants. Elle peut être appliquée lors des différentes phases du cycle de vie du logiciel. Elle est souvent décrite entre deux ou plusieurs interfaces respectivement par des règles de correspondance ou de coordination entre les interfaces.

Dans une approche CBSE, l'utilisation de méthodes formelles (*e.g.* Z, B, VDM, ASM, algèbres de processus, *etc.*) s'avère nécessaire puisque les vérifications formelles permettent d'obtenir des

logiciels sûrs, fiables et corrects. Plus le système considéré est grand, plus la tâche de vérification est compliquée et par conséquent nous devons penser à des mécanismes pour simplifier et décomposer la vérification tel que le raffinement.

1.2 Contributions

Nos contributions visent les trois objectifs suivants : le raffinement des systèmes de transitions étiquetés (STEs) en B, l'aide à la correction de spécifications B à partir de l'échec de la preuve et l'application de la méthode B dans le contexte d'une approche de construction de logiciel par assemblage des composants. Plus précisément, la troisième contribution propose des solutions pour un assemblage de composants basé sur l'adaptation, elle correspond à un domaine d'application pour les deux premières contributions. Nous détaillons successivement chacune de ces contributions dans les sections suivantes.

1.2.1 Raffinement des systèmes de transitions étiquetés en B

Nous spécifions les protocoles des composants en terme de STEs, notion très utilisée pour décrire le comportement des composants, de plus la traduction d'un STE en B est simple, directe et se justifie par le fait que la sémantique opérationnelle de B peut se décrire naturellement par des STEs. Cette étude sert de base pour la spécification et la vérification au niveau protocole dans les cas d'assemblage de composants que nous considérons.

Nous nous intéressons au raffinement B en considérant des spécifications B modélisant des STEs, le raffinement B peut alors être vu comme un raffinement de STEs. Notre étude consiste à déterminer quelles sont les relations entre les STEs que nous pouvons modéliser grâce au raffinement B, cela se fait par l'analyse des obligations de preuve du raffinement. En vu de renforcer les relations exprimées par le raffinement, nous complétons les spécifications dans certains cas en ajoutant des assertions.

En B, cette étude se concrétise par des schémas de spécifications B utilisant principalement les clauses **REFINES** et **INCLUDES**. Nous considérons deux types de schémas :

- Les schémas basés uniquement sur la clause **REFINES**, nous permettent de définir des relations de raffinement entre deux STEs définis sur un même alphabet et par rapport à une relation entre les états des deux STEs.
- Les schémas basés sur les clauses **REFINES** et **INCLUDES**, nous permettent de définir des relations entre des STEs munis d'alphabets différents. Dans ce cas, les raffinements considérés incluent aussi le raffinement des étiquettes par l'intermédiaire d'une relation entre étiquettes (proche du raffinement atomique d'actions [GR01]). C'est ce qui a nécessité l'introduction de la notion de composition d'étiquettes, qui permet d'exprimer la relation entre les étiquettes et d'associer une sémantique à cette relation.

1.2.2 Aide à la construction de spécifications B à partir de l'échec de la preuve

Nous nous intéressons à la détection de l'échec, le diagnostic et la proposition d'aide à la correction. Notre objectif consiste donc à étudier les réponses du prouveur afin de proposer des aides à la correction des spécifications B.
Les corrections que nous proposons s'appuient à la fois sur une analyse statique des obligations de preuve fausses et sur la sémantique des spécifications considérées. Concrètement, nos études sont

menées en considérant les retours de l'échec de la preuve du prouveur de l'atelier B. Nous proposons initialement un ensemble de corrections selon le type de l'obligation de preuve considérée. Ces corrections sont, ensuite, complétées en tenant compte de la sémantique des spécifications, pour les deux cas d'assemblages étudiés dans notre travail (assemblage de deux interfaces requise et fournie et assemblage de plusieurs composants).

1.2.3 Application à une approche CBSE

Méthode générale. Le schéma de la Figure 1.1 décrit la méthode générale que nous adoptons pour l'assemblage des composants. Cette méthode est composée de trois étapes : la première étape consiste à élaborer une spécification initiale en utilisant des notations graphiques en UML et des STEs. Dans la deuxième étape, la spécification initiale est transformée en B par des règles de traduction systématique et complétée par la traduction de conditions supplémentaires, *e.g.* pour modéliser les conditions de compatibilité. La troisième étape est la vérification, si elle échoue, une étape de détection d'erreurs, d'analyse et de correction est effectuée de façon à obtenir une spécification correcte.

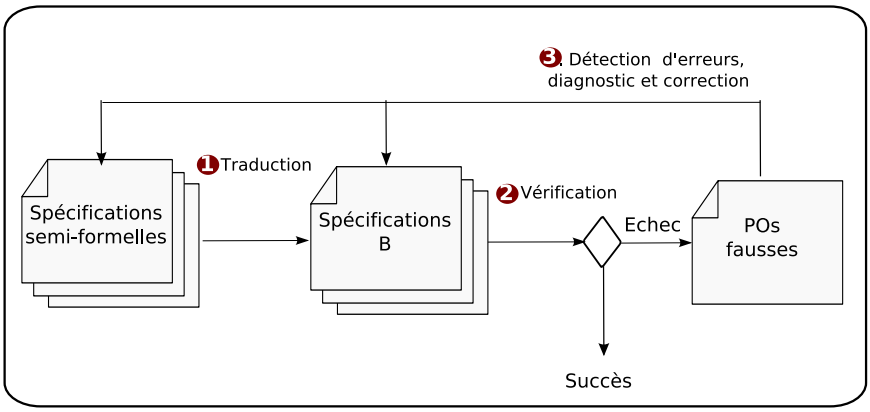


FIGURE 1.1 – Vue d'ensemble de notre approche

Spécifications UML/B. Afin de bénéficier des avantages de l'intégration de deux formalismes semi-formel et formel pour la spécification et la vérification des propriétés dans une approche CBSE, nous adoptons une approche de spécification basée sur l'utilisation conjointe de B et d'UML. Plus précisément, nous nous sommes basés sur les approches [Led02, TS06] pour la traduction d'UML en B. Les diagrammes étudiés sont les diagrammes de séquences et les diagrammes permettant de décrire les interfaces des composants. La traduction que nous considérons se compose de deux étapes. La première étape consiste à associer une sémantique formelle aux diagrammes considérés. Cette sémantique formelle est parfois réductrice puisque notre but est d'enlever les cas ambigus qu'on ne peut pas vérifier ainsi que de centrer notre formalisation sur les propriétés à tester. Dans la deuxième étape, la traduction des diagrammes se fait de manière systématique à partir de la sémantique formelle considérée.

Assemblage et adaptation de deux interfaces. L'adaptation des interfaces a été étudiée intensivement dans la littérature [YS97, SR02, dAH01, BCP06]. Néanmoins, peu de travaux ont été effectués sur des interfaces incluant la description des niveaux syntaxique, sémantique et protocole. Or, le processus d'adaptation peut conduire à plusieurs solutions. Ainsi, plus nous incluons d'informations dans notre étude, plus les résultats de l'adaptation seront précis et vont correspondre le mieux aux besoins réels. Nous proposons une approche pour l'assemblage de deux

interfaces requise et fournie en considérant ces trois niveaux et nous étudions principalement le problème d'adaptation. La description des interfaces est donnée par des diagrammes UML qui seront traduits en B. La vérification de la compatibilité des interfaces est effectuée grâce à la validation du raffinement B, dans le cas contraire la description d'une adaptation est fournie par le concepteur en terme de règles de correspondance entre les deux interfaces qui sont directement traduites en B. En cas d'échec après une adaptation, nous guidons la correction de la spécification de l'adaptateur en analysant les erreurs détectées. En outre, la validation des spécifications B (avec adaptation ou non), nous permet de vérifier : (i) les propriétés d'absence de réception non spécifiée et d'absence de blocage, au niveau protocole et (ii) les conditions d'appariement entre les pré/postconditions des opérations des deux interfaces, au niveau sémantique.

Assemblage et coordination entre plusieurs composants. Nous présentons une approche pour l'assemblage de plusieurs composants au niveau protocole, en proposant la construction d'un médiateur qui assure la coordination entre les composants du système en cours de construction. Nous optons pour une approche par raffinement afin de bien gérer la complexité de la preuve des spécifications. Il s'agit de décrire, de manière abstraite, le protocole du système en termes de fonctionnalités. Raffiner un système abstrait consiste à détailler les interactions entre les composants synchronisés par un médiateur permettant ainsi de réaliser ses fonctionnalités. Les protocoles du système abstrait et des composants sont décrits respectivement par des STEs et des STEs gardés. Les diagrammes de séquences sont utilisés pour la description des interactions.

Nous traduisons ensuite ces spécifications en B pour réaliser la vérification de l'assemblage. La vérification de l'assemblage des composants revient à la vérification des interactions entre les composants du système et du médiateur. En cas d'échec, des étapes de détection d'erreur, d'analyse et de correction vont suivre afin de terminer la construction et la vérification du médiateur.

1.3 Plan de la thèse

Après cet exposé de notre problématique et de nos contributions majeures, nous organisons le reste du mémoire en suivant le plan suivant :

Chapitre 2 « La méthode B »

Dans le premier chapitre, nous décrivons brièvement les fondements de la méthode B en rapport avec notre cadre de travail. Nous présentons également le prouveur de l'atelier B. Le chapitre s'achève par une étude sur la détection, le diagnostic et la correction d'erreurs dans les spécifications B lorsque la preuve échoue.

Chapitre 3 « Adaptation logicielle statique »

Ce chapitre a pour but d'étudier quelques travaux effectués pour l'adaptation logicielle statique (*i.e.* applicable durant la phase de conception). Nous commençons par faire le tour de quelques approches d'adaptation selon le problème considéré. Nous présentons ensuite les principales caractéristiques de l'adaptation logicielle statique en comparant les approches d'adaptation évoquées.

Chapitre 4 « Raffinement des systèmes de transitions étiquetés »

Dans ce chapitre, nous introduisons un ensemble de schémas de spécifications B et nous détaillons leurs propriétés. Nous commençons par présenter des notions préliminaires. Ensuite, nous étudions la traduction à la fois des STEs et des relations entre STEs en des spécifications B. Nous exposons ensuite un ensemble de schémas composés de spécifications B et basés sur les clauses de modularité B et nous introduisons des propositions définissant leurs propriétés [MA09].

Chapitre 5 « Assemblage de deux interfaces »

Ce chapitre propose une approche d'assemblage de deux interfaces requise et fournie. Nous commençons par donner la description des interfaces considérées et la relation de compatibilité considérée. Dans un deuxième temps, nous détaillons successivement les principales caractéristiques de notre approche d'adaptation, la relation de compatibilité engendrée par l'adaptation et la correction de la spécification de l'adaptateur. Une partie de ce travail a fait l'objet de l'article [MLS06].

Chapitre 6 « Assemblage de plusieurs composants »

Le but de ce chapitre est de proposer une approche pour l'assemblage de plusieurs composants par l'intermédiaire d'un médiateur. Les premières étapes de cette approche basées sur des spécifications données par le concepteur sont initialement présentées. Nous détaillons ensuite les étapes de vérification, de détection d'erreur, de diagnostic et de correction. Les travaux de ce chapitre ont fait l'objet de deux publications [MSA08a] et [MSA08b].

Nous concluons en résumant le travail effectué et en indiquant quelques perspectives.

2

La méthode B

Sommaire

2.1 Introduction . . . 9
2.2 Fondement de la méthode B . . . 10
2.2.1 Principe du développement en B . . . 10
2.2.2 Les substitutions généralisées . . . 11
2.2.3 Machine abstraite . . . 14
2.2.4 Raffinement . . . 16
2.3 Modularité en B . . . 17
2.3.1 Les clauses **INCLUDES** et **IMPORTS** . . . 17
2.3.2 Les clauses **PROMOTES** et **EXTENDS** . . . 18
2.3.3 Les clauses **SEES** et **USES** . . . 18
2.4 Obligations de preuve . . . 18
2.4.1 Obligations de preuve associées à l'instanciation d'une machine . . . 20
2.4.2 Obligations de preuve associées aux assertions . . . 20
2.4.3 Obligations de preuve associées aux initialisations . . . 21
2.4.4 Obligations de preuve associées aux opérations . . . 21
2.5 Brève présentation du prouveur de l'atelier B . . . 22
2.5.1 Obligations de preuves élémentaires . . . 22
2.5.2 Le prouveur automatique . . . 23
2.5.3 Le prouveur interactif . . . 23
2.6 Détection, analyse et correction des erreurs dans une spécification B 24
2.6.1 Détection des erreurs : échec de la preuve . . . 24
2.6.2 Analyse des erreurs . . . 25
2.6.3 Diagnostic et correction des spécifications B . . . 26
2.7 Conclusion . . . 32

2.1 Introduction

Le but de ce chapitre est d'une part donner une brève présentation de la méthode B. Nous mettons l'accent sur ses principales caractéristiques dans les sections 2.2, 2.3 et 2.4. Ces notions sont nécessaires pour la présentation de notre apport aussi bien théorique que pratique. D'autre part, nous rappelons les principaux aspects du prouveur de l'atelier B dans le section 2.5. Dans la section 2.6, nous proposons ensuite une étude pour la correction des spécifications en nous basant sur l'analyse de l'échec de la preuve.

2.2 Fondement de la méthode B

La méthode B [Abr96a, Sch02] est une méthode formelle de développement de programmes de l'étape de spécification jusqu'à l'étape d'implantation.

Les principales caractéristiques de la méthode B sont :

- Les fondements théoriques de la méthode B sont la théorie des ensembles, la logique des prédicats et le langage de substitution généralisée. Grâce aux ensembles, on est capable de modéliser les données et l'on dispose également de notions telles que les relations, les applications ou les fonctions. Les formules de la logique des prédicats permettent d'exprimer les propriétés statiques des systèmes. Les substitutions généralisées fournissent un langage pour exprimer des modifications sur les données de spécification.
- La notion de machine abstraite B (AMN, « Abstract Machine Notation ») permet de modéliser une spécification ou une partie d'une spécification. Elle contient des variables représentant l'état de la machine et des opérations permettant d'observer ou de modifier ces variables. Elle comprend également un invariant décrivant les conditions vérifiées par les variables de la machine ainsi que d'autres clauses pouvant faire intervenir d'autres machines B.
- Le raffinement B établit une relation entre une spécification abstraite et une spécification plus concrète. Par une suite de raffinements, il est ainsi possible de transformer pas à pas une modélisation mathématique, qui correspond à une spécification initiale, en une représentation informatique (implantation).
- La modularité en B est assurée par les machines abstraites, les raffinements et les implantations qui sont en quelque sorte les modules B, cela permet de développer plus facilement des systèmes de grande taille, puisque les modules peuvent être développés et validés de manière indépendante. Les relations entre les différents modules se font par l'intermédiaire de clauses B comme **INCLUDES**, **SEES**, *etc.*
- La notion d'obligation de preuve est intégrée à la méthode B. Lors de l'écriture des spécifications (machines abstraites, raffinements, implantation), des formules, appelées obligation de preuve (PO), doivent être générées et ensuite prouvées pour valider les spécifications.

Dans la suite, nous allons détailler ces points. Pour des définitions plus détaillées et plus rigoureuses de B, nous recommandons le BBook [Abr96a] qui est le livre de référence de la méthode.

Une autre version de la méthode B est introduite, nommée B événementiel [Abr96b, CM06, Abr10] et dédiée à la spécification des systèmes distribués. Cette version offre principalement la notion d'évènement et la capacité de définir des propriétés dynamiques. La présentation du B événementiel n'est pas considérée.

2.2.1 Principe du développement en B

Le principe du développement par raffinements en B est décrit par la Figure 2.1, où l'on voit les différentes étapes menant de la spécification d'une machine abstraite à une implantation, par l'intermédiaires des raffinements successifs. Le raffinement permet de prendre en considération de nouveaux détails pris dans le cahier des charges afin d'aboutir à une spécification plus proche d'une implantation. À chaque étape, la vérification de la cohérence de la spécification par rapport à la spécification qui la précède est garantie par des obligations de preuve. De la même manière, la préservation des propriétés spécifiées par l'invariant est vérifiée pour chaque spécification. Notons que la méthode B peut être utilisée seulement dans certaines phases du cycle de vie d'un système, par exemple les dernières phases.

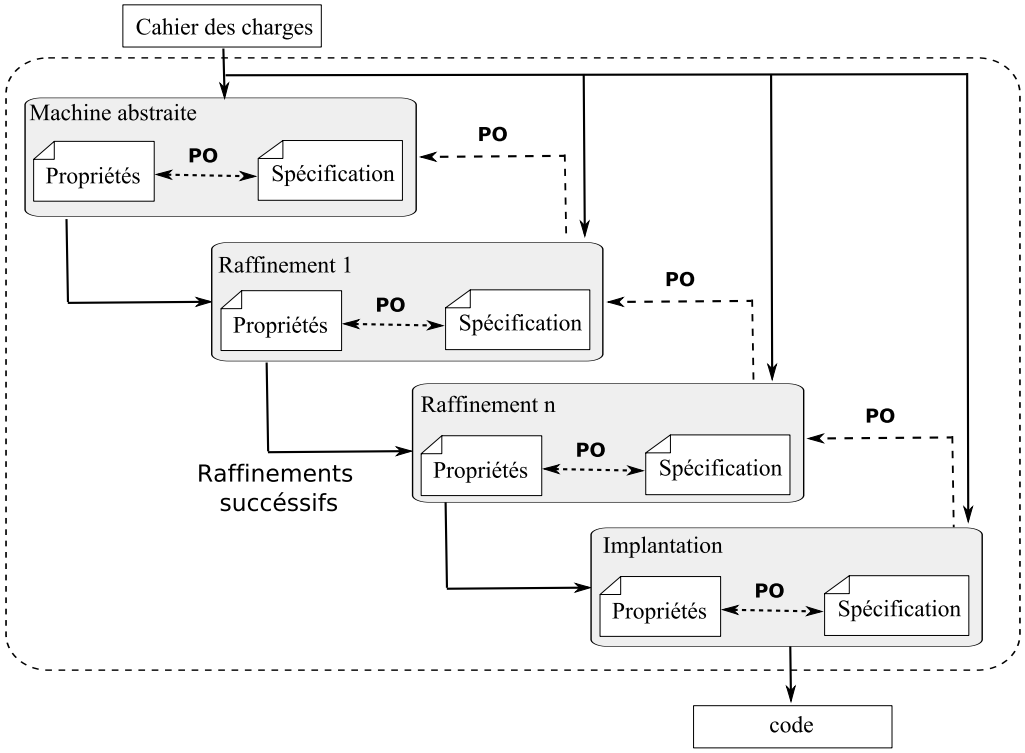


FIGURE 2.1 – Principe du développement par raffinement en B

2.2.2 Les substitutions généralisées

Les instructions dans les spécifications B sont exprimées par le langage des substitutions généralisées, qui est proche des langages de programmation abstraits. Le langage des substitutions généralisées est basé sur la théorie des transformateurs de prédicats introduite par Dijkstra [Dij76] qui est inspiré de la logique de Hoare [Hoa69]. En effet, la sémantique des substitutions généralisées est souvent définie en terme de calcul de la plus faible précondition, noté $\mathcal{WP}$ (« Weakest Precondition »). Si S est une substitution généralisée et R un prédicat, $[S]R$ assure que S termine et c'est la plus faible précondition qui après l'exécution de S garantit que le prédicat R est vérifié.

Les substitutions primitives

Les substitutions primitives sont des substitutions généralisées de base à partir desquelles les autres sont définies. Le tableau 2.1 présente les substitutions généralisées primitives avec deux notations : la première notation est appelée *mathématique* et la seconde notation est appelée *syntaxique* qui est proche de la syntaxe des instructions dans les langages de programmation.

Restriction 2.2.1. Les substitutions généralisées « séquencement » et « itération » ne sont pas autorisées dans les machines abstraites. Ces restrictions sont principalement d'ordre méthodologique.

Les axiomes du calcul de $\mathcal{WP}$ sur les substitutions primitives sont présentés dans le tableau 2.2.

Le tableau 2.3 présente quelques règles d'équivalence de substitutions.

Nom	Notation mathématique	Notation syntaxique
Substitution simple	$x := E$	x:=E
Substitution multiple simple	$x, y := E, F$	x,y:=E,F
Substitution sans effet	$skip$	**skip**
Substitution pré-conditionnée	$P \mid S$	**PRE** P **THEN** S **END**
Substitution choix borné	$S \,[]\, T$	**CHOICE** S **OR** T **END**
Substitution gardée	$P \Rightarrow S$	**SELECT** P **THEN** S **END**
Substitution choix non borné	$@z.S$	**VAR** z IN S **END**
Substitutions séquencée	$S;T$	S;T
Substitution d'itération	$\mathcal{W}(P,S,J,V)$	**WHILE** P **DO** S **INVARIANT** J **VARIANT** V **END**

TABLE 2.1 – Substitutions primitives

Substitution	$\mathcal{WP}$	Condition
1. $[x := E]R$	remplacement par E des occurrences libres de x dans R	
2. $[x, y := E, F]R$	$[z := F][x := E][y := z]R$	$z \backslash E, E, R$
3. $[skip]R$	R	
4. $P \mid S$	$P \wedge [S]R$	
5. $[P \,[]\, S]R$	$[P]R \wedge [S]R$	
6. $[P \Rightarrow S]R$	$P \Rightarrow [S]R$	
7. $[@z.S]R$	$\forall z.[S]R$	$z \backslash R$
8. $[S;T]R$	$[S][T]R$	
9. $\mathcal{W}(P,S,J,V)$	$J \wedge$ $\forall x.((J \wedge P) \Rightarrow [S]J) \wedge$ $\forall x.(J \Rightarrow V \in \mathbb{N}) \wedge$ $\forall x.((J \wedge P) \Rightarrow [n := V][S](V < n)) \wedge$ $\forall x.((J \wedge \neg P) \Rightarrow R)$	

TABLE 2.2 – Les $\mathcal{WP}$ des substitutions primitives

Les substitutions dérivées

Le tableau 2.4 présente un ensemble de substitutions dérivées. Ces substitutions sont obtenues à partir des substitutions primitives.

Les appels d'opération

En plus des substitutions généralisées, on peut utiliser des appels d'opérations dans les instructions d'une spécification B. Évidemment, on peut faire un appel d'une opération *op* d'une machine M si et seulement si on a **INCLUDES** M ou **IMPORTS** M ou **SEES** M (voir section 2.3). Un appel d'opération a la forme suivante :

$$r \longleftarrow op(v)$$

Règle	Gauche	Droite
$R1$	$P \Rightarrow (Q\|S)$	$(P \Rightarrow Q)\|(P \Rightarrow S)$
$R2$	$(P\|S);T$	$P\|(S;T)$
$R3$	$S;(P\|T)$	$[S]P\|(S;T)$
$R4$	$P\|(Q\|S)$	$(P \wedge Q)\|S$
$R5$	$(P\|S) \,[\!]\, T$	$P\|(S \,[\!]\, T)$
$R6$	$S \,[\!]\, T$	$T \,[\!]\, S$
$R7$	$T \,[\!]\, (P\|S)$	$P\|(T \,[\!]\, S)$
$R8$	$S;(T \,[\!]\, U)$	$(S;T) \,[\!]\, (S;U)$
$R9$	$(T \,[\!]\, U);S$	$(T;S) \,[\!]\, (U;S)$
$R10$	$P \Rightarrow (T \,[\!]\, S)$	$(P \Rightarrow T) \,[\!]\, (P \Rightarrow S)$
$R11$	$P \Rightarrow (T;S)$	$(P \Rightarrow T);S$
$R12$	$P \Rightarrow (Q \Rightarrow S)$	$(P \wedge Q) \Rightarrow S)$
$R13$	$(x := E);(P \Rightarrow T)$	$[x := E]P \Rightarrow (x := E;T)$

TABLE 2.3 – Quelques règles d'équivalence de substitutions

Notation	Définition
$x := E\|\|y := F$	$x, y := E, F$
IF P THEN S ELSE T END	$P \Rightarrow S \,[\!]\, \neg P \Rightarrow T$
x := bool(P)	IF P THEN x :=true ELSE x := false END
CHOICE S OR T ... OR U END	S[] T... [] U
ANY z WHERE P THEN S END	$@z.(P \Rightarrow S)$
$x :\in E$	ANY x' WHERE x' $\in$ E THEN x :=x' END
$x : P$	ANY x' WHERE [x$0,x :=x,x']P THEN x :=x' END

TABLE 2.4 – Substitutions dérivées

où r est une liste de variables et v une liste d'expressions, toutes les deux peuvant être vides.

L'application d'une substitution « appel d'opération » se fait par remplacement de cet appel par le corps de l'opération spécifiée dans sa machine abstraite d'origine. Par exemple, si $y \longleftarrow op(x) \mathrel{\widehat{=}} S$ est la définition d'une opération, le résultat de l'appel $r \longleftarrow op(v)$ est défini par : $[y, x := r, v]S$.

Prédicats définis sur les substitutions généralisées

Une substitution peut être caractérisée par sa terminaison, sa faisabilité et la relation entre les valeurs des variables avant et après son exécution. Dans le tableau 2.5, on présente un ensemble de prédicats qu'on peut appliquer sur les substitutions généralisées. Les prédicats $trm(S)$, $fis(S)$ et $prd_x(S)$ définissent respectivement la terminaison, la faisabilité et le prédicat avant-après pour la substitution S.

Le tableau 2.6 présente le calcul des prédicats $trm(S)$, $fis(S)$ et $prd_x(S)$ sur quelques substitutions.

Symbole	Définition	Définition mathématique
$trm(S)$	La terminaison de la substitution S	[S](btrue)
$prd_x(S)$	le prédicat avant-après	$\langle S \rangle (x' = x)$
$fis(S)$	La faisabilité de la substitution S	$\langle S \rangle btrue$

TABLE 2.5 – Prédicats définis sur les substitutions généralisées

Substitution généralisée : S	trm(S)	fis(S)	prd_x(S)
$x := E$	TRUE	TRUE	x'=E
$skip$	TRUE	TRUE	x'=x
$x :\in S$	TRUE	$S \neq \emptyset$	$x' \in S$
$x : P$	TRUE	$\exists x'.[x\$0, x := x, x']P$	$[x\$0, x := x, x']P$
$@x.S$	$\forall x.trm(S)$	$\exists x.fis(S)$	$\exists z.prd_x(S)\ if\ z \neq x'$
$choice\ S_1\ or\ S_2\ end$	$trm(S_1) \wedge trm(S_2)$	$fis(S_1) \vee fis(S_2)$	$prd_x(S_1) \vee prd_x(S_2)$
$select\ G\ then\ T\ end$	$G \Rightarrow trm(T)$	$G \wedge fis(T)$	$G \wedge prd_x(T)$
$pre\ G\ then\ T\ end$	$G \wedge trm(T)$	$G \Rightarrow fis(T)$	$G \Rightarrow prd_x(T)$
$any\ t\ where\ G\ then\ T\ end$	$\forall t.(G \Rightarrow trm(T))$	$\exists t.(G \wedge fis(T))$	$\exists t.(G \wedge prd_x(T))$

TABLE 2.6 – Définition des prédicats

2.2.3 Machine abstraite

Dans la méthode B, la notion de module est modélisée par la notion de machine [Pot03] (on parle parfois de modèle ou encore de système dans certaines versions de la méthode B).

```
MACHINE
 M(X,x)              /*M le nom de la machine, X (de type ensemble
                       et x (de type entier) ses paramètres*/
CONSTRAINTS
 C                   /*Les contraintes sur les paramètres X et x de la machine*/
SETS
 s                   /*Déclaration des ensembles*/
CONSTANTS
 c                   /*Déclaration des constantes*/
PROPERTIES
 Prop                /*Définition des propriétés sur les constantes c et les ensembles s*/
VARIABLES
 v                   /*Déclaration des variables*/
INVARIANT
 I                   /*Définition des invariants sur les variables v*/
ASSERTIONS
 J                   /*Assertions sur les variables v*/
INITIALISATION
 U                   /* Initialisation ( substitution )*/
OPERATIONS           /*Liste des opérations*/
 res⟵op(param)=      /*Forme générale d'une opération*/
  PRE    P          /*Précondition de l'opération op*/
  THEN   Q          /*Cops de l'opération op*/
  END;
 ...
END
```

FIGURE 2.2 – Forme générale d'une machine abstraite B

Une machine B est constituée schématiquement de variables (les données) et d'opérations qui ont pour rôle de modifier les variables de la machine. Dans une machine B, les données sont exprimées par des ensembles, les traitements par des substitutions généralisées et les propriétés par des prédicats du premier ordre.

Dans la Figure 2.2, on donne la forme générale d'une machine abstraite B. Elle peut être structurée comme suit : (i) entête, (ii) les machines vues, (iii) définition des données, (iv) caractérisation des données et (v) partie dynamique.

Entête

Elle comprend le nom de la machine suivi par des paramètres (optionnels).

Les machines vues

Les clauses **INCLUDES**, **SEES** et **USES** suivies d'un nom ou plusieurs noms d'instances de machines permettent respectivement d'inclure, de voir ou d'utiliser les machines désignées. Ces clauses expriment la modularité de B, elles sont détaillées dans la section 2.3.

Définition des données

On trouve dans cette partie, la déclaration des ensembles, des constantes et des variables. Les ensembles sont déclarés dans la clause **SETS**. Il existe deux types de variables : des variables abstraites et des variables concrètes qui sont définies respectivement dans les clauses **ABSTRACT_VARIABLES** (ou **VARIABLES**) et **CONCRETE_VARIABLES**. Les variables concrètes ne sont pas raffinées et par conséquent elles doivent respecter des conditions fortes de typage afin qu'elles soient implémentables telles quelles. Il est de même pour les constantes, on peut les déclarer dans les clauses **ABSTRACT_CONSTANTS** et **CONCRETE_CONSTANTS** (ou **CONSTANTS**). Dans la clause **VALUES**, on peut donner des valeurs aux constantes concrètes et aux ensembles abstraits.

Caractérisations des données

Dans cette partie, les données introduites sont caractérisées. Le typage et l'expression de propriétés sur les ensembles et les constantes sont définies dans la clause **PROPERTIES**.

La caractérisation des variables est effectuée dans les deux clauses **ASSERTIONS** et **INVARIANT**.

- Dans la clause **ASSERTIONS**, on peut définir des propriétés sur les variables d'une spécification B. Une assertion diffère de l'invariant, puisqu'elle doit être démontrée sous l'hypothèse que l'invariant et les assertions précédentes sont vraies. En outre, l'ajout d'une assertion A engendre l'ajout systématique de l'hypothèse A à toutes les obligations de preuve de la spécification B considérée.
- L'invariant défini dans la clause **INVARIANT** caractérise l'état d'une machine (*i.e.* le type des variables et leurs propriétés), contrairement aux clauses **PROPERTIES** et **ASSERTIONS** la clause **INVARIANT** n'est pas optionnelle. L'invariant d'une machine doit être vérifié par son état avant et après chaque opération indépendamment de la séquence d'opérations considérée. En B, on ne considère que des invariants inductifs. Un invariant est dit inductif s'il est vérifiable par induction, *i.e.* établi par l'initialisation et préservé par les opérations.

Remarque 2.2.1 (Terminologie). Dans le langage B, btrue et bfalse sont deux prédicats qui signifient respectivement vrai et faux.

Partie dynamique

La partie dynamique comprend l'initialisation, définie dans la clause **INITIALISATION**, et les opérations introduites dans la clause **OPERATIONS**. L'état initial des données est spécifié dans l'initialisation. Les opérations permettent de modifier l'état des données. Une opération, est dite une opération de requête, si elle fournit des informations sur l'état de la machine (*e.g.* des données de sortie) et ne modifie pas son état.

2.2.4 Raffinement

Un raffinement est un incrément par rapport à une spécification. Les données et les traitements peuvent être raffinés. Dans la Figure 2.3, nous donnons la définition du raffinement M_r associé à la machine M (Figure 2.2). Le raffinement est structuré de la même manière qu'une machine abstraite, la première différence est qu'il possède en plus la clause **REFINES** suivie du nom de la machine à raffiner. La sémantique d'un raffinement est différente de celle d'une ma-

```
REFINEMENT
 M_r(X,x)
REFINES          /*Nom de la machine à raffiner*/
 M
CONSTRAINTS
 C_r
SETS
 s_r
CONSTANTS
 c_r
PROPERTIES
 Prop_r
VARIABLES
 v_r
INVARIANT
 I_r
ASSERTIONS
 J_r
INITIALISATION
 U_r
OPERATIONS       /*Liste des opérations identiques que celles de la machine M*/
 res⟵op(param)=
  PRE    P_r
  THEN   Q_r
  END;
END
```

FIGURE 2.3 – Raffinement B associé à la machine M (Fig. 2.2)

chine abstraite puisqu'il fait intervenir la machine qui est raffinée et le raffinement lui-même. Le raffinement entre spécifications est transitif. Cette propriété permet de construire des systèmes pas à pas, ainsi que de diviser la complexité et la preuve de leur développement.

Le dernier niveau de raffinement d'une spécification, appelé implantation, se distingue par rapport aux autres raffinements. À ce niveau, toutes les variables doivent respecter des restrictions fortes de typage afin qu'elles soient directement programmables. Dans une implantation, les instructions indéterministes, préconditionnées et parallèles ne doivent plus apparaître, cependant l'itération ne peut être utilisée que dans une implantation.

Raffinement des variables

En général, les variables dans les premières spécifications dans un développement B sont des variables mathématique, par exemple des ensembles, de relations, *etc.* Le raffinement successif de ces variables revient à les remplacer successivement jusqu'à arriver à des structures informatiques, appelées variables concrètes.

Les *invariants de collage* relient les variables de niveaux différents. Ils peuvent être définis dans un raffinement (respectivement une implantation) pour lier entre les variables abstraites ou concrètes de ce raffinement (respectivement cette implantation) avec les variables abstraites d'une machine abstraite ou du raffinement précédent.

Les variables abstraites peuvent être héritées dans les raffinements. Cependant elles sont renommées systématiquement dans les obligations de preuve, en leur ajoutant le signe $1 comme extension à la fin de leur nom. Également, un invariant de collage sous forme d'une égalité est généré et il est dit implicite puisqu'il n'apparaît pas dans la clause **INVARIANT**.

Raffinement des instructions

Le raffinement des instructions consiste à redéfinir dans la machine raffinement chaque opération de la machine abstraite ou du raffinement précédent. Le raffinement des instructions (*i.e.* substitutions) permet principalement de réduire le non déterminisme et d'affaiblir les préconditions. Il permet également d'adapter les traitements aux modifications des données, les instructions écrites sont ainsi de plus en plus proches de l'implantation.

2.3 Modularité en B

Diverses possibilités de partage et de composition entre les machines B sont possibles et utiles au développement des projets B. Plus précisément, le développement modulaire des projets B est assuré par les clauses de modularité suivantes, permettant la structuration des projets :

- Les clauses **INCLUDES** et **IMPORTS**.
- Les clauses **PROMOTES** et **EXTENDS**.
- Les clauses **USES** et **SEES**.

Le point fort de la structuration modulaire dans un projet B est que la vérification des spécifications est aussi modulaire. En effet, la vérification de chaque machine du projet est indépendante et elle n'est pas remise en cause dans une composition.

Le graphe des dépendances d'un projet est obtenu en reliant chaque machine du projet aux machines qu'elle voit ou importe.

Restriction 2.3.1. Le graphe des dépendances ne doit pas contenir de cycle.

2.3.1 Les clauses INCLUDES et IMPORTS

La clause **INCLUDES** est utilisée dans les machines abstraites ou des raffinements. Elle est suivie d'une liste de machines $M_1, ..., M_k$. Les ensembles, constantes et variables des machines incluses M_i deviennent des ensembles, des constantes et des variables de la machine incluante. Les initialisations des machines M_i sont exécutées juste avant l'initialisation de la machine incluante. Les opérations des machines incluses peuvent être appelées à partir de l'initialisation ou des opérations de la machine incluante. Une machine incluse M_i peut être paramétrée par des ensembles et des scalaires, alors les paramètres seront instanciés dans la clause d'inclusion. Notons aussi que l'inclusion est transitive : si une machine M_1 est incluse dans une machine

M_2 qui est elle même incluse dans une machine M_3, alors M_1 est incluse dans M_3. Une machine peut être incluse plusieurs fois dans un projet à condition de faire un renommage, les instances considérées sont alors différentes.

Restriction 2.3.2. Afin de garantir la préservation de l'invariant des machines incluses, certaines restrictions sont imposées sur l'utilisation des variables et des opérations de ces dernières dans la machine incluante (initialisation et opérations). Ces restrictions sont :

- Les variables des machines incluses ne peuvent être utilisées qu'en lecture.
- Les opérations des machines incluses peuvent être appelées dans la machine incluante, à condition qu'il n'y ait pas d' appels en parallèle d'opérations qui modifient l'état d'une même machine incluse.

La clause **IMPORTS** est analogue à la clause **INCLUDES** et elle est seulement utilisée dans des implantations.

2.3.2 Les clauses PROMOTES et EXTENDS

La clause **PROMOTES** est suivie d'un ou plusieurs noms d'opérations d'une machine incluse ou importée. Ces opérations promues font partie de l'interface visible de la spécification qui inclut ou qui importe et doivent respecter son invariant.

La clause **EXTENDS** suivie d'un ou plusieurs noms d'instances de machines est utilisée lorsqu'on veut promouvoir toutes les opérations d'une machine incluse ou d'une machine importée.

2.3.3 Les clauses SEES et USES

Les clauses **SEES** et **USES** sont des clauses de partage. Elles sont suivies d'une liste de machines. Ces deux clauses ne sont pas transitives.

La clause **SEES** permet à une spécification B d'accéder en lecture aux informations d'une autre machine, plus précisément :

- Les paramètres et les opérations des machines vues ne sont pas accessibles.
- Les constantes et les ensembles des machines vues sont accessibles par toute les clauses.
- Les variables des machines vues ne peuvent être accessibles qu'en lecture et uniquement dans les opérations. La lecture des valeurs des variables peut être directe, comme elle peut être à travers des opérations de requête (*i.e.* ne modifient pas l'état de la machine vue).

Si une spécification B voit directement une machine, alors le raffinement de cette spécification doit voir aussi cette machine.

La clause **USES** permet à une spécification B d'utiliser les composants d'une autre machine à l'exception de ses opérations, en particulier :

- Les paramètres des machines utilisées sont accessibles dans l'invariant et les opérations.
- Les variables des machines utilisées sont accessibles dans l'invariant et les opérations, en lecture.
- Les constantes et les ensembles des machines utilisées sont accessibles dans les propriétés, l'invariant et les opérations.
- Les opérations ne sont pas accessibles.

2.4 Obligations de preuve

On distingue deux types de vérification en B : la vérification syntaxique (« type check ») et la vérification des propriétés des spécifications. Dans cette section, nous nous intéressons à

la seconde. Il est à noter que la vérification syntaxique est un présupposé à la vérification des propriétés. Cette seconde vérification est assurée par des obligations de preuve (POs). En effet, des POs sont générées, pour chaque spécification B et la réussite de leur preuve garantit sa correction.

Nous présentons dans la suite les POs associées aux spécifications B qu'on appelle des POs théoriques. Ces POs dépendent du type de la spécification considérée : une machine abstraite, un raffinement ou bien une implantation. Pour chacun d'eux on peut classer les obligations de preuves selon les quatre critères suivants :

- Vérification des assertions
- Vérification de l'inclusion d'une machine
- Vérification des opérations
- Vérification des initialisations

Pour les implantations, il y a d'autres POs qui concernent les valuations des constantes, les ensembles et la vérification des opérations locales, nous ne présentons pas ces obligations de preuves.

Dans cette section nous considérons une spécification M avec les paramètres X et x, I son invariant, J son assertion, S et $T = \{a, b\}$ ses ensembles, $Prop$ des propriétés sur ses constantes et ses ensembles, C les contraintes des paramètres de M (si M est une machine abstraite) ou de la machine référencée par M (si M est un raffinement ou une implantation), E une conjonction de prédicats qui dépend du contexte, U son initialisation ainsi que $op \mathrel{\widehat{=}} P|Q$ une de ses opérations. Les formules suivantes A et B décrivent respectivement les conditions sur les paramètres de M et les propriétés sur les constantes et les ensembles déclarés dans M :

$$A \stackrel{def}{=} C \ \wedge \ X \in \mathbb{P}_1(INT)$$
$$B \stackrel{def}{=} Prop \ \wedge \ S \in \mathbb{P}_1(INT) \ \wedge \ T \in \mathbb{P}_1(INT) \ \wedge \ T = \{a, b\} \ \wedge \ a \neq b$$

Une spécification M peut être une machine abstraite, un raffinement ou une implantation. Elle peut inclure des machines par la clause **INCLUDES** (M est une machine abstraite ou un raffinement) ou par la clause **IMPORTS** (M est une implantation), voir des machines par la clause **SEES** et raffiner une autre spécification par la clause **REFINES** (nous ne considérons pas la clause **USES**). Par conséquent, les POs associées à M peuvent être composées des éléments des raffinements antérieurs, des machines incluses et des machines vues. Nous considérons les notations suivantes indicées (s l'indice pour les machines vues, inc pour les machines incluses et imp pour les machines importées) :

- B_s : Propriétés sur les constantes et les ensembles des machines vues.
- $I_s \ \wedge \ J_s$: Invariants et assertions des machines vues.
- B_{inc} : Propriétés sur les constantes et les ensembles des machines incluses.
- $I_{inc} \ \wedge \ J_{inc}$: Invariants et assertions des machines incluses.
- R : Substitution qui permet de remplacer les paramètres formels des machines incluses ou des machine importées par des paramètres effectifs.
- B_{imp} : Propriétés sur les constantes et les ensembles des machines importées.
- $I_{imp} \ \wedge \ J_{imp}$: Invariants et assertions des machines importées.
- B_r : Propriétés sur les constantes et les ensembles des raffinements antérieurs et de la machine abstraite, y compris les propriétés de leurs machines incluses.
- $I_r \ \wedge \ J_r$: Invariants et assertions des raffinements et de la machine abstraite antérieurs.

Les machines vues (**SEES**) ne nécessitent pas d'obligations de preuve supplémentaires. Cependant, les hypothèses des POs générées pour la machine M sont renforcées en tenant compte des entités qui peuvent être vues. Les propriétés (B_s) sur les constantes et les paramètres des

machines vues sont ajoutées à toutes les hypothèses des POs. Les invariants et les assertions ($I_s \wedge J_s$) des machines vues ne sont présents que dans les hypothèses des POs générées pour l'initialisation et les opérations.

2.4.1 Obligations de preuve associées à l'instanciation d'une machine

On instancie les paramètres d'une machine incluse (respectivement importée) avant de l'utiliser. Ainsi, l'inclusion (respectivement l'importation) d'une machine est correcte si les contraintes des paramètres de la machine incluse (respectivement importée) sont vérifiées par les paramètres effectifs spécifiés.

Soient $N(X_N, x_N)$ une machine incluse ou importée dans M avec les paramètres effectifs T_N et t_N et C_N un ensemble de contraintes sur les paramètres de N. L'obligation de preuve suivante permet de vérifier que les instances T_N et t_N satisfont les contraintes C_N. Les paramètres effectifs T_N et t_N peuvent provenir des ensembles et des constantes de la machine M, des machines vues (**SEES**) et des paramètres de M, ce qui explique les hypothèses de l'obligation de preuve suivante.

PO 2.4.1. (Instanciation des paramètres)

$$\boxed{A \wedge E \wedge B \Rightarrow [X_N, x_N := T_N, t_N]C_N}$$

Dans le cas où M est une machine abstraite, E est définie par $E \stackrel{def}{=} B_s$. Dans le cas où M est un raffinement ou une implantation, E contient, en plus, les propriétés des constantes et des ensembles des raffinements antérieurs :

$$E \stackrel{def}{=} B_s \wedge B_r.$$

Remarque 2.4.1. Dans la suite les hypothèses des POs qui sont générées pour la vérification d'une spécification M, sont renforcées par les propriétés sur les constantes et les paramètres des machines incluses, ainsi que par les invariants et les assertions des machines incluses.

2.4.2 Obligations de preuve associées aux assertions

On donne la forme générale des POs associées à l'assertion J dans la spécification M. Une assertion peut être déduite des propriétés et de l'invariant de la spécification B où elle apparaît, ainsi que des propriétés et des invariants des machines incluses ou importées et des raffinements précédents. On considère que l'assertion J est une conjonction des prédicats de la forme : $j_1 \wedge ... \wedge j_{k-1}$. Ces assertions sont mises en séquence, ainsi l'assertion J_k peut être démontrée en utilisant les assertions $j_1, .., j_{k-1}$

PO 2.4.2. (Assertions)

$A \wedge E \wedge B \wedge I \Rightarrow j_1$	première assertion
$A \wedge E \wedge B \wedge I \wedge j_1 \wedge ... \wedge j_{k-1} \Rightarrow j_k$	assertion k

E est une conjonction de prédicats qui correspondent aux propriétés sur les constantes et les ensembles des machines référencées dans les clauses **SEES** et **INCLUDES** (respectivement **IMPORTS**). Elle comprend également les invariants et les assertions des machines incluses (respectivement importées) qui sont instanciés par les valeurs effectives des paramètres. Pour le raffinement et l'implantation, E est composée, en plus, des conjonctions des prédicats à partir des raffinements antérieurs. E est défini comme suit :

- Pour une machine abstraite, $E \stackrel{def}{=} B_s \wedge B_{inc} \wedge [R](I_{inc} \wedge J_{inc})$.
- Pour un raffinement, $E \stackrel{def}{=} B_s \wedge B_{inc} \wedge B_r \wedge [R](I_{inc} \wedge J_{inc}) \wedge I_r \wedge J_r$
- Pour une implantation, $E \stackrel{def}{=} B_s \wedge B_{imp} \wedge B_r \wedge [R](I_{imp} \wedge J_{imp}) \wedge I_r \wedge J_r$

Remarque 2.4.2. Les assertions dans une spécification permettent de renforcer les hypothèses de certaines de ses POs. En effet, à chaque fois que l'invariant apparaît dans les hypothèses d'une PO, les assertions doivent apparaître aussi dans ces hypothèses.

2.4.3 Obligations de preuve associées aux initialisations

Pour une spécification M, on présente les POs associées à l'initialisation. Si la machine M inclut (respectivement importe) des machines alors l'initialisation des machines incluses (respectivement importées) doit être faite avant celle de M. U_i représente le séquencement des substitutions des initialisations des machines incluses dans M (si M est une machine abstraite ou un raffinement) ou des machines importées dans M (si M est une implantation).

Machine abstraite

À l'initialisation de la machine, on doit s'assurer que l'état initial obtenu après l'exécution de l'initialisation, vérifie l'invariant de la machine. Par conséquence, l'obligation de preuve suivante est générée.

PO 2.4.3. (Initialisation)

$$\boxed{A \wedge E \wedge B \Rightarrow [U_i;U]I}$$

E est composé des conjonctions des prédicats définissant les invariants, les assertions et les propriétés des constantes et des ensembles déclarés dans les machines incluses et vues :
$E \stackrel{def}{=} B_s \wedge B_{inc} \wedge I_s \wedge J_s \wedge [R](I_{inc} \wedge J_{inc})$

Raffinement ou implantation

L'initialisation U doit établir l'invariant du raffinement (respectivement de l'implantation) sans contredire l'initialisation du niveau précédent, par exemple U'.

PO 2.4.4. (Initialisation d'un raffinement ou d'une implantation)

$$\boxed{A \wedge E \wedge B \Rightarrow [U_i;U]\neg[U']\neg I}$$

Si M est un raffinement, E est défini par des conjonctions de prédicats comme suit :

$$E \stackrel{def}{=} B_s \wedge B_{inc} \wedge B_r \wedge I_s \wedge J_s \wedge [R](I_{inc} \wedge J_{inc}).$$

Si M est une implantation, E est défini par des conjonctions de prédicats comme suit :

$$E \stackrel{def}{=} B_s \wedge B_{imp} \wedge B_r \wedge I_s \wedge J_s \wedge [R](I_{imp} \wedge J_{imp})$$

2.4.4 Obligations de preuve associées aux opérations

Dans ce qui suit, on présente les POs associées à l'opération *op* de la spécification M, ces POs doivent être vérifiées pour chaque opération de M.

Machine abstraite

La PO suivante permet de vérifier que si une opération est appelée sous sa précondition alors elle préserve l'invariant.

PO 2.4.5. (Opération)

$$\boxed{A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P \Rightarrow [Q]I}$$

E est composé des conjonctions de prédicats comme suit :

$$E \stackrel{def}{=} B_s \ \wedge \ B_{inc} \ \wedge \ [R](I_{inc} \ \wedge \ J_{inc}) \ \wedge \ I_s \ \wedge \ J_s$$

Raffinement ou implantation

Les opérations doivent préserver l'invariant du raffinement sans contredire l'opération spécifiée au niveau précédent.

PO 2.4.6. (Opération dans un raffinement ou dans une implantation)

Précondition	$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow P$
Corps	$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow [Q]\neg[Q']\neg I$

Si M est un raffinement, E est défini par une conjonctions de prédicats comme suit :

$$E \stackrel{def}{=} B_s \ \wedge \ B_{inc} \ \wedge \ B_r \ \wedge \ I_s \ \wedge \ J_s \ \wedge \ [R](I_{inc} \ \wedge \ J_{inc}) \ \wedge \ I_r \ \wedge \ J_r.$$

Si M est une implantation, E est défini par une conjonction de prédicats comme suit :

$$E \stackrel{def}{=} B_s \ \wedge \ B_{imp} \ \wedge \ B_r \ \wedge \ I_s \ \wedge \ J_s \ \wedge \ [R](I_{imp} \ \wedge \ J_{imp}) \ \wedge \ I_r \ \wedge \ J_r.$$

2.5 Brève présentation du prouveur de l'atelier B

Dans ce paragraphe, nous présentons le prouveur de l'atelier B [Cle08a, Cle08b]. Nous introduisons d'abord les formules manipulées par le prouveur, que nous appelons des obligations de preuves élémentaires. Les obligations de preuves élémentaires sont construites par le générateur d'obligations de preuves à partir des obligations de preuve théoriques décrites dans la section 2.4. Nous décrivons ensuite brièvement les deux composantes du prouveur qui sont le prouveur automatique et le prouveur interactif.

2.5.1 Obligations de preuves élémentaires

Les obligations de preuve élémentaires sont de la forme $HYP \vdash B$ où HYP est une conjonction de formules logiques qui sont les hypothèses et B est une formule logique appelée but. Montrer une telle obligation de preuve signifie montrer le but B en utilisant les hypothèses de HYP.

Les obligations de preuve élémentaires sont obtenues à partir des obligations de preuve théoriques du paragraphe 2.4, en composant les substitutions généralisées et en appliquant des transformations. Les obligations de preuves élémentaires obtenues sont plus uniformes, plus simples. Elles sont également plus nombreuses mais elles se prêtent mieux à une utilisation par le prouveur.

Les principales transformations utilisées pour obtenir les obligations de preuves élémentaires sont les suivantes :

- **Transformations des buts implicatifs**. On remplace $HYP \vdash P \Rightarrow Q$ par $HYP \wedge P \vdash Q$, la partie P du but $P \Rightarrow Q$ est devenue une hypothèse dans $HYP \wedge P$ et Q est le nouveau but, on dit que P est une hypothèse dérivée. Les hypothèses qui ne sont pas dérivées s'appellent des hypothèses contextuelles.
- **Transformation de buts conjonctifs**. On remplace la formule $HYP \Rightarrow B_1 \wedge \ldots \wedge B_n$, par les n formules $HYP \Rightarrow B_i$ où $i \in [1, n]$;
- **Normalisation**. La normalisation consiste à réécrire certains termes ou prédicats de façon à ce que plusieurs formes équivalentes aient une représentation unique. La normalisation permet de réduire ainsi le nombre de règles de réécriture utilisées dans la suite. Le tableau A.1 de l'annexe A présente certaines formes normales.

2.5.2 Le prouveur automatique

Le prouveur automatique de l'Atelier B est censé démontrer automatiquement les obligations de preuve, en appliquant un ensemble de mécanismes généraux. Le prouveur est paramétrable par des forces : force rapide, 0, 1, 2 ou 3, plus la force est élevée, plus le temps de la preuve s'accroît. Le prouveur automatique comporte :

- Une base de règles qui peuvent être appliquées par le prouveur. L'utilisateur peut également ajouter ses propres règles par l'intermédiaire des fichiers ppm (« Proof Manual Method ») ou des fichiers « Patchprover ».
- Des stratégies permettant de guider la preuve en décrivant le choix et l'ordre des règles de la base à appliquer.

Les règles

Les règles sont des formules de la forme $A \Rightarrow B$ où A et B sont des conjonctions de prédicats qui sont respectivement appelés l'antécédent et le conséquent de la règle.

Il existe trois types d'application pour les règles :

- Une règle $A \Rightarrow B$ peut être appliquée par un chaînage arrière. Si B est le but courant alors pour prouver B il suffit de prouver A qui devient le but courant à prouver. Cette application de règle se fait lorsque A est censé être plus simple à prouver que B.
- Une règle $A \Rightarrow B$ peut être appliquée par chaînage avant si les éléments de A apparaissent dans l'ensemble des hypothèses alors les éléments de B sont ajoutés aux hypothèses s'ils n'y étaient pas déjà.
- Une règle peut être appliquée comme une règle de réécriture si B est de la forme $C == D$. Si A est vérifiée alors C est réécrit en D. Les règles de réécriture s'appliquent sur le but courant ou sur des sous-formules contenues dans le but courant.

Les règles peuvent contenir des jokers. Un joker est une variable, qui peut prendre n'importe quelle valeur (littéral, expression,...). On instancie un joker en lui affectant une valeur.

2.5.3 Le prouveur interactif

Le prouveur interactif permet à l'utilisateur de reprendre la main lorsque c'est nécessaire, c'est-à-dire lorsque le prouveur automatique n'a pas réussi à prouver une formule. L'utilisateur peut alors orienter la preuve en ajoutant des hypothèses ou en appliquant par exemple des stratégies de preuve par contradiction ou par cas. Pour cela l'utilisateur dispose de commandes interactives qui sont appliquées à une obligation de preuve et qui peuvent être sauvegardées. La liste suivante décrit quelques fonctionnalités offertes par le prouveur interactif :

- Choix du niveau de preuve

- Appel des prouveurs
- Application d'une règle
- Réécriture
- Cas particuliers de règle d'inférence
- Opérations sur les hypothèses

À chaque moment d'une preuve interactive l'utilisateur peut faire appel au prouveur automatique pour prouver certaines formules, les preuves interactives sont en fait des preuves semi-automatiques alternant les actions de l'utilisateur et celles du prouveur. La Figure 2.4 présente l'interface du prouveur interactif de l'atelier B.

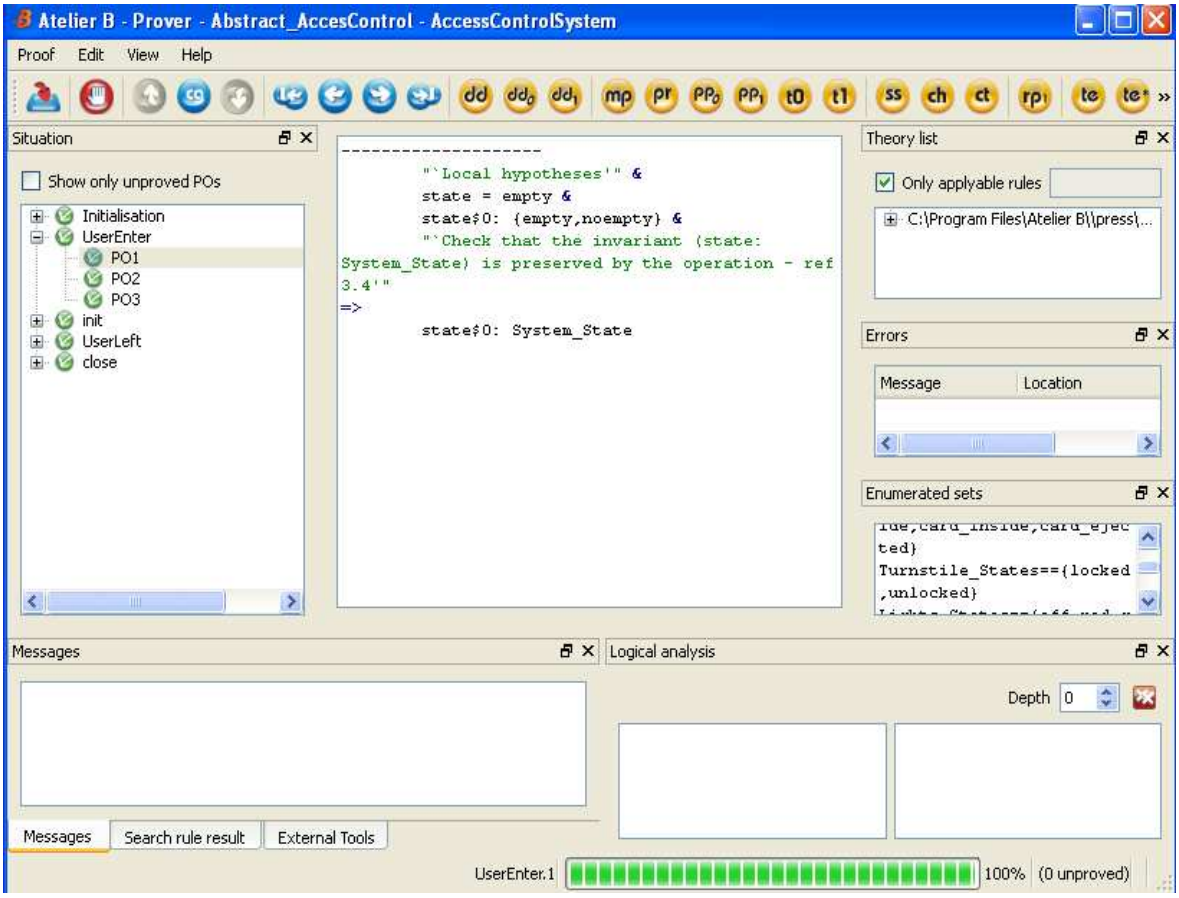


FIGURE 2.4 – Prouveur de l'atelier B

2.6 Détection, analyse et correction des erreurs dans une spécification B

2.6.1 Détection des erreurs : échec de la preuve

L'échec de la preuve se produit lorsqu'une ou plusieurs obligations de preuve ne peuvent être prouvées de façon automatique ou interactive. Cet échec peut être dû soit à des erreurs dans la spécification, soit à l'incapacité du prouveur ou de l'utilisateur à mener à bien la preuve. Dans la suite, nous nous plaçons dans le premier cas, lorsque la spécification B est fausse.

Définition 2.6.1. (Spécification fausse ou erronée) Une spécification B est dite fausse ou erronée si elle génère au moins une obligation de preuve fausse.

S'il est important de détecter les erreurs dans les spécifications B, il est encore plus intéressant de corriger ces erreurs. En utilisant les messages d'erreur de l'Atelier B et la façon dont les obligations de preuve sont générées, nous proposons une liste de corrections générales des spécifications B. Cette liste, que nous avons déduite à partir de nombreuses expérimentations menées avec l'Atelier B, n'est pas exhaustive et peut être enrichie.

2.6.2 Analyse des erreurs

Nous effectuons le diagnostic des obligations de preuves non prouvées de manière séquentielle pour chaque opération (y compris l'initialisation). Nous nous intéressons à la première erreur rencontrée dans le processus de vérification, celle-ci pouvant provoquer des erreurs en cascade.

Pour une erreur donnée, l'analyse d'erreur consiste à détecter les obligations de preuve élémentaires fausses et à déterminer quelle partie de la spécification est la cause de l'erreur. Pour cela, nous utilisons les retours du prouveur de l'Atelier B.

Dans le prouveur de l'atelier B, les obligations de preuve sont regroupées selon les clauses auxquelles elles sont associées, on distingue les cas suivants :

- Initialisation d'une machine abstraite, d'un raffinement ou d'une implantation.
- Nom d'une opération.
- Assertion dans une machine abstraite, un raffinement ou une implantation. Une assertion est appelée dans le prouveur de l'atelier B « AssertionLemmas ».
- Inclusion d'une machine abstraite.

En outre, les obligations de preuve sont en général commentées par les messages en anglais illustrés dans le tableau 2.7. où J_k est un prédicat parmi la conjonction des prédicats qui forment

Notation	Message
$M1$	« Check that the constraint C_k is preserved by the parameters instanciation »
$M2$	« Check assertion J_k deduction »
$M3$	« Check that the invariant I_k is established by the initialisation »
$M4$	« Check precondition pre_k deduction »
$M5$	« Check preconditions of called operation, or While loop construction, or Assert predicates »
$M6$	« Check that the invariant I_k is preserved by the operation »

TABLE 2.7 – Les messages associés aux obligations de preuve élémentaires

l'assertion J, I_k est un prédicat parmi la conjonction des prédicats qui forment l'invariant I et pre_k est une condition parmi la conjonction des conditions spécifiées dans la précondition du raffinement de l'opération considérée.

Les messages suivants permettent de donner des informations sur la cause de l'échec de la preuve, en précisant ce que l'obligation de preuve courante sert à vérifier.

- $M1$ signifie que l'obligation de preuve courante sert à vérifier que la contrainte C_k est préservée par l'instanciation des paramètres ;
- $M2$ signifie que la PO courante sert à vérifier que J_k peut être déduit à partir d'un ensemble d'hypothèses telles que l'invariant ;
- $M3$ signifie que la PO courante sert à vérifier que I_k est établi par l'initialisation ;

- $M4$ signifie que la PO courante sert à vérifier que la précondition pre_k peut être déduite à partir d'un ensemble d'hypothèses telles que la précondition abstraite et l'invariant ;
- $M5$ signifie que la PO courante sert à vérifier soit la précondition d'une opération appelée, soit la preuve d'une boucle **WHILE**, soit la preuve d'un prédicat ASSERT ;
- $M6$ signifie que la PO courante sert à vérifier que I_k est préservé par l'opération considérée ;

Ces différentes indications nous permettent d'identifier la PO théorique (Section 2.4) qui est à l'origine de la PO élémentaire considérée et nous donne d'autres informations sur l'emplacement de l'erreur dans la spécification.

Un autre type d'analyse peut se faire sur la PO en analysant ses hypothèses et sa conclusion. Pour chaque erreur, ces informations concernent :

- les variables sources de cette erreur,
- l'état du système avant l'erreur, i.e. la valeur des différentes variables et des hypothèses prises par le système sur ces variables.

Ces données peuvent servir à une analyse se basant principalement sur leur sémantique. Dans ce chapitre, cette étape n'est pas considérée puisqu'on considère le cas général, par contre dans les chapitres 5 et 6 cette étape est primordiale.

2.6.3 Diagnostic et correction des spécifications B

Dans ce paragraphe, nous proposons des corrections de spécifications, nous sommes guidés par la correction des obligations de preuve élémentaires fausses et par les messages de l'Atelier B qui nous permettent de pointer sur la partie de la spécification qui pose problème.

Une spécification B peut dépendre d'autres spécifications telles que les raffinements antérieurs, les machines incluses ou les machines vues. Parmi les alternatives des corrections proposées, nous pouvons mettre en cause les spécifications référencées et proposer de les corriger.

Dans ce qui suit, nous nous référons toujours à l'obligation de preuve théorique qui a généré l'obligation de preuve élémentaire à corriger. Puisqu'une obligation de preuve théorique découle des éléments de la spécification considérée, alors nous pouvons retrouver les éléments de la spécification à corriger. Bien que les obligations de preuve élémentaires soient obtenues à partir des obligations de preuves théoriques, retrouver les éléments de la spécification à corriger seulement à partir des obligations de preuve élémentaires n'est pas évident. En effet, le générateur des POs applique un ensemble de transformations pour obtenir des POs élémentaires de telle façon que souvent on ne retrouve plus les éléments de la spécification mis en jeu.
En effet, analyser la PO théorique qui a généré un PO élémentaire nous permet de nous focaliser seulement sur les éléments de la spécification qui apparaissent dans la PO théorique. En plus, cela nous permet de connaître l'origine des hypothèses et le but de la PO élémentaire et ainsi de pouvoir remonter à la spécification et la corriger. Prenons l'exemple de la PO élémentaire fausse suivante : $HYP \vdash bfalse$ (on rappelle que $bfalse$ est le prédicat signifiant faux). Supposons que l'on n'a pas de contradiction dans les hypothèses et que l'on désire seulement corriger son but $bfalse$. Corriger la spécification considérée en analysant seulement le but de cette formule n'est pas évident, car on n'a pas suffisamment d'information, alors qu' une analyse de cette PO et de sa PO théorique peut nous amener à déduire, par exemple que le but $bfalse$ est obtenu par l'application de la substitution $[x := 5]$ sur le prédicat $x > 10$. Donc, une correction possible est de corriger la substitution $[x := 5]$ ou le prédicat $x > 10$ qui sont deux parties de la spécification considérée.

Dans la suite, nous commençons par présenter un ensemble de corrections générales concernant les formules implicatives indépendamment du contexte. Ces corrections peuvent toucher les

hypothèses et le but de la formule. Nous parcourons toutes les POs théoriques présentées auparavant (Section 2.4) et pour chacune d'elles, nous présentons un ensemble de corrections associées aux formules élémentaires générées. Finalement, nous proposons deux stratégies de correction.

Remarque 2.6.1. La correction des obligations de preuve fausses est délicate dans le sens où la correction d'une formule fausse a une influence sur la spécification considérée. Par conséquent, les corrections peuvent introduire soit de nouvelles obligations de preuve à prouver soit la modification d'autres obligations de preuves. Notre but est de corriger la spécification, donc d'éliminer toutes les obligations de preuves fausses. Il est donc nécessaire d'avoir une vision assez générale pour corriger une spécification fausse.

Correction d'une formule implicative

On considère les obligations de preuve fausses de la forme :
- $HYP \vdash B$
- $HYP \vdash bfalse$

Pour le premier cas, on propose les corrections suivantes :
- Renforcer les hypothèses HYP
- Modifier les hypothèses HYP
- Affaiblir le but B
- Modifier le but B

Nous pouvons combiner ces corrections, par exemple renforcer les hypothèses et modifier le but.

Pour le second cas nous proposons les corrections suivantes :
- Introduire une contradiction dans les hypothèses HYP en les renforçant ou en les corrigeant
- Effectuer des modifications sur la façon dont le prédicat $bfalse$ est obtenu

Instanciation d'une machine incluse ou importée

L'obligation de preuve 2.4.1 associée à la vérification des instances des paramètres d'une machine incluse ou importée est :

$$A \ \wedge \ E \ \wedge \ B \Rightarrow [X_N, x_N := T_N, t_N]C_N$$

Cette obligation de preuve peut être décomposée en plusieurs POs élémentaires de la forme :

$$A \ \wedge \ E \ \wedge \ B \ \wedge \ H \Rightarrow B_k$$

Dans le prouveur interactif de l'Atelier B, ces POs élémentaires sont regroupées sous le label « InstanciatedConstraintsLemmas ». Elles sont aussi commentées par le message $M1$ qui précise la partie C_k des contraintes C qui a généré la PO considérée. On a :
- H est un ensemble d'hypothèses qui peut être vide et qui provient des formules implicatives dans la contrainte C_k.
- La formule B_k est générée par l'application de la substitution de l'instanciation $[X_N, x_N := T_N, t_N]$ sur C_k.

On note ce type de PO élémentaire fausse par $po1$.
Effectuer la correction de cette PO correspond à corriger ses hypothèses ou son but.

Les assertions

Rappelons les POs 2.4.2 associées à la clause **ASSERTIONS** (« AssertionLemmas ») :

$$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ j_1 \ \wedge \ ... \ \wedge \ j_{k-1} \Rightarrow j_k$$

Pour cette obligation de preuve, les transformations pour obtenir les obligations de preuve élémentaires sont directes, puisque j_k est un prédicat. Les obligations de preuve élémentaires sont commentées par le message $M2$ qui précise la partie j_k de l'assertion J qui a généré l'obligation de preuve considérée. On note ce type de PO élémentaire fausse $po2$.

Effectuer la correction de cette PO consiste à corriger ses hypothèses ou son but.

Partie dynamique d'une machine abstraite

Dans une machine abstraite, l'obligation de preuve concernant une opération fait principalement intervenir l'opération proprement dite et l'invariant de la machine, l'échec de la preuve peut provenir de ces deux éléments. On peut distinguer schématiquement les cas suivants :

- l'erreur peut être due à l'opération, cela peut se produire lorsque la précondition permet un appel non autorisé, la correction se fait par un renforcement de la précondition ou lorsque le corps de l'opération ne préserve pas l'invariant, la correction se fait en modifiant le corps de l'opération.
- L'opération peut décrire correctement le comportement requis mais l'incohérence est causée par l'invariant. Ceci peut être du à un invariant trop fort de telle façon qu'il exclue certains états satisfaisant simplement parce qu'ils sont oubliés. Dans ce cas, l'opération atteint des états à partir d'autres états légitimes et ces états sont exclus par l'invariant, alors on doit relâcher l'invariant. En outre, l'invariant peut être assez faible qu'il inclut des états qu'il ne doit pas permettre. Dans ce cas, l'opération échoue à préserver l'invariant quand elle est appelée à partir de ces états inaccessibles, alors l'invariant doit être renforcé pour exclure explicitement ces états.

INITIALISATION

Rappelons l'obligation de preuve 2.4.3 de l'initialisation :

$$A \ \wedge \ E \ \wedge \ B \Rightarrow [U_i; U]I$$

Cette obligation de preuve peut être décomposée en plusieurs obligations de preuves élémentaires de la forme :

$$A \ \wedge \ E \ \wedge \ B \ \wedge \ H \Rightarrow B_k$$

où H est un ensemble d'hypothèses dérivées.

On distingue deux types d'erreurs qui sont identifiées à partir du message associé à l'obligation de preuve considérée. Pour chacun d'eux, on propose un ensemble de corrections possibles.

1. Si l'obligation de preuve élémentaire est commentée par le message $M3$ où I_k est une partie de l'invariant spécifié dans ce message, alors, l'erreur provient de la violation de la partie I_k de l'invariant par l'initialisation. Dans ce cas :
   - H a comme origine les formules implicatives dans l'invariant I ou les substitutions gardées dans $U_i; U$.
   - La formule B_k provient de l'application de $U_i; U$ à I_k.

   Ce type de PO fausse est noté par $po3$.

   Par conséquent, corriger les hypothèses H revient à modifier l'invariant ou l'initialisation ou les deux. Effectuer des corrections sur le but B_k revient à corriger l'invariant ou l'initialisation ou les deux.
2. Si le message est $M5$, alors l'erreur provient du non respect de la précondition d'une opération par exemple $op_k \mathrel{\widehat{=}} K|L$ avec $K = K_1 \ \wedge ... \wedge \ K_n$.

   Dans ce cas :

- H a comme origine les substitutions gardées de $U_i; U$ qui précèdent l'appel de op_k.
- B_k est obtenue par l'application de U_i sur K_k où K_k est la partie de la précondition de K qui pose problème.

Ce type de PO fausse est noté par $po4$.
Par conséquent, corriger les hypothèses H revient à modifier $U_i; U$. Effectuer des corrections sur le but B_i revient à corriger U_i et K_k.

OPERATIONS

Rappelons l'obligation de preuve 2.4.5 associée à chaque opération :

$$A \wedge E \wedge B \wedge I \wedge J \wedge P \Rightarrow [Q]I$$

Cette obligation de preuve peut être décomposée en plusieurs obligations de preuve à prouver de la forme :

$$A \wedge E \wedge B \wedge I \wedge J \wedge P \wedge H \Rightarrow B_i$$

où H un ensemble d'hypothèses dérivées. Pour ces POs, on distingue deux types d'erreurs. On identifie les erreurs et on propose des corrections, selon le message associé à l'obligation de preuve considérée :

1. Si le message est $M6$, alors l'erreur provient de la violation de la partie I_k de l'invariant par l'opération considérée. I_k est une partie de l'invariant spécifié par le message $M6$. Dans ce cas :
   - H a comme originie les formules implicatives de l'invariant I ou les substitutions gardées de Q.
   - La formule B_i est générée à partir de l'application de Q sur I_k.

   Ce type de PO fausse est dénoté $po5$.
   Par conséquent, renforcer ou modifier les hypothèses H revient à modifier I ou Q ou les deux, de même effectuer des corrections sur le but B_i revient à corriger I ou Q ou les deux.
2. Si le message est $M5$, alors l'erreur est produite par la non respect de la précondition d'une opération appelée par exemple $op_k \mathrel{\widehat{=}} K|L$. Par conséquent :
   - H a comme origine les substitutions gardées de Q qui précèdent l'appel de op_k.
   - La formule B_i est de la forme K_k où K_k est une partie de K.

   Ce type de PO fausse est dénoté $po6$.
   Par conséquent, corriger les hypothèses H revient à modifier l'initialisation. Effectuer des corrections sur le but B_i revient à corriger K_k.

Partie dynamique d'un raffinement ou d'une implantation

Dans ce paragraphe, nous considérons les corrections relatives aux obligations de preuve concernant les raffinements et les implantations. Comme pour les machines abstraites, nous distinguons les obligations de preuve pour l'initialisation et pour les opérations. Les erreurs qu'on peut rencontrer dans un raffinement ou une implantation peuvent être dues à la non cohérence de l'initialisation et des opérations de la spécification considérée avec l'invariant. En outre, ces erreurs peuvent être provoquées par un lien qui est mal défini ou manquant entre les données de la spécification considérée et son raffinement (respectivement son implémentation), alors une correction possible est de corriger l'invariant ou de le renforcer en ajoutant les liens manquants.

INITIALISATION

Rappelons l'obligation de preuve 2.4.4 pour l'initialisation d'un raffinement ou d'une implantation :

$$A \ \wedge \ E \ \wedge \ B \Rightarrow [U_i;U]\neg[U']\neg I$$

Cette obligation de preuve peut être décomposée en plusieurs obligations de preuves à prouver qui sont de la forme :

$$A \ \wedge \ E \ \wedge \ B \ \wedge \ H \Rightarrow B_i$$

où H est un ensemble d'hypothèses dérivées. On distingue deux types d'erreurs qui sont identifiées à partir du message associé à la PO considérée et pour chacun d'eux un ensemble de corrections possibles.

1. Si la PO élémentaire considérée est commentée par le message $M3$ où I_k est une partie de l'invariant spécifié dans ce message, alors l'erreur provient de la violation de la partie I_k de l'invariant par l'initialisation. Dans ce cas :
   - H a comme origine les formules implicatives de l'invariant I ou les substitutions gardées de $U_i;U$.
   - La formule B_k est générée par l'application de la substitution $[U_i;U]\neg[U']$ sur I_k.

   Ce type de PO fausse est dénoté $po7$.
   Par conséquent, corriger les hypothèses H revient à modifier l'invariant I ou l'initialisation U ou les deux, il en est de même de la correction du but B_k.
2. Si le message est $M5$, alors l'erreur est générée par la non respect de la précondition d'une opération appelée par exemple $op_k \mathrel{\widehat{=}} K|L$ avec $K = K_1...K_n$. Dans ce cas :
   - H a comme origine les substitutions gardées de $U_i;U$ qui précèdent l'appel de op_k.
   - La formule B_k est générée par l'application de U_i sur K_k où K_k est la partie de la précondition de K qui pose problème.

   Ce type de PO fausse est dénoté $po8$.
   Par conséquent, corriger les hypothèses H revient à modifier $U_i;U$. Effectuer des corrections sur le but B_i revient à corriger U_i et K_k.

OPERATIONS

L'obligation de preuve 2.4.6 relative au raffinement d'une opération op est :

$$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow [Q]\neg[Q']\neg I$$

Pour une opération, nous distinguons cinq type d'erreurs et nous proposons des corrections.

1. Erreur dans la preuve de la précondition de l'opération raffinée. Cette erreur est identifiée par le message $M4$. L'obligation de preuve relative au raffinement de la précondition d'une opération, générée à partir de la PO 2.4.6 est :

   $$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow P$$

   Cette obligation de preuve peut être décomposée en plusieurs POs à prouver de la forme :

   $$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow P_i$$

   où P_i est la partie de la condition P qui est spécifiée dans le message $M4$. Ce type de PO fausse est dénoté $po9$.
   Par conséquent, corriger cette PO revient à corriger ses hypothèses A, E, B, I, J et P_a et son but P_i.
2. Erreur dans la preuve du corps de l'opération raffinée identifiée par le message $M6$. L'obligation de preuve relative au raffinement du corps d'une opération, générée à partir de la PO 2.4.6 est :

   $$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \Rightarrow [Q]\neg[Q']\neg I$$

   Cette obligation de preuve peut être décomposée en plusieurs POs à prouver de la forme :

   $$A \ \wedge \ E \ \wedge \ B \ \wedge \ I \ \wedge \ J \ \wedge \ P_a \ \wedge \ H \Rightarrow B_i$$

où H un ensemble d'hypothèses dérivées. H a comme origine les formules implicatives de l'invariant I ou les substitutions gardées dans Q. On peut encore décomposer ce type d'erreur selon la partie de l'invariant qui est impliquée dans la PO considérée.

- Dans le cas où I_k lie les variables de la spécification et les variables de la spécification qu'elle raffine (invariant de collage), la formule B_i est générée à partir de l'application de $[Q]\neg[Q']$ sur $\neg I_k$. Ce type de PO fausse est dénoté $po10$.
- Dans le cas contraire où I_k ne contient pas de variable de la spécification antérieure, la formule B_i est générée à partir de l'application de $[Q]$ sur I_k. Ce type de PO fausse est dénoté $po11$.

3. Erreur dans la preuve du corps de l'opération raffinée à cause d'une précondition non vérifiée d'un appel d'une opération $op_k = K|L$ avec $K = K_1 \wedge ... \wedge K_n$. Cette erreur est identifiée par le message $M5$. Rappelons l'obligation de preuve relative à l'appel d'une opération op_k dans une opération raffinée, cette PO élémentaire est générée à partir de la PO 2.4.6.

$$A \wedge E \wedge B \wedge I \wedge J \wedge P_a \Rightarrow [Q_i]K_i$$

où Q_i sont les substitutions qui précèdent l'appel de op et où K_i est la partie de K qui a déclenché l'erreur. Cette obligation de preuve peut être décomposée en plusieurs POs à prouver de la forme :

$$A \wedge E \wedge B \wedge I \wedge J \wedge P_a \wedge H \Rightarrow B_i$$

On a :

- H a comme origine les substitutions gardées de Q_i.
- La formule B_i est générée à partir de l'application de la substitution Q_i sur la condition K_i.

Ce type de PO fausse est dénoté $po12$.

Par conséquent, corriger les hypothèses revient à corriger I, J, P et Q_i. Effectuer des corrections sur le but B_i revient à corriger Q_i ou K_i ou les deux simultanément.

Stratégies de corrections

Dans ce qui précède, nous avons présenté des cas élémentaires de correction selon le type de PO fausse. Corriger une spécification revient souvent à enchaîner plusieurs corrections élémentaires. Ceci nous emmène à étudier les stratégies permettant de guider l'enchaînement de plusieurs corrections élémentaires. Nous proposons les deux stratégies suivantes.

- **Stratégie de correction par ordre.** Nous considérons les erreurs générées selon leur ordre de détection. Par conséquent, on a une stratégie globale implicite qui traite les erreurs selon leur ordre.
- **Stratégie de correction en cascade.** Certaines erreurs peuvent être corrigées par la modification de l'invariant. Dans la plupart des cas, modifier l'invariant influe sur la correction des autres parties de la spécification. Cette modification peut avoir comme répercutions :
  - La correction des erreurs qui apparaissent dans les autres opérations. Alors la correction faite est confirmée.
  - L'introduction de nouvelles erreurs dans les autres opérations. Dans ce cas la correction faite peut être soit mise en cause, soit elle va entraîner d'autres corrections dans d'autres opérations ou dans l'invariant.

  Si nous détectons des erreurs introduites par la modification de l'invariant, alors nous mettons en cause cette dernière modification ou nous terminons la correction des opérations concernées.

2.7 Conclusion

La méthode B est une méthode de développement de logiciels prouvés, bien outillée. Dans ce chapitre, nous avons présenté les fondements de la méthode, en particulier le langage associé et la notion de raffinement. Nous avons rappelé que la méthode offre un développement modulaire et que des obligations de preuve permettent de vérifier les différentes étapes de spécification des systèmes.

Nous avons également présenté brièvement le prouveur de l'atelier B. Ceci nous a permis d'introduire notre étude sur la détection des erreurs, l'analyse et la correction des spécifications B, en nous basant sur le retour de la preuve de prouveur de l'atelier B. Cette étude peut être caractérisée comme étant une étude statique puisqu'elle est basée sur la forme des obligations de preuve indépendamment de leur sémantique. Les chapitres 5 et 6 traitent des applications possibles de cette étude en tenant compte des informations sur la sémantique des spécifications.

3

Adaptation logicielle statique

Sommaire

3.1 Introduction 33
3.2 Interopérabilité 34
3.2.1 Objets, composants, services et interfaces 34
3.2.2 Niveaux d'interopérabilité 35
3.2.3 Incompatibilités 38
3.3 Quelques approches d'adaptation 38
3.3.1 Assemblage des composants 39
3.3.2 Adaptation des services 44
3.3.3 Recherche, composition et adaptation des composants 47
3.3.4 Composition et adaptation des connecteurs 50
3.3.5 Discussion 51
3.4 Caractéristiques d'une approche d'adaptation 51
3.4.1 Niveaux d'interopérabilité 51
3.4.2 Spécification 53
3.4.3 Spécification semi-automatique vs. manuelle 55
3.4.4 Types d'adaptation 56
3.4.5 Processus de construction de l'adaptateur 57
3.5 Conclusion 58

3.1 Introduction

L'adaptation est un sujet très étudié dans le contexte de la réutilisation et de l'intégration de logiciels. La fonction de base de l'adaptation est d'apporter des solutions aux problèmes d'incompatibilités entre composants. L'adaptation permet également d'assurer le respect de certaines propriétés décrites dans des formalismes différents.

De nombreux travaux ont considéré ce sujet : certains ont consisté à définir des adaptateurs, d'autres ont considéré l'adaptation comme la construction d'une enveloppe permettant l'encapsulation d'un ou plusieurs composants et donc la définition d'un nouveau composant.

Les incompatibilités entre composants sont le point de départ d'une approche d'adaptation. Elles dépendent de la description des interfaces et des propriétés à tester entre les interfaces. Les plates-formes actuelles prennent en compte l'adaptation seulement au niveau syntaxique.

Cependant, la plupart des travaux récents sur l'adaptation se sont intéressés à l'adaptation aux niveaux sémantique, protocole et qualité de service.

L'adaptation peut avoir lieu lors des différentes phases du cycle de vie du logiciel et par conséquent les techniques d'adaptation peuvent être classifiées selon la phase à laquelle elles s'appliquent. Nous distinguons deux types d'adaptation : statique et dynamique. L'adaptation statique s'applique durant la phase de conception alors que l'adaptation dynamique a lieu lors de l'exécution d'un système. Dans ce chapitre nous nous intéressons à l'adaptation statique.

Nous commençons par introduire la notion d'interopérabilité en Section 3.2. Dans la section 3.3, nous passons en revue quelques approches d'adaptation. Nous présentons ces approches en les regroupant selon le problème considéré. Nous présentons en Section 3.3.1 les travaux ayant étudié l'assemblage de composants. Dans cette catégorie, nous distinguons trois types d'approches : assemblage entre deux interfaces [YS97, SR02, BBC05, BCP06], assemblage de plusieurs composants [IT03, TA06, CPS06, CPS08] et assemblage des composants hiérarchiques [AAA07]. Nous donnons en Section 3.3.3 quelques travaux ayant pour but la recherche, la composition et l'adaptation des composants [Hem05, MA03]. Les approches d'adaptation de services sont ensuite présentées en Section 3.3.2, elles considèrent les problèmes de connexion entre deux services [DSW06], ainsi que le remplacement d'un service par un autre [BCG+05, MNBM+07]. Nous résumons ces approches en Section 3.3.5 et nous présentons quelques unes de leurs limites. La description des principales caractéristiques d'une approche d'adaptation est abordée dans la section 3.4.

3.2 Interopérabilité

La notion d'interopérabilité [VHT99, VHT00, Weg96] est fondamentale dans le domaine de l'ingénierie des systèmes. L'interopérabilité intervient dans plusieurs contextes tels que la réutilisation des composants, les systèmes distribués, l'assemblage de composants, l'architecture des logiciels, l'architecture SOA (*e.g.*, Websevices), *etc.*

L'interopérabilité est définie comme la capacité de deux ou plusieurs composants logiciels à coopérer indépendamment des différences concernant leurs langages d'implantation, leurs interfaces et leurs plate-formes et leurs environnements d'exécution [Kon95, Weg96]. Elle repose sur la présence de standards ouverts pour la description des interfaces. L'interopérabilité se base essentiellement sur deux concepts : la compatibilité et le remplacement. La compatibilité caractérise les connexions entre les interfaces. Le remplacement (ou la substitution) est la capacité d'une interface à être substituée par une autre interface sans effectuer de modifications sur l'application.

3.2.1 Objets, composants, services et interfaces

L'interopérabilité est très étudiée dans le cadre de la technologie orientée objet, en particulier pour le développement des applications hétérogènes. La programmation orientée objet (POO) est basée sur les classes qui contiennent des données et des opérations. Les objets sont des instances des classes. La POO reste encore la plus considérée pour le développement de logiciel. Pourtant, elle requiert une interaction forte entre les éléments d'un système et par conséquent il est souvent difficile de séparer entre les différents objets.

Le paradigme composant a un niveau d'abstraction plus élevé que le paradigme objet, il est ainsi mieux adapté à une industrie de réutilisation. Les composants sont plus autonomes par rapport au système et moins dépendants entre eux. Une définition consensuelle d'un composant est :
« A software component is a unit of composition with contractually specified interfaces and

explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. » [Szy99]

Les technologies orientées services sont ensuite apparues, en garantissant un couplage très faible et en se basant sur un ensemble de standards ouverts. Un service consiste en une fonction ou une fonctionnalité bien définie. Il s'agit d'un composant autonome qui ne dépend d'aucun contexte ou service externe. Il comporte des opérations qui constituent autant d'actions spécifiques que le service peut réaliser.

Ces différentes technologies peuvent être utilisées ensemble. En effet, les services sont normalement construits en se basant sur des composants. La relation entre les composants et les objets est similaire à celle entre services et composants puisque les composants peuvent être construits en se basant sur un ensemble d'objets.
Un point commun entre ces différents paradigmes est l'encapsulation des fonctionnalités requises et fournies. Ces fonctionnalités sont décrites par des interfaces qui peuvent être spécifiées par plusieurs formalismes selon l'information qu'on veut inclure et le niveau de détail de la spécification. La plupart des problèmes liés à l'interopérabilité sont les mêmes pour les objets, les composants ou les services.

3.2.2 Niveaux d'interopérabilité

Nous distinguons différents niveaux d'interopérabilité [BBG+06, CMP06], illustrés dans la Figure 3.1 : syntaxique, sémantique, protocole et service (QoS).

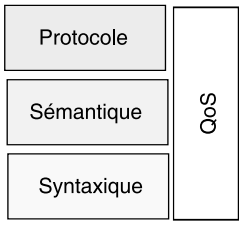


FIGURE 3.1 – Classification des niveaux d'interopérabilté

Dans cette section, nous présentons d'une manière brève les concepts de base qui sont en rapport avec l'interopérabilité. En particulier, nous nous intéressons aux niveaux syntaxique, protocole et sémantique.

Niveau syntaxique

La description syntaxique des interfaces est constituée des signatures des différents éléments constituant l'interface d'un composant. Elle comprend les données manipulées par l'interface (*e.g.*, les noms des données et leurs types) et les opérations (*e.g.*, les noms des opérations, les types de leurs arguments et les valeurs de retour). La définition des données des interfaces est optionnelle. En effet, plusieurs approches de description d'interfaces ne supportent pas les données.

Les plate-formes actuelles de composants permettent de décrire correctement le niveau syntaxique du composant grâce à des langages de description d'interface (IDL, pour « Interface description language »).

Il existe principalement deux types de vérification qui peuvent être effectuées entre des composants : la compatibilité et la substitution.

- La compatibilité peut être décrite par le fait que deux composants peuvent fonctionner ensemble correctement s'ils sont connectés, *i.e.* tous les messages et les données échangés sont bien reconnus des deux côtés.
- La substitution (connue aussi comme le sous-type dans le contexte objet) d'un composant A par un composant B peut être définie par le fait que toutes les fonctionnalités offertes par A sont aussi offertes par B [ZW95].

Niveau protocole

La description du protocole d'un composant est primordiale pour vérifier l'interoperabilité entre composants. En effet, plusieurs travaux se sont intéressés à l'extension des IDLs pour prendre en compte l'aspect protocole [YS97, PV02, CFP+03, AAA07] en considérant des BIDLs (« Behavioural IDL »). À ce niveau, les interfaces décrivent principalement l'ordre relatif d'utilisation des opérations d'un composant ainsi que l'ordre dans lequel un composant appelle les opérations d'autres composants. La compatibilité et la substitution des composants sont définies au niveau protocole.
Les deux conditions suivantes sont les plus étudiées pour vérifier la compatibilité entre composants : préservation des restrictions d'interaction imposées par les composants quand ils interagissent et l'absence de blocage. Certaines approches définissent, en plus, d'autres propriétés à vérifier par la communication telles que les propriétés de sûreté.
La substitution d'un composant A par un composant B peut être effectuée, si ces deux conditions sont satisfaites [YS97] : d'abord, il faut vérifier que tous les messages acceptés par A sont aussi acceptés par B. Ensuite, les messages sortant de B pour la réalisation des services de A (messages de réponse et des opérations requises) sont un sous-ensemble des messages sortant de A. L'ordre relatif entre les messages sortant et entrant des deux composants est consistant.

Niveau sémantique

L'interopérabilité sémantique des composants est un sujet difficile vu qu'il est généralement impossible de donner la sémantique complète d'un composant.

Parmi les spécifications de la sémantique, celle qui consiste à spécifier les opérations d'une interface en termes de préconditions et postconditions est l'une des plus utilisée. Elle considère également la définition des propriétés invariantes sur le modèle de données de l'interface considérée.
À ce niveau, la compatibilité est aussi la capacité de deux composants de communiquer correctement ensemble lorsqu'ils sont connectés. En d'autre termes, le comportement fourni par un composant doit être en accord avec les exigences du comportement attendu de l'autre. Par exemple, nous pouvons citer la conception par contrat [Mey97] qui est de plus en plus supportée par les langages de programmation (*e.g.*, Eiffel, Java avec iContract, C++). Dans le contexte de la conception par contrat, les préconditions, les postconditions et l'invariant permettent la formalisation du *contrat*. Celui-ci est défini de la manière suivante :
« Pourvu que le client appelle la méthode d'un contractant dans un état où l'invariant de classe et la précondition de cette méthode sont satisfaits, ce contractant promet que lorsque la méthode sera exécutée, le travail nécessaire à la satisfaction de la postcondition sera effectué et l'invariant de classe sera toujours satisfait. »

En se basant sur cette même définition des interfaces en fonction des pré/postconditions et des invariants, d'autres travaux se sont intéressés à définir une relation de substitution appelée aussi sous-type de comportement [Ame91, LW94]

Dans [ZW97], Zaremski et Wing s'intéressent à la spécification de la sémantique des opérations, en terme de pré/postconditions. Ils définissent plusieurs conditions d'appariement entre composants. Ces conditions n'expriment pas seulement des relations de compatibilité et de substitution, mais aussi des relations telles que la spécialisation et la généralisation. Les conditions d'appariement étudiées sont décrites de façon détaillée dans ce qui suit.

Soient op_1 et op_2 deux opérations, pre_1 et pre_2 leurs préconditions respectives et $post_1$ et $post_2$ leur postconditions respectives. Les conditions d'appariement possible entre op_1 et op_2 sont classifiées en deux catégories :

(a) Des conditions d'appariement Pre/Post permettant de lier les préconditions et les postconditions des opérations. Cinq différentes conditions d'appariement Pre/Post sont définies.
- La condition d'appariement Pre/Post exact(« Exact Pre/Post Match »). Elle exige que les spécifications des deux opérations soient équivalentes. Formellement, cette condition d'appariement est définie comme suit :
$$match_{E\text{-}pre/post}(op_1,\ op_2) = (pre_1 \Leftrightarrow pre_2) \wedge (post_1 \Leftrightarrow post_2)$$
- La condition d'appariement « Plug-In ». Dans ce cas, l'opération op_1 est appariée avec op_2 si sa précondition est plus faible que celle de op_2 et sa postcondition est plus forte que celle de op_2. Cette condition permet de vérifier que l'opération op_1 peut se substituer à l'opération op_2 sans engendrer de faux résultats. Formellement, cette condition d'appariement est définie comme suit :
$$match_{plug\text{-}in}(op_1,\ op_2) = (pre_2 \Rightarrow pre_1) \wedge (post_1 \Rightarrow post_2)$$
- La condition d'appariement « Plug-In Post ». Il s'agit d'un affaiblissement de la condition d'appariement « Plug-In ». En particulier, cette condition d'appariement est utile, si on veut considérer seulement les conditions sur la postcondition (ce qui nous intéresse c'est l'effet de l'opération). Formellement, cette condition d'appariement est définie comme suit :
$$match_{plug\text{-}in\text{-}post}(op_1,\ op_2) = (post_1 \Rightarrow post_2)$$
- La condition d'appariement « Plug-In » gardé. Il s'agit d'une forme de la condition d'appariement « Plug-In » telle que la condition sur la postcondition est vérifiée seulement pour les valeurs en entrée permises par la précondition. On ajoute pre_1 dans l'hypothèse de la condition sur la relation. Cette condition d'appariement est définie comme suit :
$$match_{guarded\text{-}plug\text{-}in}(op_1,\ op_2) = (pre_2 \Rightarrow pre_1) \wedge ((pre_1 \wedge post_1) \Rightarrow post_2)$$
- La condition d'appariement Post gardé (« guarded postmatch »). Elle est définie comme la condition précédente mais en enlevant les conditions sur les préconditions. Formellement, elle est définie comme suit :
$$match_{guarded\text{-}post}(op_1,\ op_2) = ((pre_1 \wedge post_1) \Rightarrow post_2)$$

(b) Des conditions d'appariement de prédicats permettant de lier les prédicats de spécification des opérations $pred_1 = (pre_1 \Rightarrow post_1)$ et $pred_2 = (pre_2 \Rightarrow post_2)$. Ces conditions sont utiles si on a besoin de comparer les spécifications des opérations globalement et non de comparer leurs parties indépendamment. Trois conditions différentes d'appariement de prédicats sont définies.
- La condition d'appariement de prédicats exacts (« Exact Predicates Match »). Cette condition est plus faible que la condition d'appariement Pre/Post exact. Elle est définie comme suit :
$$match_{E\text{-}pred}(op_1,\ op_2) = (pred_1 \Leftrightarrow pred_2) = ((pre_1 \Rightarrow post_1) \Leftrightarrow (pre_2 \Rightarrow post_2))$$
- La condition d'appariement généralisé (« Generalized Match »). Cette condition est plus faible que la condition d'appariement « Plug-In ». Elle est définie comme suit :
$$match_{gen\text{-}pred}(op_1,\ op_2) = (pred_1 \Rightarrow pred_2) = ((pre_1 \Rightarrow post_1) \Rightarrow (pre_2 \Rightarrow post_2))$$
- La condition d'appariement spécialisé (« Specialized Match »). Cette condition est l'in-

verse de la condition d'appariement généralisé. Elle est définie comme suit :
$match_{spcl\text{-}pred}(op_1,\ op_2) = match_{gen\text{-}pred}(op_2,\ op_1) = (pred_2 \Rightarrow pred_1) = ((pre_2 \Rightarrow post_2) \Rightarrow (pre_1 \Rightarrow post_1))$

La Figure 3.2 résume les conditions d'appariement et les relations entre elles, évoquées dans [ZW97].

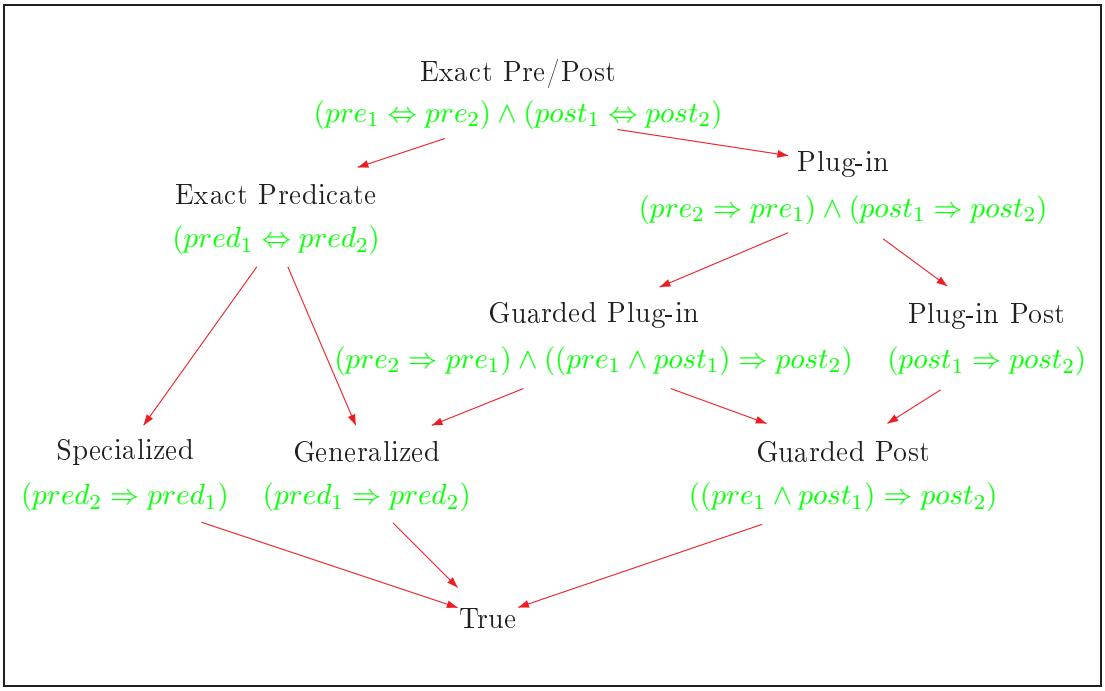


FIGURE 3.2 – Treillis de correspondances de spécification

3.2.3 Incompatibilités

Les incompatibilités peuvent se produire à n'importe quel niveau d'interopérabilité cité précédemment :

- Au niveau signature, les noms des opérations peuvent être différents, les paramètres dans des ordres différents ou de différents types, *etc.*
- Au niveau protocole, les protocoles des deux composants peuvent être incompatibles. Par exemple, une non-conformité dans l'ordre d'une séquence de messages.
- Au niveau sémantique, on peut trouver, par exemple, deux opérations appariées mais avec des sémantiques différentes.

L'adaptation logicielle est donc nécessaire à chacun des niveaux d'interopérabilité.

3.3 Quelques approches d'adaptation

Cette section a pour but de survoler quelques travaux effectués dans le domaine de l'adaptation logicielle statique.

3.3.1 Assemblage des composants

Dans le contexte des systèmes basés sur l'assemblage des composants (CBSE), de nombreux travaux se sont intéressés à l'adaptation des composants. Dans cette section, nous présentons quelques-uns de ces travaux. Nous regroupons ces travaux selon le problème considéré : assemblage de deux interfaces de deux composants, assemblage d'un ensemble de composants, assemblage de composants hiérarchiques.

Connexion entre deux interfaces

Approche proposée par Yellin et Strom (YS) [YS97]. Commençons par citer le célèbre article de Yellin et Strom. Cet article est parmi les premiers travaux qui ont défini les fondations formelles de la notion d'adaptation entre composants. Les auteurs considèrent l'adaptation de l'assemblage de deux interfaces, aux niveaux syntaxique et protocole.
Description des interfaces. L'interface d'un composant est décrite par les signatures des messages envoyés et reçus par ce composant ainsi que par son protocole. Des machines à états finis (FSM, pour« Finite state machine ») sont utilisées pour la description des protocoles des interfaces. Les transitions des FSMs considérées sont étiquetées par les messages envoyés et reçus des interfaces. En outre, ces FSMs sont caractérisées par un seul état initial et plusieurs états finals.
Processus d'adaptateur. L'adaptateur est décrit par une FSM qui possède des interfaces vers les deux composants à connecter. Puisque tous les messages échangés par les composants doivent passer par l'adaptateur, les transitions de l'adaptateur sont étiquetées par les messages envoyés et reçus entre l'adaptateur et les deux composants à connecter. Lorsqu'une transition dans le protocole de l'adaptateur est déclenchée par l'envoi ou la réception d'un message, l'adaptateur peut exécuter un ensemble d'actions (*e.g.*, sauvegarder les paramètres d'un message reçu ou envoyé afin de les utiliser dans la suite).
Une spécification déclarative est introduite permettant de relier les paramètres et les messages des deux composants et ainsi l'automatisation de la construction de l'adaptateur. Cette spécification est définie par un ensemble de règles de correspondance entre les interfaces à adapter. Les règles de correspondance considérées sont complètes et non ambiguës, par hypothèse. Un algorithme est proposé pour synthétiser la spécification de l'adaptateur à partir de sa spécification déclarative. L'algorithme prend en entrée les protocoles des deux composants et un ensemble de règles de correspondances I et fournit en sortie soit un adaptateur valide par rapport à I, soit le résultat qu'un tel adaptateur n'existe pas.
Compatibilité. Les auteurs définissent la compatibilité de l'adaptateur comme suit : (i) un adaptateur doit supporter un protocole compatible avec les deux protocoles des composants à adapter et (ii) il doit satisfaire les contraintes imposées par les règles de correspondance considérées.
Le protocole de l'adaptateur est compatible avec les protocoles des deux composants à connecter, si toutes les collaborations entre eux vérifient ces deux conditions :

- Pas de réception non spécifiée. C'est-à-dire qu'à chaque fois qu'un composant envoie un message, l'autre composant doit pouvoir le recevoir.
- Absence de blocage. Une collaboration se termine et les deux composants sont dans un état final ou cette collaboration peut continuer.

Approche proposée par Reussner et Schmidt (RS) [SR02]. Les auteurs développent des algorithmes pour l'adaptation des composants traitant un sous-ensemble de problèmes de synchronisation entre des composants concurrents (niveau protocole). Les composants sont connectés par l'intermédiaire de leurs interfaces. Deux types d'interfaces sont considérés : les interfaces

requises et les interfaces fournies. En particulier, les auteurs présentent des algorithmes d'adaptation au niveau protocole entre une ou plusieurs interfaces requises connectées à une ou plusieurs interfaces fournies.
Description des interfaces et compatibilité Chaque interface requise ou fournie est décrite par la signature de ses opérations et par son protocole. Les protocoles des interfaces sont décrits en termes de machines à états finis (FSM).
La compatibilité entre deux interfaces requise et fournie est définie par la condition suivante : toutes les séquences d'appels générées par la FSM de l'interface requise sont acceptées par la FSM de l'interface fournie.
Processus d'adaptation par type. Le protocole de l'adaptateur est également décrit par une FSM. La compatibilité d'un adaptateur est considérée de la même manière que la compatibilité des autres composants.
Trois types d'adaptation sont supportés. Pour ces différents types d'adaptation, des algorithmes sont présentés. Ces algorithmes permettent de générer une FSM décrivant le protocole d'un adaptateur entres deux composants à partir des FSMs de ces composants. Pour chaque type d'adaptation considéré, un algorithme spécifique est proposé.

- « 1 :n-Adaptateur ». Cet adaptateur est généré pour un composant qui utilise un ensemble de composants. L'adaptateur répartit les appels du premier composant au bon composant. Dans ce cas, le rôle de l'adaptateur est de combiner le protocole des interfaces fournies en un seul protocole. Ainsi, la FSM associée à l'adaptateur est obtenue par le calcul du produit cartésien des FSMs associées aux interfaces fournies.
- « n :1-Adaptateur ». À l'inverse de l'adaptation précédente, ce type d'adaptation est généré pour deux ou plusieurs composants qui utilisent simultanément un autre composant. Le rôle de l'adaptateur est de combiner le protocole des interfaces requises en un seul protocole. De même, la FSM associée à l'adaptateur est obtenue par le calcul du produit cartésien des FSMs associées aux interfaces requises. Cependant, il faut tenir compte des contraintes supplémentaires de synchronisation, puisque plusieurs composants ont besoin des mêmes ressources (une seule interface fournie). L'algorithme proposé réduit la FSM obtenue par le produit cartésien afin de vérifier les contraintes de synchronisation.
- Adaptateur de changement de protocole. C'est dans le cas où deux composants sont connectés ensemble mais leur protocole sont incompatibles. L'algorithme proposé commence par construire une première version de la FSM associée à l'adaptateur. Cette version contient tous les comportements possibles entre les interfaces considérées avec une synchronisation par rapport aux messages communs (les messages de même nom mais de signes contraires). L'étape suivante consiste à détecter les erreurs dans la FSM construite et à la corriger permettant ainsi d'obtenir la FSM finale de l'adaptateur. À noter que cet algorithme ne peut s'appliquer que s'il existe des messages en commun entre les deux interfaces à assembler. Deux cas d'incompatibilité sont identifiés, pour chacun d'eux une adaptation spécifique est proposée.
  - le problème de l'état initial. Pour résoudre ce problème, les auteurs proposent de préfixer la communication par une séquence d'événements appropriés.
  - le problème des états finals. Dans ce cas, les auteurs proposent de postfixer la communication par une séquence d'événements appropriés.

Outil. Cette approche est implémentée dans l'outil « CoConut/J suite » [Reu03]. Des algorithmes et des outils pour spécifier et analyser les interfaces des composants sont également présentés dans [Reu03].

Approche proposée par Alfaro *et al.* (AHPS) [dAH01]. Les auteurs présentent une méthodologie formelle pour la description des protocoles des composants. Cette description supporte la vérification automatique de la compatibilité lors de l'assemblage de deux interfaces. L'élément clé de l'approche est l'interprétation des protocoles comme un jeu entre un composant et son environnement, par l'utilisation de l'algorithme « game-theoretic » pour la vérification de la compatibilité. Dans [PdAHSV02] l'algorithme « game-theoretic » est non seulement appliqué pour vérifier la compatibilité des interfaces mais aussi pour synthétiser l'interface d'un adaptateur lorsque les interfaces ne sont pas compatibles. Cet adaptateur est nommé un convertisseur de protocole.
Description des interfaces et compatibilité. Les protocoles des interfaces à connecter sont décrits par des automates. Une description des propriétés qui doivent être vérifiées par la communication entre deux composants, est en plus fournie. L'assemblage entre deux interfaces doit vérifier des propriétés décrivant ces deux types de conditions :

- des conditions de sûreté (« safety ») exprimées par un ou plusieurs automates, dont l'alphabet est le produit cartésien des alphabets des deux automates à connecter,
- des conditions d'équité (« fairness ») décrites en logique temporelle.

Processus d'adaptation. Le protocole de l'adaptateur est décrit par un automate, obtenu en calculant le produit des automates à connecter et ensuite en éliminant tous les états et toutes transitions qui violent les propriétés spécifiées. Le processus de synthèse de l'adaptateur est basé sur l'algorithme « game-theoretic ». Cet algorithme retourne la spécification des réarrangements appropriés (appelée stratégie gagnante) entre deux interfaces et génère ainsi de manière automatique le convertisseur de protocole. Dans le cas contraire, cet algorithme retourne un échec.

Approche proposée par Bracalia *et al.* (BBCP) [BBC05, BCP06]. Les auteurs présentent une méthodologie pour l'adaptation de deux interfaces de composants au niveau syntaxique et protocole. Cette approche est fondée sur l'article de Yellin et Strom [YS97]. Les principales étapes de leur approche sont les suivantes.
Description des interfaces. Elle comprend les signatures et les protocoles des interfaces. Les protocoles des composants sont spécifiés en utilisant une algèbre de processus, plus précisément un sous-ensemble du π-calcul est considéré.
Spécification de l'adaptateur. Les auteurs présentent une notation simple pour spécifier l'adaptateur entre deux composants. Cette spécification correspond à un ensemble de correspondances entre les opérations des deux composants à adapter. Ces correspondances sont définies par des règles qui lient les opérations et les paramètres des deux composants.
Un adaptateur concret est automatiquement généré à partir des règles de correspondance et des interfaces des deux composants. Un algorithme qui construit l'adaptateur est proposé, il vérifie les conditions suivantes :

- il n'existe pas de blocage entre les deux interfaces considérées et l'adaptateur,
- toutes les règles de correspondance sont bien satisfaites,
- les deux protocoles des deux interfaces sont bien respectés.

Dans le cas contraire, l'algorithme échoue.
Types d'adaptation. Ils dépendent des types de correspondances entre les opérations des deux interfaces :

- Correspondance un-à-un. Elle permet de faire correspondre une opération d'une interface à une opération d'une autre interface. Ces deux opérations peuvent être de noms différents ou de paramètres différents.
- Correspondances multiples. Elles permettent de faire correspondre des groupes d'opérations

des deux interfaces.
- Opération sans correspondance. Certaines opérations dans une interface ne possèdent aucune correspondance avec les opérations de l'interface à connecter. Des règles qui lient ces opérations à « none » sont alors définies.
- Correspondance indéterministe d'opérations. Elle permet de faire correspondre à une opération d'une interface, un choix d'opérations d'une autre interface.

Assemblage d'un ensemble de composants

Approche proposée par Inverardi et Tivoli (IT) [IT03, Tiv05, TA06]. Les auteurs présentent une approche d'adaptation d'un assemblage de composants au niveau protocole. Ils proposent la construction d'un adaptateur entre les différents composants à connecter, nommé connecteur.
Description des interfaces. Le comportement des composants est exprimé par des STEs. Ces derniers sont générés à partir de diagrammes de séquences : des bMSCs (« basic message sequence chart ») et des hMSCs (« high level message sequence chart »), qui donnent une spécification partielle des composants.
Processus d'adaptation. La construction du protocole de l'adaptateur comprend un ensemble d'étapes. Premièrement, le protocole de l'adaptateur est synthétisé à partir des interfaces des composants. À ce niveau, le STE du protocole de l'adaptateur contient toutes les interactions possibles entre les composants, en considérant la synchronisation des évènements en commun. L'adaptateur joue ainsi le rôle d'un simple composant de routage. Deuxièmement, une spécification décrivant les propriétés de coordination entre les composants est fournie. Ces propriétés sont initialement spécifiées en logique temporelle LTL, puis traduites vers des automates de Büchi. Le STE de l'adaptateur est alors exploré afin de trouver et supprimer les états de blocage ou de comportement indésirable (*i.e.* violer les propriétés de coordination). La correction d'un adaptateur est donc vérifiée par construction et elle est définie par : (i) l'absence de blocage et (ii) le respect de la politique de coordination.
Outil. Cette approche est implémentée par l'outil SYNTHESIS [SYN]. Cet outil a été validé et appliqué à l'assemblage des composants Microsoft COM/DCOM.

Approche proposée par Canal *et al.* (CPS) [CPS06, CPS08]. Les auteurs proposent une approche d'adaptation de plusieurs composants, qui communiquent ensemble, aux niveaux syntaxique et protocole.
Description des interfaces. Les interfaces des composants sont décrites par des STEs ou en π-calcul.
Processus d'adaptation. La notion de *contrat* d'adaptation est introduite. Elle est définie par la description des contraintes à respecter pour assurer la communication des composants considérés. Des STEs dont les étiquettes sont des vecteurs, sont utilisés pour décrire les *contrats* d'adaptation.
L'adaptateur est alors généré automatiquement à partir d'une description comportementale des interfaces et d'un *contrat* d'adaptation. Deux algorithmes pour générer un adaptateur avec absence de blocage sont proposés : le premier est basé sur le produit synchronisé et le second est basé sur les réseaux de Petri.
Types d'adaptation. Les algorithmes proposés permettent de corriger les erreurs en effectuant :
- des renommages
- des préfixages
- des réordonnancements

– des suppressions de comportements non désirés

Outil. Cette approche est implémentée dans l'outil *Adaptator* [Ada07].

Assemblage de composants hiérarchiques

Dans les modèles de composants hiérarchiques (*e.g.*, Kmelia [AAA06], Fractal, Sofa [PV02], UML 2.0, *etc.*), différents composants peuvent être assemblés en créant un nouveau composant. Celui-ci peut lui même être assemblé avec d'autres composants dans un niveau de hiérarchie supérieur. Les composants hiérarchiques présentent un intérêt croissant : d'une part, ils permettent de modéliser des niveaux de hiérarchie dans un système. D'autre part, ils cachent pour chaque niveau la complexité des niveaux inférieurs.

Approche proposée par André *et al.* (AAA) [AAA07]. Les auteurs proposent une approche d'adaptation permettant de résoudre des problèmes d'incompatibilité pour l'assemblage des composants dans le modèle Kmelia [AAA06]. Ce modèle est basé sur les services et il supporte des interfaces hiérarchiques (HBIDL, « Hierarchical Behavioural Interface Description Language »).

Description d'un composant Kmelia. Un composant Kmelia est décrit par : (i) un espace d'états, (ii) un invariant et (iii) une interface définie principalement par les services requis et offerts de ce composant. La description d'un service comprend une signature, une déclaration des variables locales, des pré/postconditions, les dépendances du service (*i.e.* ses sous-services, et les services qu'il requiert) et un système de transition étiqueté étendu (eLTS, « extended labelled transition system »). L'assemblage des composants Kmelia consiste à assembler leurs services requis avec les services fournis. En vue de supporter les communications entre ces services, un canal implicite est introduit.

Compatibilité. La vérification d'un assemblage consiste à vérifier la compatibilité entre les services, par rapport aux niveaux suivants :

– signature (simple) des services (*e.g.*, les arguments, la compatibilité des types, *etc*),
– signature *profonde* (*e.g.*, les liens avec les sous-services)
– contrat d'assemblage (*i.e.* la compatibilité entre les pré/postconditions)
– vérification dynamique du service (*i.e.* vérification des interactions entre les eLTSs)

Processus d'adaptation. Les auteurs proposent un processus pour détecter les problèmes d'adaptation dans une architecture HBIDL et particulièrement ils donnent des solutions d'adaptation pour le modèle Kmelia. Ils supposent initialement que les correspondances entre les noms sont déjà faites. Ils commencent alors par identifier le niveau de compatibilité dans lequel se produit le problème. Ils identifient ensuite le type du problème d'adaptation considéré. Ces étapes permettent de générer un adaptateur ainsi que de vérifier sa compatibilité.

Types d'adaptation. Deux types d'adaptation sont identifiés pour les interfaces hiérarchiques :

– Adaptation des incompatibilités de type « paramètre vs. message ». Dans certains cas, un service d'un composant contient des paramètres mais le service correspondant dans un autre composant utilise des messages pour envoyer les valeurs des paramètres du premier service. Cette technique d'adaptation propose des solutions pour ce genre de problème.
– Adaptation des incompatibilités hiérarchiques. Cette adaptation permet de résoudre le non respect des niveaux hiérarchique entre deux services. Par exemple, lorsqu'un composant attend l'appel d'un service qui n'est pas hiérarchique et il est connecté avec un autre composant où le service correspondant est décrit avec des niveaux de hiérarchie.

Outil. Cette approche se base sur un langage de spécification appelé Kmelia et un prototype d'une boîte à outils (COSTO pour « Component Study Toolbox ») permettant de supporter le modèle Kmelia et la vérification des propriétés. Les techniques d'adaptation proposées ne sont pas encore

incluses dans l'outil.

3.3.2 Adaptation des services

L'architecture orientée services SOA (« Services oriented architecture ») est une architecture logicielle se basant sur un ensemble de services. Dans SOA, les ressources logicielles disponibles sont considérées comme des services. Ces services sont décrits d'une manière abstraite, indépendante de leur implémentation ils sont faiblement couplés et utilisent des protocoles standards.
Les services Web constituent une alternative prometteuse pour la mise en place d'une architecture SOA, on parle alors de WSOA (« Web SOA »). Ils sont basés sur le standard XML (« extensible markup language ») pour la description et l'échange des données. Les interfaces des services sont décrites dans le langage WSDL (« Web services description language ») et communiquent en utilisant des protocoles standards tel que SOAP au-dessus des protocoles de l'Internet, garantissant ainsi une architecture faiblement couplée et interopérable.

Plusieurs travaux relativement récents concernent le problème de l'adaptation des services, en particulier dans le contexte de WSOA. Nous détaillons par la suite quelques-uns de ces travaux.

Connexion entre deux services

Approche proposée par Dumas *et al.* (DSW) [DSW06]. Les auteurs proposent une approche d'adaptation entre une interface fournie par un service et une interface requise qui a besoin de ce service. Seule l'adaptation aux niveaux syntaxique et protocole est considérée dans cette approche.
Description des services. À un service deux types d'interfaces sont associés :

- une interface structurelle qui décrit les messages échangés entre les services (niveau syntaxique),
- une interface comportementale qui décrit l'ordre des messages échangés entre les services (niveau protocole),

Les auteurs se sont intéressés à la description des interfaces comportementales. Ces dernières sont définies par des schémas d'actions de communication. Une action de communication (ou action) est définie par un nom, un type (c-à-d une action envoyée ou reçue) et le type du message envoyé ou reçu par l'action. Formellement, les interfaces comportementales des services sont décrites par un ensemble de traces formées sur un alphabet généré à partir des actions de communication.
Processus d'adaptation. L'approche d'adaptation proposée s'appuie sur un modèle de transformation d'interface. Ce modèle est basé sur un ensemble d'opérateurs algébriques permettant d'exprimer différentes manières de lier deux interfaces comportementales. Les opérateurs prennent en entrée une interface comportementale, appelée interface source, et produisent en sortie une autre interface comportementale, appelée interface cible. Une interface source ou cible peut être une interface requise ou fournie.
Par conséquent, l'adaptation entre deux interfaces comportementales est définie par un ensemble de correspondances. Ces correspondances sont définies par un ensemble d'expressions de transformation d'interfaces $(E_1, ..., E_n)$.
Une notation visuelle est également présentée pour la description des correspondances entre les interfaces. L'adaptateur est généré à partir de ces notations visuelles, mais aucune solution n'est présentée pour la dérivation automatique d'un adaptateur. La vérification de l'adaptateur n'est pas traitée de manière exhaustive dans ce travail. En effet, seulement quelques conditions sur les correspondances ont été proposées pour éviter certains scénarios de blocage (« deadlock ») dans l'adaptation.

Types d'adaptation. Six opérateurs de transformation d'interface sont présentés :

- L'opérateur « Flow » décrit la transformation d'une action définie dans l'interface source qui devient une autre action dans l'interface cible.
- L'opérateur « Scatter » est appliqué quand une seule action dans l'interface source correspond à un ensemble d'actions dans l'interface cible.
- L'opérateur « Gather » est appliqué quand plusieurs actions définies dans l'interface source correspondent à une action dans l'interface cible.
- L'opérateur « Collapse » permet de lier un flot de messages constitué de plusieurs instances d'une même action à une seule action.
- L'opérateur « Burst » se base sur le même principe que l'opérateur *Collapse*, mais dans l'autre sens.
- L'opérateur « Hide » est appliqué quand une action dans l'interface source est non requise dans l'interface cible.

Remplacement d'un service par un autre

Approche proposée par Benatallah *et al.* (BCT) [BCG+05]. Les auteurs proposent une technique pour développer des adaptateurs qui permettent de remplacer un service $Service_2$ par un autre $Service_1$. Cette technique supporte principalement les niveaux syntaxique et protocole. Le rôle de l'adaptateur est de permettre au protocole P_1 du service $Service_1$ d'interagir comme le protocole P_2 du service $Service_2$. Par conséquent, l'adaptateur implémente le protocole de P_2 en appelant les méthodes du service $Service_1$ et son protocole doit être compatible avec P_1.
Description des services. Un service est décrit par :

- une interface qui correspond à l'ensemble des opérations supportées par le service (niveau syntaxique),
- un protocole de gestion (« Business protocol ») qui décrit les séquences permises d'échange de messages. Il est spécifié par un modèle de protocole présenté dans [BCT04].

Processus d'adaptation. Cette approche est fondée sur un ensemble de patterns d'erreurs. Dans ces patterns, le choix de l'adaptation à effectuer est défini selon les erreurs identifiées. Le langage utilisé pour définir les patterns est formé essentiellement par des notations BPEL avec d'autres annotations supplémentaires. La génération de l'adaptateur est faite à partir de ces patterns. Cependant, les auteurs ne présentent ni comment les patterns vont être composés pour décrire le comportement de l'adaptateur, ni comment l'adaptateur va être généré.
Types d'adaptation.

- Niveau opération (niveau syntaxique). Pour ce niveau deux patterns sont décrits.
  - le pattern SMP (« Signature Mismatch Pattern ») résout les erreurs de signature. Ceci se produit dans le cas où les opérations des deux services ont les mêmes fonctionnalités mais avec des noms différents, le nombre de paramètres (input et output) n'est pas le même ou les paramètres ne sont pas dans le même ordre ou de types différents.
  - le pattern PCP (« Parameter Constraints Pattern ») traite le cas où une opération d'un service fourni impose des contraintes sur les paramètres d'entrée qui sont moins restrictives que les paramètres de l'opération requise.
- Niveau protocole. Un ensemble de patterns est identifié pour l'adaptation des erreurs au niveau protocole.
  - Le pattern OCP (« Ordering Constraint Pattern ») est applicable dans le cas où deux protocoles P_1 et P_2 supportent les mêmes messages mais dans un ordre différent.
  - Le pattern EDP (« Extra Message Pattern ») est applicable dans le cas où le protocole P_1 émet un message que P_2 n'émet pas. Le pattern d'adaptation permet donc d'intercepter

ce message et de le supprimer.
- Le pattern MMP (« Missing Message Pattern ») est applicable quand le protocole P_2 émet un message que P_1 n'émet pas. Pour résoudre ce problème, le pattern d'adaptation génère un nouveau message.
- Le pattern OMP (« One to Many messages Pattern ») est utilisé quand le protocole P_2 a besoin d'un seul message pour accomplir certaines fonctionnalités, par contre le protocole P_1 a besoin de recevoir plusieurs messages pour accomplir cette fonctionnalité.
- Le pattern MOP (« Many to One message Pattern ») est utilisé quand le protocole P_2 a besoin de recevoir plusieurs messages pour accomplir certaines fonctionnalités, par contre le protocole P_1 a besoin d'un seul message.

Approche proposée par Motahari Nezhad *et al.*(MBMCC) [MNBM+07]. Les auteurs présentent une approche et un outil pour l'adaptation entre deux services. Cette approche permet de résoudre les incompatibilités au niveau syntaxique et protocole par l'intermédiaire d'un adaptateur de service.

Description d'un service. Un service est décrit par :
- Une interface permettant de déclarer toutes les opérations du service (Niveau syntaxique) et décrite par le langage WSDL.
- Un protocole de gestion définissant les contraintes d'ordre sur les opérations (Niveau protocole) et décrit par un STE.

Adaptation. L'adaptateur est défini par un STE, de manière semblable aux adaptateurs proposés dans [YS97]. Les états de ce STE sont composés des couples d'états des deux services à adapter. Les transitions sont étiquetées par les messages des deux services. Des actions peuvent également être associées à chaque transition. Les incompatibilités considérées sont un sous-ensemble des incompatibilités présentées dans [BCG+05].

Types d'adaptation. Soient SP le service du fournisseur et SC le service du client à adapter. Les différentes incompatibilités qui peuvent se produire entre ces deux services sont les suivantes :
- Signature d'un message. Un message dans l'interface SP a un nom ou un type différent dans l'interface SC.
- Deviser/ fusionner les messages. Un message dans SP correspond à plusieurs messages dans dans SC ou inversement.
- Message oublié/supplémentaire. Un ou plusieurs messages dans SP n'ont pas de correspondant dans SC ou inversement.
- Ordre des messages. La définition du protocole de SP peut attendre un message m dans un ordre différent par rapport au protocole de SC ou inversement.

Processus d'adaptation. Les principales étapes de cette approche d'adaptation sont :

1. Correspondances d'interfaces. Un support semi-automatique est fourni afin de définir des règles de correspondances entre les messages échangés des deux services à adapter. Pour cela, les approches d'appariement de schémas XML sont exploitées et elles sont raffinées et étendues.
2. Génération de la simulation de l'adaptateur. Un algorithme inspiré de celui proposé par Yellin et Strom [YS97] est présenté. Cet algorithme génère le protocole de l'adaptateur et un arbre des incompatibilités à partir des interfaces des deux services et des règles de correspondances. Cet arbre fournit une représentation concise pour la description de tous les blocages et les messages impliqués dans chaque blocage.
3. Traitement des erreurs de blocage. Cette étape est basée sur l'arbre des incompatibilités construite dans l'étape précédente. Les incompatibilités considérées sont générées soit

à cause d'une correspondance erronée, soit à cause d'une correspondance oubliée. L'analyse de l'arbre des incompatibilités guide l'utilisateur dans la mise en cause des règles de correspondance définies précédemment.

4. Génération de l'adaptateur final.

Outil. Le point fort de cette approche est qu'elle fournit un support semi-automatique pour identifier les incompatibilités syntaxiques et pour définir les règles de correspondance qui résolvent ces incompatibilités. Elle fournit également un support automatique pour l'identification des incompatibilités au niveau protocole et pour la génération d'un adaptateur.
Cette approche est supportée par un outil implanté dans IBM WID (« WebSphere Integration Developer »). Cet outil permet d'assister l'utilisateur dans les étapes de mise en correspondance, génération et analyse de l'arbre des incompatibilités ainsi que dans la génération de la spécification de l'adaptateur.

3.3.3 Recherche, composition et adaptation des composants

La recherche de composants est fondamentale dans l'ingénierie des systèmes à base de composants. L'augmentation du nombre de modèles de composants ainsi que l'avènement des composants sur étagère (COTS) nécessitent la mise en place de systèmes sophistiqués de recherche de composants.

Dans ce contexte, plusieurs approches se sont intéressées à la description formelle de la sémantique des interfaces et à la définition des techniques d'appariement des spécifications [ZW97]. Ces approches tentent à retrouver des composants dans une bibliothèque de composants, dont les spécifications coïncident avec la spécification d'une requête donnée. Autrement dit, les conditions d'appariement considérées doivent être vérifiées entre les composants sélectionnés et la spécification de la requête. Les conditions d'appariements sont définies pour les deux niveaux suivants.

- Le niveau syntaxique basé sur l'appariement des signatures.
- Le niveau sémantique basé sur un sous ensemble des conditions d'appariement de spécifications [ZW97].

Certains autres travaux récents abordent le problème de l'échec de la recherche en proposant la composition et l'adaptation des composants d'une bibliothèque pour obtenir un nouveau composant qui satisfait la recherche. Nous présentons brièvement deux approches d'adaptation introduites dans [MA03] et [Hem05]. Les auteurs proposent de satisfaire un problème par l'utilisation des composants existants d'une bibliothèque. Les niveaux syntaxique et sémantique sont considérés.
Les composants considérés disposent d'un ensemble de paramètres input et output et ils sont décrits par une précondition et une postcondition (niveau sémantique).
Nous pouvons considérer ces composants comme étant des composants élémentaires puisqu'ils ont une seule fonctionnalité. Ces approches sont caractérisées par le fait qu'elles utilisent la composition des composants existants pour réaliser l'adaptation. La façon de composer les composants est générée de manière semi-automatique à partir des mécanismes de recherche et d'appariement. Si nous voulons généraliser les résultats obtenus par ces travaux à des composants non élémentaires (satisfaisant plusieurs fonctionnalités), alors nous pouvons proposer deux applications possibles de ces règles de composition.

- Puisque les composants sont élémentaires, alors ils peuvent être considérés comme des opérations. Par conséquent, les règles de composition proposées peuvent être considérées comme des règles de correspondance entre les opérations des deux composants.

– Dans le cas où on n'arrive pas à faire correspondre certaines opérations ou bien on les fait correspondre partiellement, alors on peut utiliser la composition de nouveaux composants pour l'adaptation.

Approche proposée par Morel et Alexander (MA) [MA03]. Les auteurs présentent SPARTACAS qui est un environnement de développement (« framework ») pour automatiser la recherche de composants et leur adaptation.
Description des composants. La spécification formelle des composants est donnée par des structures axiomatiques de la forme :

$$\forall d \in D, \exists r \in R | I(d) \Rightarrow O(d, r)$$

avec D et R sont respectivement le domaine(l'ensemble des valeurs input) et le codomaine (ensemble des valeurs output), I est un prédicat qui définit les inputs légales du composant et O est un prédicat qui définit les outputs faisables pour chaque input.
Processus de recherche et d'adaptation. Le but de cet environnement de développement est la recherche des solutions possibles, dans une bibliothèque de composants, afin de résoudre un problème de recherche donné. La recherche se fait en deux étapes. Initialement, un filtre basé sur l'appariement des signatures des composants est effectué. Ce qui permet d'éliminer les composants qui n'ont pas de signatures compatibles. Ensuite, un deuxième filtre est effectué basé sur les conditions d'appariement des spécifications entre le problème considéré et les composants de la bibliothèque. Les conditions d'appariement des spécifications utilisées sont un sous-ensemble des conditions d'appariement des spécifications définies par Zaremski et Wing [ZW97] : « Plug-in Post », « Weak Post », « Plug-in Pre », « Weak Post ». L'étape de recherche peut retourner un appariement de la spécification total ou partiel.

Dans le cas d'un appariement partiel, une adaptation est nécessaire puisque la spécification du composant sélectionné satisfait en partie la spécification du problème à résoudre. L'environnement SPARTACAS fournit un ensemble de théories d'adaptation d'architecture. Ces théories spécifient ; (i) les contraintes dans la connexion des composants, (ii) les effets de l'adaptation sur le comportement du composant et (iii) la fonctionnalité de tout le système en utilisant les composants instanciés. La définition de ces théories d'adaptation dépend de la condition d'appariement retenue entre le problème spécifié et le composant à rechercher dans la bibliothèque. En se basant sur les informations données par le composant sélectionné, le problème et la spécification de l'architecture, un sous-problème est synthétisé spécifiant la fonctionnalité manquante. Une deuxième recherche est alors lancée pour trouver un autre composant qui satisfait le sous-problème et l'ajouter à l'architecture. De même pour le sous problème synthétisé, on peut échouer dans son appariement total alors il faut synthétiser un nouveau sous problème et ainsi de suite jusqu'à arriver à satisfaire complètement le problème par la composition d'un ensemble de composants.
Types d'adaptation. Trois théories d'adaptation sont étudiées :

– Adaptation d'architecture séquentielle. Cette adaptation consiste à connecter deux composants d'une manière séquentielle de telle façon que les valeurs de sortie du premier composant soient les valeurs d'entrée du second.
– Adaptation d'architecture alternative. Cette adaptation consiste à connecter deux composants qui s'exécutent simultanément et dont les valeurs de sortie sont combinées pour satisfaire le problème requis.
– Adaptation d'architecture parallèle. Cette adaptation consiste à connecter deux composants, de manière similaire à l'architecture alternative, sauf qu'entre ces deux composants il n'y a pas d'interaction .

Approche proposée par Hemer (H) [Hem05]. L'auteur propose d'utiliser des modèles (« template ») pour définir des stratégies d'adaptation, permettant de modifier et de combiner des composants existants dans une bibliothèque, afin d'obtenir un nouveau composant qui satisfait un problème de spécification donné.
Cette approche est réalisée dans le langage CARE qui est un langage, une méthodologie et un ensemble d'outils conçus pour générer un code vérifié à partir de spécification de haut niveau. Le langage est de nature fonctionnelle avec des programmes constitués d'un ensemble d'unités. Ces unités peuvent être des types qui modélisent la structure de données ou des fragments qui sont similaires aux fonctions utilisées dans les langages de programmation fonctionnelle. CARE supporte la notion de réutilisation en fournissant une bibliothèque de modules et de modèles. Un module correspond à une collection de fragments et types liés. Un modèle est une bibliothèque générique de composants, qui est constitué comme un module mais qui peut aussi être instancié par rapport au comportement fonctionnel aussi bien que par des types.
Dans cette approche, un ensemble de modèles sont définis, permettant d'adapter et de composer des composants CARE. Les modèles définis supportent la composition des fragments.
Types d'adaptation. Plus précisément, trois cas de composition sont supportés, appelés : architecture séquentielle, architecture indépendante et architecture alternée.

- les modèles d'emballage (« wrapper ») prennent en entrée un seul fragment et ils effectuent des modifications sur ce fragment afin d'en obtenir un nouveau. Deux exemples de modèles d'emballage sont présentés :
  - Le modèle « DropInput » permet de transformer un fragment en lui enlevant un paramètre.
  - Le modèle « WeakenPrecond » permet à un fragment d'être implémenté par un autre fragment avec une précondition plus faible.
- L'architecture séquentielle permet de résoudre un problème par composition séquentielle de deux fragments. Deux exemples de modèles associés à cette architecture sont donnés.
  - Le modèle « FunDecomp » permet à un problème d'être décomposé en deux sous problèmes qui peuvent être résolus séparément par deux fragments. Les outputs des fragments sont ensuite combinés pour former l'output désiré.
  - Le modèle « SeqDecomp » permet à un fragment d'être implémenté de manière séquentielle par deux fragments vérifiant certaines conditions.
- L'architecture indépendante permet de résoudre un problème en le devisant en sous problèmes qui sont par la suite résolus indépendamment. Un exemple de modèle, nommé le modèle « ParDecomp », est proposé. Ce modèle permet d'implanter un fragment en divisant le problème en deux cas. Ces cas sont implémentés séparément.
- L'architecture alternée permet de résoudre un problème par des cas d'analyse. Le modèle « CaseAnalysis » est proposé. Ce modèle permet d'implanter un fragment par un choix entre plusieurs fragments. Seulement les choix déterministes dépendants des inputs du fragment sont traités.

Processus de recherche et d'adaptation. Le développement de programme commence avec un problème de spécification (spécification d'un fragment) qui va être implémenté avec les fragments des modules de la bibliothèque. Ceci revient à chercher dans la bibliothèque un fragment telle que sa spécification s'apparente bien avec la spécification du problème spécifié. Trois approches de recherches formelles sont utilisées pour trouver des correspondances entre le problème de spécification et la bibliothèque des fragments.

- recherche basée sur l'appariement de signature faite en premier lieu.
- recherche basée sur l'appariement syntaxique qui s'applique quand l'un des fragments contient des paramètres de haut ordre et permet d'instancier ces paramètres. L'instan-

ciation des paramètres doit vérifier les contraintes spécifiées.
- recherche basée sur l'appariement des spécifications qui supporte une variété de conditions d'appariement présentés dans [ZW97].

Ces approches de recherches sont combinées en utilisant des tactiques. Dans le cas où le fragment considéré n'est pas apparié avec le problème de spécification, des modèles d'adaptation sont alors utilisés. Ces modèles leur permettent d'appliquer des transformations sur les fragments de la bibliothèque et de composer ces fragments jusqu'à résoudre le problème complètement.

3.3.4 Composition et adaptation des connecteurs

Les langages d'architecture (ADL) proposent la notion de connecteur pour assembler les composants. Un connecteur représente le protocole de communication entre différents composants. Il modélise les interactions et spécifie les règles qui doivent être respectées durant la communication des composants. Un connecteur peut être considéré comme étant un composant, mais avec des fonctionnalités particulières. Nous nous intéressons plus spécifiquement à ces connecteurs vis-à-vis du problème de l'adaptation. Lors de la conception d'un système les concepteurs peuvent utiliser des connecteurs existants prédéfinis, mais il n'est pas toujours possible de trouver un connecteur existant qui satisfait les besoins du système. Par conséquent, une adaptation des connecteurs existants candidats est nécessaire pour réussir un assemblage. Dans [SG01, SG03, Spi04], Spitznagel et Garlan (SG) présentent une approche d'adaptation de connecteurs permettant de produire de nouveaux connecteurs de façon systématique et compositionnelle. Le protocole des connecteurs est décrit par un langage de l'algèbre de processus appelé processus à état fini (FSP). La définition d'un nouveau connecteur adopte une approche par transformation de connecteurs existants. L'effet de ces transformations peut être décomposé en trois types d'adaptation : adaptation des interfaces liées par le connecteur, composition de nouvelles interfaces et transformation du protocole de communication.

Types d'adaptation. L'ensemble des transformations génériques sur les connecteurs sont :

- L'adaptation des interfaces peut être effectuée soit sur les données, soit sur le protocole.
  - Transformation des données. Elle permet de changer le format des données qui sont échangées lors d'une interaction. Ces changements peuvent avoir lieu soit à un des deux bouts de la communication, soit aux deux. Cette transformation n'affecte pas le protocole d'interaction.
  - Combinaison (« Splice »). Cette transformation permet de combiner des connecteurs binaires en un nouveau connecteur binaire. Le nouveau connecteur a une interface avec chacun des deux connecteurs initiaux. Cette transformation permet de modifier le protocole utilisé. Elle est utilisée principalement afin de permettre à deux composants avec des protocoles incompatibles de communiquer ensemble.
- La composition de nouvelles interfaces est définie par la transformation d'ajout d'un rôle. Cette transformation permet d'ajouter une nouvelle interface dans une interaction permettant à une autre partie d'être impliquée. Deux types d'interfaces ajoutées sont possibles observateur et participant. Un observateur écoute la communication entre les composants connectés et un participant peut prendre une partie active dans la communication.
- La transformation du protocole de communication est définie de deux manières :
  - Modification de protocole en mode session (« Sessionize »). Cette transformation permet de passer à un protocole orienté session. Le connecteur résultant maintient un état (l'identité, le temps, *etc*).
  - Agrégation de connecteurs. Cette transformation combine deux ou plusieurs connecteurs avec un contrôleur. À un moment donné, un seul connecteur est actif. Le contrôleur

détermine quel connecteur va être actif à un moment donné mais ne change pas le protocole des connecteurs. Par exemple, on veut construire un connecteur qui supporte plusieurs protocoles et il négocie à l'initialisation quel protocole va être utilisé entre les composants qu'il relie.

Ces transformations peuvent ensuite être combinées de plusieurs manières, afin d'obtenir un nouveau connecteur qui répond aux besoins.

3.3.5 Discussion

Un résumé des approches d'adaptation évoquées dans ce chapitre est donné par le Tableau 3.1. Pour chacun de ces travaux, nous rappelons les formalismes et les niveaux d'interopérabilité considérés pour la description des interfaces, les propriétés à vérifier ainsi que le degré d'automatisation de l'approche.

Pour certaines de ces approches, nous avons décelé les limites suivantes :

- Elles sont décrites par des algorithmes ou partiellement implantées.
- Elles s'appuient sur des notations formelles difficiles à utiliser pour effectuer les vérifications.
- Les adaptations traitent séparément les incompatibilités au niveau protocole et au niveau sémantique, alors que la prise en compte simultanée des deux niveaux permettrait un résultat de l'adaptation plus précis.
- Dès que le nombre de composants à adapter est grand, la vérification devient compliquée et difficile à gérer. Cependant, les approches d'adaptation évoquées pour l'assemblage de plusieurs composants ne proposent pas des mécanismes pour gérer cette complexité (*e.g.* décomposer la vérification pour la simplifier).

3.4 Caractéristiques d'une approche d'adaptation

Toutes les approches d'adaptation décrites en Section 3.3 ont pour objectif la définition et la construction des adaptations pour corriger des incompatibilités. L'objectif de la section actuelle est de mettre en évidence les principales caractéristiques d'une approche d'adaptation. Nous avons déduit ces caractéristiques à partir de l'étude de l'existant.

3.4.1 Niveaux d'interopérabilité

Les approches proposées dans la littérature diffèrent par leur description des interfaces. En particulier, plusieurs langages sont utilisés avec des degrés d'expressivité différents et les niveaux d'intéropérabilité considérés ne sont pas les mêmes. Tous les travaux cités ci-dessous donnent une description des interfaces en prenant en considération un ou plusieurs niveaux d'intéropérabilité, en particulier les niveaux syntaxique, sémantique ou protocole sont considérés. Nous regroupons les approches selon les niveaux considérés, comme suit.

- Niveaux syntaxique et protocole : Yellin *et al.* (des FSMs), Passerone *et al.* (des automates), Bracciali *et al.* (π-calcul), Canal *et al.* (des STEs), Dumas *et al.* (des traces), Benatallah *et al.* (un modèle de protocole [BCT04]) et Spitznage *et al.* (FSP).
- Niveau protocole : Schmidt *et al.* (des automates) et Inverardi *et al.* (des automates).
- Niveaux syntaxique et sémantique : Morel *et al.* (pré/post-conditions) et Hemer (pré/post-conditions).
- Niveaux syntaxique, sémantique et protocole : André *et al.* (composant Kmelia).

Approche	Description d'interface	Niveau	Vérification	Implantation
YS	FSM	syntaxique protocole	absence de blocage absence de réception non spécifiée respect des correspondances	algorithmes
RS	FSM	protocole	absence de réception non spécifiée	algorithmes
BBCP	π-calcul	syntaxique protocole	absence de blocage absence de réception non spécifiée respect des correspondances	algorithmes
AHPS	automate	syntaxique protocole	absence de blocage respect des interfaces respect des propriétés	algorithmes
IT	bMCS et HMCS CCS	protocole	absence de blocage respect des interfaces respect des propriétés	outil SYNTHESIS
CPS	LTS ou π-calcul	syntaxique protocole	absence de blocage respect des interfaces respect des propriétés	outil Adaptor
AAA	signature pré/post- conditions eLTS	syntaxique sémantique protocole	signature des services signature « profonde » Conditions d'appariement interaction des eLTS	outil COSTO
DSW	trace d'actions	protocole syntaxique	absence de blocage respect des interfaces	
BCT	automate	syntaxique protocole		
MBMCC	WSDL STE	syntaxique protocole	absence de blocage respect des interfaces respect des correspondances	outil
MA	pré/post- conditions	sémantique syntaxique	conditions d'appariement*	SPARTACAS
H	pré/post- conditions	sémantique syntaxique	conditions d'appariement*	CARE
SG	FSP Process algebra	syntaxique protocole		outil

TABLE 3.1 – comparaison des approches d'adaptation

* : les conditions définies par la Figure 3.2

Selon le niveau étudié dans ces approches et selon le langage utilisé pour la description des interfaces, la définition de l'interopérabilité est différente et par conséquent l'adaptation est différente.

La plupart des approches citées, qui considèrent le niveau syntaxique, prennent en considération la différence des noms et des paramètres des opérations des interfaces. Les approches qui décrivent le niveau protocole définissent l'interopérabilité par le respect des protocoles des interfaces mis en jeux et par la vérification de l'absence de blocage dans les communications. Les approches décrivant le niveau sémantique définissent l'interopérabilité par la vérification des conditions d'appariement des spécifications [ZW97]. Ces travaux se sont également intéressés à deux principaux problèmes d'interopérabilité : le problème de connexion entre interfaces et le remplacement d'un composant par un autre (appelé aussi recherche de composant ou substitution).

3.4.2 Spécification

Les approches citées peuvent être décomposées en deux catégories :

- L'adaptation consiste à construire un composant particulier entre les composants existants. Ce composant est appelé adaptateur dans [YS97, SR02, BBC05, BCP06, CPS06, CPS08, BCG+05], convertisseur de protocole dans [PdAHSV02], connecteur dans [IT03, TA06] et médiateur ou adaptateur de services dans [DSW06].
- L'adaptation consiste à construire un nouveau composant [Hem05, MA03] en composant et modifiant des composants existants. L'adaptation consiste donc à construire une enveloppe permettant d'encapsuler un ou plusieurs composants afin de construire un nouveau composant. D'une manière analogique construire un nouveau connecteur [SG01, SG03, Spi04] en composant et modifiant les connecteurs existants.

Pour ces deux catégories, la spécification de l'adaptation peut être constituée par :

- Des règles de correspondance ou de coordination.
- Des règles de composition.
- Des propriétés.
- Des données.

Ces éléments n'existent pas nécessairement tous dans les approches d'adaptations.

Règles de correspondance ou de coordination

Un adaptateur joue le rôle d'un pont. Sa principale fonction est de lier les interfaces à adapter. Par conséquent, la définition des règles de correspondance ou de coordination est une étape fondamentale dans l'adaptation. En général, ces règles sont nommées des règles de correspondance si deux interfaces sont considérées et des règles de coordination si plusieurs interfaces sont considérées. On distingue deux types de règles de correspondance ou de coordination : des règles pour lier les *données* des interfaces et d'autres pour lier les *opérations* des interfaces.

Données. Dans le cas où les interfaces contiennent des données dont on veut tenir compte de l'incompatibilité syntaxique, la définition des règles de correspondance entre les données est nécessaire. Ces correspondances dépendent des types des données supportées par les interfaces : des données simples, des données structurées et même des modèles de données complexes (*e.g.*, ensemble de classes avec des associations).

Opérations. La génération de l'adaptation dépend d'un ensemble de règles de correspondance ou de coordination entre les interfaces. La plupart des approches supposent que ces règles de correspondance soient complètes. Par exemple, si on a des interfaces requises et des interfaces fournies alors toutes les opérations des interfaces requises doivent au moins apparaître une fois dans une règle de correspondance.

Plusieurs formalismes ont été proposés avec des niveaux différents d'expressivité. Il existe principalement trois types de formalismes :

- Des contraintes de synchronisation. Les vecteurs de synchronisation introduits par Arnold [Arn94]. Une extension est donnée dans [CPR00] dans lequel des vecteurs de synchronisation avancés sont définis en utilisant des formules de logique temporelle. Dans [SP05], des vecteurs de synchronisation sont utilisés avec des diagrammes d'interaction (MSC ou les diagrammes de séquence d'UML) afin d'exprimer l'interaction entre des diagrammes d'états étendus.
  Dans certaines approches d'adaptation, les correspondances ou les coordinations sont implicites, elles concernent les messages de même nom qui sont reçus par une interface et envoyés par une autre interface (synchronisation par émission réception) dans le cas d'une connexion et de même signe dans le cas d'un remplacement.
- Des langages basés sur des règles de correspondance.
- Des automates étiquetés par le produit cartésien des messages des interfaces à adapter.

Le point fort des deux premiers formalismes provient de leur définition qui est plus intuitive que les automates pour le concepteur, alors que l'utilisation des automates permet de décrire des propriétés sur la communication, ce pouvoir d'expression est donc le point fort du troisième formalisme.

Propriétés

L'adaptation au niveau protocole peut servir à résoudre les incompatibilités, mais en plus elle peut permettre d'imposer un ensemble de propriétés. Plusieurs formalismes ont été proposés avec des niveaux différents d'expressivité. Par exemple, nous pouvons citer : les automates, la logique temporelle, *etc.* Au niveau sémantique, l'adaptation peut également permettre d'imposer certaines propriétés sur les données manipulées par les interfaces (*e.g.* invariant).

En particulier, nous avons rencontré les formalismes ci-dessous utilisés pour décrire les correspondances ou les coordinations ainsi que pour définir des propriétés sur l'adaptation.

- Des formalismes permettant de définir des règles de correspondance.
  - Un langage de haut niveau pour présenter les correspondances entre les opérations des interfaces [YS97, BBC05, BCP06].
  - Un ensemble d'expressions de transformation d'interfaces [DSW06].
  - Des patterns [BCG+05] spécifiant les correspondances en fonction des erreurs rencontrées.
- Des formalismes décrivant des règles de correspondances et des propriétés.
  - Dans [PdAHSV02], un ensemble de propriétés décrites par des automates, dont l'alphabet est dérivé à partir du produit cartésien des alphabets des deux protocoles à connecter. Ces automates permettent de spécifier des correspondances entre les interfaces et aussi de définir des contraintes sur la communication des deux composants. Des formules en logique temporelle ont été utilisées pour décrire les propriétés.
  - Un LTS de vecteurs [CPS06, CPS08] permet de décrire les règles de correspondance et certaines propriétés.
  - Un ensemble de propriétés décrites par les automates et par la logique temporelle [IT03, TA06]. Dans cette approche, les règles de correspondance ne sont pas données explicitement puisque la correspondance est faite directement par les message reçus et envoyés de même nom.

Règles de composition

La définition des règles de composition peut être nécessaire principalement à cause des incompatibilités au niveau sémantique. Plus particulièrement, si l'ensemble des opérations d'une interface sont partiellement réalisées alors une adaptation possible est de chercher un ou plusieurs composants et de les composer afin d'arriver à la réalisation complète. Ainsi, plusieurs formalismes sont définis pour décrire ces compositions. Nous distinguons deux manières d'effectuer les compositions : soit à travers l'adaptateur soit directement en connectant des composants sélectionnés. Le premier cas nous permet d'avoir des compositions plus élaborées mais il exige que toutes les communications passent par l'adaptateur. Dans le deuxième cas, nous distinguons essentiellement trois types de compositions : séquentielles, parallèles et alternées.

Ainsi, plusieurs formalismes ont été introduits :

- Des patterns proposés dans [MA03] permettent de modifier et de composer les composants de la bibliothèque.
- Des théories d'adaptation définies dans [Hem05] permettent de composer les composants de la bibliothèque.
- Des transformations introduites dans [SG01, SG03, Spi04] permettent de modifier et de composer des connecteurs existants.

Données

La plupart des approches d'adaptation évoquées supportent que la spécification de l'adaptateur puisse contenir des données. Ces données permettent de sauvegarder les paramètres de certains messages reçus et des informations sur l'historique des anciennes communications. Les données sauvegardées permettent par la suite de synthétiser certains messages et les paramètres de certains messages. Ces données sont utilisées pour supporter des techniques d'adaptation telles que l'ordonnancement des messages dans une communication et la synthèse des paramètres des messages.

3.4.3 Spécification semi-automatique vs. manuelle

La spécification de l'adaptation peut être donnée par le concepteur comme elle peut être définie en partie de manière semi-automatique. Par exemple, les règles de correspondance et de composition peuvent être définies en se basant sur des techniques de recherche et d'appariement syntaxique ou sémantique.

Dans le cas où la spécification est fournie, le concepteur doit avoir une connaissance approfondie des composants et des problèmes d'incompatibilité. En outre, le risque d'erreur est important puisque la spécification est donnée manuellement par le concepteur.

Dans le second cas, l'avantage est que la spécification de l'adaptateur est effectuée de manière semi-automatique, le risque d'erreur est donc minime. Cependant, l'inconvénient de l'utilisation des techniques de recherche et d'appariement est leur complexité ainsi que le grand nombre de solutions qui peuvent être générées par ces techniques (problème d'indéterminisme). Il est à noter que plus des détails sont considérés dans la description des interfaces, plus des contraintes seront imposées dans le processus de recherche et d'appariement, et donc plus le nombre des solutions proposées va être réduit. Si nous prenons l'exemple des incompatibilités au niveau protocole, un grand nombre de solutions de correspondance est possible (un à un, un à plusieurs, plusieurs à un, *etc*). Au niveau sémantique, nous remarquons également la difficulté d'établir des correspondances pour les opérations sans effet qu'on appelle des opérations de requêtes.

3.4.4 Types d'adaptation

Nous rappelons et classifions les différentes règles de correspondances définies explicitement dans les approches évoquées dans ce chapitre :

1. Correspondances entre deux opérations qui sont différentes syntaxiquement. Ce type de correspondance est considéré dans [BCP06] par la correspondance un-à-un, dans [DSW06] par l'opérateur *Flow* et dans [BCG+05] par le pattern SMP. Cette correspondance peut être décomposée par :
   - 1.1 Les noms des opérations sont différents.
   - 1.2 Une opération a un paramètre en plus, cette correspondance est considérée par le modèle « DropInput » dans [Hem05].
   - 1.3 Une opération a un paramètre manquant.
   - 1.4 Les paramètres des deux opérations ne sont pas dans le même ordre.
2. Correspondances entre deux opérations telle que l'opération fournie, impose des contraintes sur les paramètres d'entrée qui sont moins restrictives que les paramètres dans l'opération requise. Ce type de correspondance est considéré dans [BCG+05] par le pattern PCP et dans [Hem05] par le modèle « WeakenPrecond ».
3. Correspondances entre deux ensembles d'opération. Ce type de correspondance est considéré dans [BCP06] par la correspondance multiple [BCP06]. Cette correspondance peut être encore décomposée comme suit.
   - 3.1 Correspondances entre deux ensembles d'opération avec un ordre différent. Ce type d'adaptation est considéré par le pattern OCP (« Ordering Constraint Pattern ») dans [BCG+05].
   - 3.2 Correspondance « paramètre vs. message » [AAA07]. Dans certains cas, une opération contient des paramètres mais l'opération correspondante dans un autre composant utilise des messages pour envoyer les valeurs des paramètres.
   - 3.3 Correspondance une opération et plusieurs opérations. Ce type de correspondance est considéré par le pattern OMP dans [BCG+05] et par l'opérateur « Scatter » dans [DSW06]. Il peut être décomposé comme suit.
     - 3.3.1 On fait correspondre à une opération plusieurs opérations qui sont exécutées de manière séquentielle. Ce type de correspondance est considéré par l'adaptation d'architecture séquentielle dans [MA03] et par les modèle « FunDecomp » et « SeqDecomp » dans [Hem05].
     - 3.3.2 On fait correspondre à une opération plusieurs opérations qui sont exécutées de manière parallèle. Ce type de correspondance est considéré par l'adaptation d'architecture parallèle dans [MA03] et par le modèle « ParDecomp » dans [Hem05].
     - 3.3.3 On fait correspondre à une opération plusieurs instance d'une même opération. Cette correspondance est considérée par l'opérateur « Burst » dans [DSW06].
   - 3.4 Correspondance de plusieurs opérations à une seule opération. Ce type d'adaptation est considéré par le pattern MOP dans [BCG+05] et par l'opérateur « Gather » dans [DSW06]. Dans [DSW06], une variante de cette adaptation est présentée par l'opérateur « Collapse » dont l'ensemble des opérations sont les mêmes.
4. Ce type de correspondance est considéré par les actions sans correspondances dans [BCP06] et par l'opérateur « Hide » dans [DSW06]. Cette correspondance peut être décomposée comme suit.

4.1 Un message attendu par une interface est non envoyé par l'autre interface. Cette correspondance est considérée par le pattern MMP dans [BCG+05].

4.2 Un message envoyé par une interface est non reçu par une autre interface. Cette correspondance est considérée par le pattern EDP [BCG+05].

5. Correspondance d'une opération à un choix d'opérations. Ces choix peuvent être déterministes comme non déterministes. Ce type d'adaptation est considéré par les correspondances d'actions indéterministes dans [BCP06], par l'adaptation d'architecture alternative dans [MA03] et par le modèle « CaseAnalysis » dans [Hem05].

3.4.5 Processus de construction de l'adaptateur

Le processus de construction de l'adaptateur peut se terminer avec succès comme il peut se terminer par un échec. Nous définissons les interfaces comme des interfaces adaptables ou des interfaces non adaptables respectivement selon le succès ou l'échec de la construction de l'adaptateur.

Dans les approches d'adaptation citées dans ce chapitre, le processus de construction de l'adaptateur est basé sur une spécification de l'adaptation (*e.g.* propriétés, règles de correspondance, *etc.*) et des interfaces considérées. Il peut être défini de différentes manières. Nous proposons la classification suivante :

- Approches d'adaptations restrictives. Une première version de l'adaptateur est construite à partir des interfaces. Cette version contient tous les comportements possibles entre les interfaces considérées sans aucune synchronisation. L'étape suivante consiste à vérifier si dans la première version générée, la spécification abstraite de l'adaptateur est bien respectée. Si la vérification échoue, alors il faut restreindre le comportement de l'adaptateur en enlevant tous les états et les transitions qui violent les propriétés spécifiées. En d'autres termes, l'adaptateur force les interfaces à se comporter en respectant le comportement désiré, *i.e.* sa description fournie.
- Approches d'adaptations génératives. Ces approches proposent la construction de l'adaptateur en tenant compte de la spécification fournie de l'adaptateur et des interfaces à adapter. La vérification des propriétés de l'adaptateur est effectuée par construction. Ce processus échoue s'il n'arrive pas à trouver une adaptation qui vérifie la spécification fournie et les deux interfaces.
- Approches d'adaptations génératives restrictives. La première étape consiste à construire une première version de l'adaptateur en considérant sa spécification fournie et les interfaces à adapter. La deuxième étape est la vérification que la première version respecte bien d'autres propriétés non prises en compte dans sa construction. Il existe donc deux types de propriétés : des propriétés qui sont vérifiées par construction de l'adaptateur et des propriétés à vérifier après construction .
- Approches d'adaptations génératives correctives. Construire une première version de l'adaptateur à partir de la spécification abstraite de l'adaptateur et des interfaces à adapter. En se basant sur l'échec de la vérification de cette version, *i.e.* les incompatibilités rencontrées, des corrections sont proposées et elles mettent en cause la spécification abstraite de l'adaptateur.

Cette classification est inspirée de la classification proposée dans [CMP06] avec certaines modifications.

Les travaux cités peuvent être classés, selon le processus de construction choisi, comme suit :

- Les approches génératives [YS97, BCP06, DSW06, BCG+05, MA03, Hem05, Spi04].
- Les approches correctives [SR02] (« Adaptateur de changement de protocole »).

- Les approches d'adaptations restrictives [IT03], [PdAHSV02] et [SR02] (« n:1-Adaptateur »).
- Les approches génératives correctives [CPS06, CPS08].

3.5 Conclusion

Dans ce chapitre, nous avons présenté quelques travaux faisant partie de l'état de l'art concernant l'adaptation logicielle statique. L'étude de ces travaux nous a permis de mettre en évidence certains éléments de la problématique de l'adaptation statique.

Nous concluons par une énumération de questions et par les réponses correspondantes :

- L'adaptation est schématiquement un morceau de code qui permet de lier des interfaces : « Sous quelle forme ce morceau de code doit-il être introduit ? » Ce morceau de code correspond à un composant spécial, un connecteur ou une enveloppe à un ou plusieurs composants.
- « Comment décrire une adaptation de manière abstraite afin de pouvoir guider la construction de l'adaptateur ? » La description abstraite peut être caractérisée par un ensemble de correspondances, un ensemble de compositions de composants, des propriétés et des données propres à l'adaptateur. Chacun de ces éléments peut être décrit avec des formalismes différents selon les approches et ils ne sont pas nécessairement tous présents dans une approche d'adaptation.
- « Comment l'adaptation est-elle construite ? » On peut distinguer les approches d'adaptation génératives, correctives et restrictives.
- « Comment vérifier la validité de l'adaptateur ? » Dans la plupart des approches d'adaptation proposées, cette étape est incluse dans l'étape de construction, car les algorithmes proposés permettent de construire l'adaptation à partir de la spécification abstraite et des interfaces considérées, les algorithmes garantissent la validité de l'adaptateur lorsqu'ils s'exécutent avec succès.

Dans les chapitres 5 et 6, nous décrivons notre approche de l'adaptation pour deux interfaces requise et fournie et pour plusieurs composants.

4

Raffinement des systèmes de transitions étiquetés

Sommaire

4.1 Introduction 59
4.2 Définitions 60
4.2.1 Système de transitions étiqueté (STE) 60
4.2.2 Système de transitions étiqueté gardé 61
4.2.3 Propriétés des systèmes de transitions étiquetés 62
4.2.4 Composition d'étiquettes 63
4.3 Traduction en B 65
4.3.1 Système de transitions étiqueté 65
4.3.2 Relation entre les états de deux STEs 66
4.3.3 Composition d'étiquettes 67
4.4 Relations entre STEs et raffinement B 70
4.4.1 Relation entre deux STEs sur un même alphabet 70
4.4.2 Relation entre deux STEs avec des étiquettes différentes 76
4.4.3 Relation entre un STE et plusieurs STEs 83
4.4.4 Raffinement entre deux STEs gardés 84
4.5 Travaux connexes 85
4.5.1 Les travaux de [BJK00] 85
4.5.2 L'approche *GeneSyst* 85
4.5.3 *ProB* 86
4.6 Conclusion 86

4.1 Introduction

Le raffinement est une notion clé dans la méthode B. Nous étudions le rapport entre cette notion et certaines relations entre systèmes de transitions étiquetés comme la simulation, la bisimulation, *etc.* Le point de départ de l'étude est la définition d'une traduction particulière des STEs en spécification B. Nous justifions cette traduction par le fait que le STE associé à la sémantique opérationnelle de la spécification B est équivalent au STE initial. Nous nous intéressons aux différentes relations que nous pouvons modéliser entre les spécifications B des

STEs. Nous présentons alors des schémas (ou des constructions) composés par des spécifications B des STEs, basés sur les clauses de modularité **INCLUDES** et **REFINES**. Ces schémas nous permettent de modéliser différentes relations entre STEs, pour chacun de ces schémas nous détaillons une proposition en terme de relation entre STEs, en nous basant sur les obligations de preuve générées. Nous proposons aussi d'ajouter des conditions sur les spécifications permettant ainsi de modéliser certaines relations entre systèmes de transitions étiquetés telle que la ready-bisimulation.

Les schémas mis en évidence sont, ensuite, utilisés dans d'autres chapitres dans le cadre de l'assemblage des composants.

La suite de ce chapitre est organisée comme suit. Nous commençons par introduire des définitions préliminaires dans la section 4.2. Dans la section 4.4 nous mettons en évidence des relations entre STEs. Pour établir ces relations, nous utilisons des constructions B qui utilisent la traduction des STEs en B définie dans la section 4.3. Nous terminons par un survol de quelques travaux qui se sont intéressés à définir la sémantique du raffinement B dans la section 4.5.

4.2 Définitions

Nous introduisons dans ce paragraphe des définitions préliminaires relatives aux systèmes de transitions étiquetés et à certaines relations entre les systèmes de transitions (simulation, bisimulation, ready-simulation). Nous présentons, également, une nouvelle notion appelée composition d'étiquettes qui permet de mettre des STEs en relation et de modéliser le raffinement B.

4.2.1 Système de transitions étiqueté (STE)

Définition 4.2.1 (Système de transitions étiqueté (STE))**.**
Un STE est un quintuplet $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$ où :

- A est un ensemble d'étiquettes
- S est un ensemble d'états
- $s_0 \in S$ désigne l'état initial de T
- $F \subseteq S$ est le sous-ensemble des états finals
- $\rightarrow \subseteq S \times A \times S$ est la relation de transition du système T

On dénote $s \xrightarrow{e} s'$ la transition $(s,\ e,\ s')$ appartenant à $\rightarrow$.
Dans la suite lorsque nous considérons un STE T, nous supposons que $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$.

Définition 4.2.2 (Étiquette et STE)**.**
Soient T un STE et e une étiquette.

- $pre(e,\ T) = \{s\ ,\ s \in S\ \wedge\ \exists s' \in S\ .\ (s,\ e,\ s') \in \rightarrow\}$ désigne l'ensemble des états à partir desquels l'étiquette e peut être déclenchée, on le note $pre(e)$ s'il n'y a pas d'ambiguïté sur le STE considéré.
- $post(e,\ T) = \{s'\ ,\ s' \in S\ \wedge\ \exists s \in S\ .\ (s,\ e,\ s') \in \rightarrow\}$ désigne l'ensemble des états obtenus après déclenchement de l'étiquette e, on le note $post(e)$ s'il n'y a pas d'ambiguïté sur le STE considéré.
- $s \xrightarrow{e}$ signifie qu'il existe un élément s' de S tel que $s \xrightarrow{e} s'$, c'est-à-dire que $s \in pre(e)$.
- $s \not\xrightarrow{e}$ signifie qu'il n'existe pas un état s' de S tel que $s \xrightarrow{e} s'$, c'est-à-dire que $s \notin pre(e)$.
- $\xrightarrow{e} = \{(s, s')\ ,\ s \in S\ \wedge\ s' \in S\ \wedge\ (s,\ e,\ s') \in \rightarrow\}$

Définition 4.2.3 (Chemins et traces)**.**

- Un chemin c dans T est une suite finie de transitions de la forme $q_i \xrightarrow{e_i} q_{i+1}$ où $0 \leq i \leq n$ telle que $q_0 = s_0$ et $q_{n+1} \in F$. $q_0 = origine(c)$ est l'origine du chemin c. $q_{n+1} = extremite(c)$ est l'extrémité du chemin c. L'ensemble des chemins dans T est noté $Chemin(T)$.
- La suite d'étiquettes $(e_i)_{0 \leq i \leq n}$ associée au chemin précédent est appelée la trace du chemin. L'ensemble des traces finies associées à T est noté $Trace(T)$.
- Nous étendons la notion de chemin à des origines et des extrémités quelconques. Une suite $q_i \xrightarrow{e_i} q_{i+1}$ où $1 \leq i \leq n$ est un chemin initialisé dans l'état q_0 et aboutissant à l'état q_{n+1}. On note $Chemin(T,\ q_0,\ q_{n+1})$ l'ensemble de ces chemins.
  Soient E et E' deux sous-ensembles de $S, Chemin(T,\ E,\ E') = \bigcup_{s \in E, s' \in E'} Chemin(T,\ s,\ s')$.
  Si $\mathcal{C}$ est un ensemble de chemins, on note $origine(\mathcal{C}) = \{origine(c),\ c \in \mathcal{C}\}$, l'ensemble constitué des origines des chemins de $\mathcal{C}$.

Nous définissons, maintenant, le produit libre de plusieurs STEs.

Définition 4.2.4 (Produit libre [Arn92])**.** Soient $T_i = (A_i, S_i, s_0^i, F_i, \rightarrow_i)$, $1 \leq i \leq n$ des STEs. Le STE associé à ce système est $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$ avec :

1. $A = \bigcup_{i=1}^{n} A_i$
2. $S = S_1 \times \ldots \times S_n$
3. $s_0 = (s_0^1, \ldots, s_0^n)$
4. $F = F_1 \times \ldots \times F_n$
5. $\rightarrow$ désigne l'ensemble des transitions de la forme $((s_1, \ldots,\ s_n), e, (s'_1, \ldots,\ s'_n))$ et telles que $\exists\ k \in [1\ ..\ n]$ vérifiant les conditions suivantes :
   (a) $(s_k, e,\ s'_k) \in \rightarrow_k$
   (b) $(\forall i \in [1\ ..\ n] \setminus \{k\})\ s_i = s'_i$

4.2.2 Système de transitions étiqueté gardé

Nous étendons la définition des STEs en tenant compte des gardes. Ainsi, on les appelle systèmes de transitions étiquetés gardés.

Définition 4.2.5 (Système de transitions étiqueté gardé)**.**
Un STE gardé est un septuplet, $T = (A,\ G(x), D,\ S,\ s_0,\ F,\ \rightarrow)$ où :

- A, S, s_0 et F sont définis de la même manière que dans un STE.
- G est un ensemble de prédicats. Ces prédicats sont définis en termes des variables x (dépendant du contexte) ou par vrai.
- D est le domaine des variables x.
- $\rightarrow \subseteq S \times G(x) \times A \times S$ est la relation de transition du système T.

Définition 4.2.6 (Transition valide)**.**
La transition $(s,\ [g]\ e,\ s')$ peut être activée à partir de l'état s dans un contexte où sa garde g est vraie. Le déclenchement de l'événement e génère l'état s'. Dans ce cas, la transition est valide.

Définition 4.2.7 (Étiquette et STE)**.**
Soient T un STE gardé, e une étiquette, q un état de S et Q un ensemble d'états de S :

- $q \overset{[g(x)]e}{\rightarrow} q' \overset{def}{\equiv} (q,\ [g]\ e,\ q') \in \rightarrow\ \wedge\ g(x)$. Ce prédicat est vrai si la transition $(q,\ [g]\ e,\ q')$ est valide et inversement.
- $q \overset{[?]e}{\rightarrow} \overset{def}{\equiv} (\exists q' \in S)\ (\exists g\ \in G)\ [(q,\ [g]\ e,\ q') \in \rightarrow]$
- $q \overset{[?]e}{\rightarrow} q' \overset{def}{\equiv} (\exists g\ \in G)[(q,\ [g]\ e,\ q') \in \rightarrow]$
- $gr(e,q,T) \overset{def}{\equiv} \bigvee_{g\ \in G'}(g)\ \wedge\ (\forall g\ \in G')\ (\exists q'\ \in S)[(q,\ [g]\ e,\ q') \in \rightarrow]$, c'est la disjonction de toutes les gardes dans des transitions issues de q et d'étiquette e.
- $gr(e,q,Q,T) \overset{def}{\equiv} \bigvee_{g\ \in G''}(g)\ \wedge\ (\forall g\ \in G'')\ (\exists q'\ \in S)[(q,\ [g]\ e,\ q') \in \rightarrow\ \wedge\ q' \in Q]$
- $pre(e,\ T) = \{(s\ ,x);\ s \in S\ \wedge\ x \in D\ \wedge\ \exists s' \in S\ .\ p \overset{[?]e}{\rightarrow_1}\ \wedge\ gr(e,p,\rightarrow_1)(x)\}$ désigne l'ensemble des couples formés des états et des valeurs de x à partir desquels l'étiquette e peut être déclenchée, on le note $pre(e)$ s'il n'y a pas d'ambiguïté sur le STE considéré.

4.2.3 Propriétés des systèmes de transitions étiquetés

Déterminisme

Définition 4.2.8 (STE déterministe)**.** T est déterministe s'il vérifie la propriété suivante :

$$(\forall e \in A)\ (\forall q \in S)\ (\forall q' \in S)\ (\forall q'' \in S)\ [q \overset{e}{\rightarrow} q'\ \wedge\ q \overset{e}{\rightarrow} q'' \Rightarrow q' = q'']$$

Relations entre systèmes de transitions

Définition 4.2.9 (Relation de simulation entre deux STEs)**.** Soient $T_1 = (A,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ et $T_2 = (A,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ deux STEs et R une relation de S_1 vers S_2. T_2 simule T_1 par rapport à R (noté $T_1 \preccurlyeq_R T_2$) ssi :

- $(s_0^1, s_0^2) \in R$.
- $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall p' \in S_1)$
  $[(p,q) \in R\ \wedge\ p \overset{e}{\rightarrow_1} p' \Rightarrow\ (\exists q' \in S_2)\ [(p',q') \in R\ \wedge\ q \overset{e}{\rightarrow_2} q']]$

Définition 4.2.10 (Relation de bisimulation entre deux STEs)**.** Soient $T_1 = (A,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ et $T_2 = (A,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ deux STEs et R une relation de S_1 vers S_2. T_2 et T_1 sont bisimilaires par rapport à R (noté $T_1 \approx_R T_2$) ssi :

- $T_1 \preccurlyeq_R T_2$
- $T_2 \preccurlyeq_{R^{-1}} T_1$

La bisimilarité implique l'équivalence de traces. Si les STEs T_1 et T_2 sont déterministes, l'équivalence de trace implique la bisimilarité.

La relation de ready-simulation est introduite par Bloom *et al.* [BIM95].

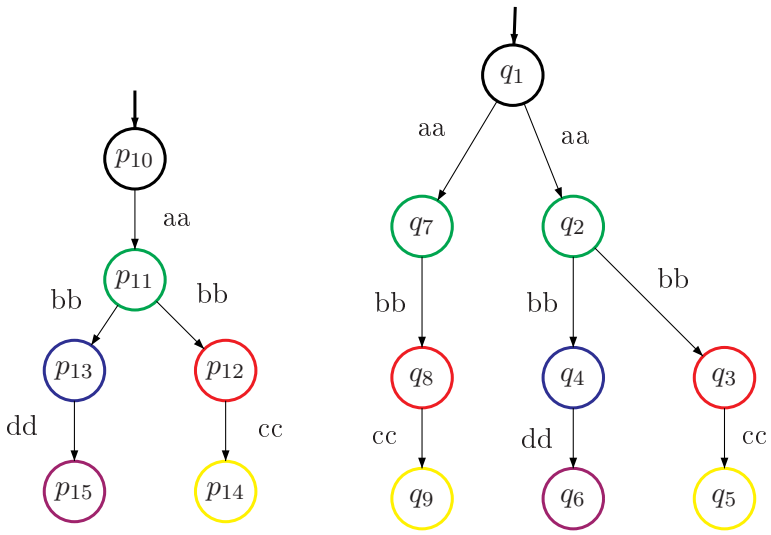
Définition 4.2.11 (Relation de ready-simulation [BIM95])**.** Soient $T_1 = (A,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ et $T_2 = (A,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ deux STEs et R une relation de S_1 vers S_2. T_2 et T_1 sont ready-similaires par rapport à R (noté $T_1 \precsim_R T_2$) ssi :

- $T_1 \preccurlyeq_R T_2$
- $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ \ [(p,q) \in R\ \wedge\ (\exists q' \in S_2)\ [q \overset{e}{\rightarrow_2} q'] \Rightarrow (\exists p' \in S_1)\ [p \overset{e}{\rightarrow_1} p']]$

Exemple 4.2.1. On considère les deux STEs T_1 et T_2 de la Figure 4.1 et on calcule la relation suivante entre leurs états :
$R^{-1} = \{(q_1,\ p_{10}), (q_2,\ p_{11}), (q_7,\ p_{11}), (q_3,\ p_{12}), (q_8,\ p_{12}), (q_4,\ p_{13}), (q_5,\ p_{14}), (q_9,\ p_{14}), (q_6,\ p_{15})\}$.
On a $T_2 \preccurlyeq_{R^{-1}} T_1$. La relation R^{-1} peut être calculée en partant du couple $(q_1,\ p_{10})$ formant les états initiaux et en générant la suite des couples :

$\Rightarrow (q_2,\ p_{11}) \in R^{-1}\ \wedge\ (q_7,\ p_{11}) \in R^{-1}$
$\Rightarrow (q_3,\ p_{12}) \in R^{-1}\ \wedge\ (q_8,\ p_{12}) \in R^{-1}\ \wedge\ (q_4,\ p_{13}) \in R^{-1}$
$\Rightarrow (q_5,\ p_{14}) \in R^{-1}\ \wedge\ (q_9,\ p_{14}) \in R^{-1}\ \wedge\ (q_6,\ p_{15}) \in R^{-1}$

On a également le deuxième item de la définition 4.2.11 est vérifié par R^{-1}. Les deux STEs T_1 et T_2 de la Figure 4.1 sont alors ready-similaires et on note $T_2 \precsim_{R^{-1}} T_1$.



(a) T_1 (b) T_2

FIGURE 4.1 – Exemple de deux STEs

4.2.4 Composition d'étiquettes

Nous introduisons une nouvelle notion appelée composition d'étiquettes. À chaque composition, nous associons un ensemble de chemins, cela nous permet d'établir des relations entre des STEs définis sur des alphabets différents. La traduction systématique des compositions d'étiquettes dans le langage B nous permet de donner une caractérisation du raffinement pour des STEs définis sur des alphabets différents.

Définition 4.2.12 (Composition d'étiquettes)**.** Soient $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$ un STE, P l'ensemble de prédicats unaires de profil $S \rightarrow Bool$ et $\{seq, alt, opt, loop\}$ un ensemble de constructeurs. L'ensemble SUB (noté aussi $\mathcal{E}(A)$) des compositions d'étiquettes associées à T est défini inductivement de la façon suivante :

- $\mathcal{B} = \{\epsilon\} \cup A$ est l'ensemble des compositions d'étiquettes de base
- Les règles de formation des compositions d'étiquettes sont :
  - $(\forall sub_1 \in\ SUB)\ (\forall sub_2 \in\ SUB)\ sub_1\ seq\ sub_2 \in\ SUB$
  - $(\forall sub_1 \in\ SUB)\ (\forall sub_2 \in\ SUB)\ sub_1\ alt\ sub_2 \in\ SUB$
  - $(\forall sub \in\ SUB)\ (\forall p \in P)\ opt\ p\ sub \in\ SUB$
  - $(\forall sub \in\ SUB)\ (\forall p \in P)\ (\forall n \in \mathbb{N})\ loop\ p\ n\ sub \in\ SUB$

Proposition 4.2.1 (Forme canonique). *Toute composition d'étiquettes peut se mettre sous la forme : sub_1 alt...alt sub_m où chaque sub_i est de la forme : $opt(g_1)e_1$ seq... seq $opt(g_n)e_n$. Cette forme de composition d'étiquettes est appelée forme canonique.*

Démonstration. La preuve de cette formule découle directement de la traduction des compositions d'étiquettes en substitutions généralisées (Tab. 4.1), de la proposition 4.3.1 et les règles de transformation des substitutions B. □

Définition 4.2.13 (Précondition d'une composition d'étiquettes). On considère des compositions d'étiquettes associées à un STE T, la précondition d'une composition d'étiquettes sub est notée par $pre(sub, T)$ ou $pre(sub)$, c'est un ensemble d'états de T défini comme suit.

- si $sub = opt(g_1)\epsilon$ alors $pre(sub) = S$
- si $sub = opt(g_1)e_1$ *seq...* *seq* $opt(g_n)e_n$ alors
  $pre(sub) = \{q_1;\ q_1 \in S \wedge \bigwedge_{j=1}^{n}((\forall q_2 \in S), ..., (\forall q_n \in S) [q_1 \xrightarrow{e_1} q_2 \xrightarrow{e_2} ... \xrightarrow{e_{j-2}} q_{j-1} \xrightarrow{e_{j-1}} q_j \ \wedge\ g_1(q_1)\ \wedge ... \wedge\ g_j(q_j) \Rightarrow q_j \xrightarrow{e_j}])\}$
- Pour toute composition d'étiquettes sub de forme canonique sub_1 *alt...alt* sub_m, $pre(sub) = \bigcap_{i=1}^{m} pre(sub_i)$.

La notion de chemin associé à une composition d'étiquettes se définit à partir de la notion de chemin valide.

Définition 4.2.14 (Chemins associés à une composition d'étiquettes).
Soient $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$ un STE.

1. Chemins valides.
   - Un chemin dans T de la forme $s_i \xrightarrow{e_i} s_{i+1}$ où $1 \leq i \leq n$ est valide par rapport à une composition d'étiquettes sub de la forme ($opt\ g_1\ e_1$) *seq...seq* ($opt\ g_n\ e_n$) si et seulement si $\forall i \in [1, n]\ g_i(s_i)$. L'ensemble des chemins dans T valides par rapport à sub est noté $CheminVal(sub, T)$ ou $CheminVal(sub)$ lorsqu'il n'y a pas d'ambiguïté sur T.
     Un chemin vide dans T d'origine et extrémité égale à s_1 est valide par rapport à une composition d'étiquettes sub de la forme ($opt\ g_1\ \epsilon$) si et seulement si $g_1(s_1)$.
   - L'ensemble des chemins valides dans T par rapport à une composition d'étiquettes sub sous forme canonique (*i.e.* de forme sub_1 *alt...alt* sub_m) est défini par :
     $CheminVal(sub, T) = \bigcup_{i=1}^{m}\ CheminVal(sub_i, T)$.
2. Chemins associés à une composition d'étiquettes.
   L'ensemble des chemins associés à une composition d'étiquettes sub est défini par l'ensemble des chemins valides par rapport à sub et dont l'origine appartient à la précondition de sub (*i.e.* les chemins déclenchables), on le note $Chemin(sub, T)$.
   $Chemin(sub, T) = \{c; c \in CheminVal(sub, T)\ \wedge\ origine(c) \in pre(sub)\}$.

Exemple 4.2.2. Nous considérons le STE T_2 de la Figure 4.1. Soient sub_1 et sub_2 deux compositions d'étiquettes sur T_2. Nous donnons l'ensemble des chemins et l'ensemble des chemins valides associés à sub_1 et sub_2.

1. $sub_1 \equiv bb$ *seq* cc, alors on a :
   - $CheminVal(sub, T) = \{q_7 \xrightarrow{bb} q_8 \xrightarrow{cc} q_9,\ q_2 \xrightarrow{bb} q_3 \xrightarrow{cc} q_5\}$
   - $pre(sub_1) = \{q_7\}$
   - $Chemin(sub, T) = \{q_7 \xrightarrow{bb} q_8 \xrightarrow{cc} q_9\}$
2. $sub_2 \equiv\ bb$ *seq* $(opt(stateT_1 \in \{q_8, q_3\})cc)$, alors on a :

- $pre(sub_1) = \{q_7,\ q_2\}$
- $Chemin(sub, T) = CheminVal(sub, T) = \{q_7 \xrightarrow{bb} q_8 \xrightarrow{cc} q_9,\ q_2 \xrightarrow{bb} q_3 \xrightarrow{cc} q_5\}$

Nous reprenons la définition des compositions d'étiquettes sur des STEs et nous la généralisons pour les STE gardés. De même les notions introduites telles que précondition, chemin sont redéfinies pour les compositions d'étiquettes sur des STEs gardés.

Définition 4.2.15 (Composition d'étiquettes sur un STE gardé)**.**
Soient $T = (A,\ G(x),\ D,\ S,\ s_0,\ F,\ \rightarrow)$ un STE gardé, P un ensemble de prédicats unaires de profil $S \times D \rightarrow Bool$ et $\{seq, alt, opt, loop\}$ un ensemble de constructeurs.

- L'ensemble SUB (noté aussi $\mathcal{E}(A)$) des compositions d'étiquettes associées à T est défini inductivement de la même manière que dans la définition 4.2.12.
- La précondition d'une composition d'étiquettes sub associée à un STE gardé T est notée par $pre(sub, T)$ ou $pre(sub)$, c'est un ensemble de couples constituées d'un état de T et d'une valeur de x définis comme suit.
  - Si $sub = opt(g_1)skip$ alors $pre(sub) = S \times D$
  - Si $sub = opt(g_1)e_1\ seq...\ seq\ opt(g_n)e_n$ alors
    $pre(sub) = \{(q_1, x);\ q_1 \in S \wedge$
    $\bigwedge_{j=1}^{n}((\forall q_2 \in S), ..., (\forall q_n \in S)\ (\forall x \in D)$
    $[q_1 \xrightarrow{[g'_1(x)]e_1} q_2 \xrightarrow{e_2} ... \xrightarrow{[g'_{j-2}(x)]e_{j-2}} q_{j-1} \xrightarrow{[g'_{j-1}]e_{j-1}} q_j \wedge$
    $g_1(q_1, x)\ \wedge ... \wedge\ g_j(q_j, x) \Rightarrow q_j \xrightarrow{[g'_j(x)]e_j}])\}$
  - Pour toute composition d'étiquettes sub de forme canonique $sub_1\ alt...alt\ sub_m$, $pre(sub) = \bigcap_{i=1}^{m} pre(sub_i)$.
- La notion de chemin associé à une composition d'étiquettes se définit de la même manière que la définition 4.2.14. La seule différence est que nous redéfinissons la notion de chemin valide comme suit. Un chemin dans T de la forme $s_i \xrightarrow{[g'_i]e_i} s_{i+1}$ où $1 \leq i \leq n$ est valide par rapport à une composition d'étiquettes sub de la forme $(opt\ g_1\ e_1)\ seq...seq\ (opt\ g_n\ e_n)$ si et seulement si $\forall i \in [1, n]\ g_i(s_i, x)\ \wedge\ g'_i(x)$.

Remarque 4.2.1. La proposition 4.2.1 concernant la forme canonique pour les compositions d'étiquettes reste valable pour les STEs gardés

La notion de composition d'étiquettes a été définie relativement à un seul STE, pour prendre en compte plusieurs STE indépendants, nous introduisons la définition suivante.

Définition 4.2.16 (Composition d'étiquettes sur plusieurs STEs)**.** Soient T_1,..., T_n, n STEs, dont les alphabets sont deux à deux disjoints, et T leur produit libre. Une composition d'étiquettes sur T_1,...,T_n est définie par une composition d'étiquettes sur T.

4.3 Traduction en B

4.3.1 Système de transitions étiqueté

Dans ce paragraphe, nous définissons la traduction des systèmes de transitions étiquetés en des machines B. Soit $T = (A,\ S,\ s_0,\ F,\ \rightarrow)$ un système de transition étiqueté, tel que $S = \{s_0, \ldots, s_n\}$. Nous lui associons un modèle B, appelé également T et défini par les éléments suivants :

- Chaque état de T est modélisé par une variable $stateT$, appelée variable de contrôle dont les valeurs possibles appartiennent à S.

– Pour chaque étiquette $e \in A$, nous considérons la suite des transitions de T de la forme (s_i, e, s'_i) et introduisons une opération B de la forme $e \mathrel{\widehat{=}} P|B$ où P est une précondition et B correspondant au corps de l'opération. Plus précisément :

$$P = \bigvee_{i=1}^{m} (stateT = s_i)$$

$$B = stateT = s_1 \Rightarrow\ stateT := s'_1 [] \\ ...[] \\ stateT = s_m \Rightarrow\ stateT := s'_m []$$

P peut aussi s'écrire : $stateT \in pre(e, T)$.

L'équivalent verbeux de la substitution B est :

$$SELECT stateT = s_1\ THEN\ stateT := s'_1 \\ WHEN stateT = s_2\ THEN\ stateT := s'_2 \\ \dots \\ WHEN stateT = s_m\ THEN\ stateT := s'_m \\ END$$

Dans la traduction présentée, le lien entre un STE et sa spécification B est évident puisque les opérations modélisent directement les transitions du STE. Si l'on se place au point de vue de la sémantique opérationnelle en B, le modèle B obtenu par traduction a comme sémantique le STE dont il est lui même la traduction, cette dernière remarque peut être corroborée par proB [LB03, LB08] lorqu'on lui soumet des traductions de STEs.

Exemple 4.3.1. La Figure 4.2 présente les modèles B associés aux deux STEs de la Figure 4.1. Ces deux STEs ont les mêmes étiquettes.

Nous étendons cette traduction pour tenir compte des STEs gardés.
Soit $T = (A,\ G(x), D,\ S,\ s_0,\ F,\ \rightarrow)$ un système de transition étiqueté gardé. Nous lui associons un modèle B que nous appelons aussi T et qui est défini de la même manière qu'un STE sans garde, avec les différences suivantes.

– Les variables x sont déclarées soit dans la spécification T soit dans une autre spécification liée à T.

– Une opération $e = P|B$, associée à une étiquette e, est définie par

$$P = \bigvee_{i=1}^{m} (stateT = s_i \wedge g_i(x))$$

$$B = stateT = s_1 \wedge g_1(x) \Rightarrow\ stateT := s'_1 [] \\ ...[] \\ stateT = s_m \wedge g_m(x) \Rightarrow\ stateT := s'_m []$$

4.3.2 Relation entre les états de deux STEs

Nous sommes amenés à définir des relations entre les états de deux STEs. En B, il est simple de définir une relation binaire par un ensemble de couples, un couple étant de la forme $x \mapsto y$. Il faut toutefois s'assurer de l'accès en lecture des données permettant de définir la relation. Nous déclarons les relations dans la clause **ABSTRACT_CONSTANTS** et les définissons dans la clause **PROPERTIES**. L'ajout de la formule (stateF $\mapsto$ stateT) $\in$ RR dans la clause **INVARIANT** permet de mettre en relation par RR les états des deux STEs considérés.

Exemple 4.3.2. La Figure 4.3 présente une relation entre les spécifications des deux STEs présentées dans la Figure 4.2.

```
MACHINE M1
SETS
 F_States = {p10,p11,p12,p13,p14,p15}
VARIABLES stateF
INVARIANT stateF ∈F_States
INITIALISATION  stateF:=p10
OPERATIONS
 aa =
  PRE   stateF =p10 THEN
   SELECT stateF = p10 THEN stateF:=p11
   END
  END;
 bb=
  PRE   stateF =p11 THEN
   SELECT stateF = p11 THEN stateF:∈{p12,p13}
   END
  END;
 cc=
  PRE   stateF =p12 THEN
   SELECT stateF = p12 THEN stateF:=p14
   END
  END;
 dd=
  PRE   stateF =p13 THEN
   SELECT stateF = p13 THEN stateF:=p15
   END
  END
END
```

(a) M_1

```
MACHINE M2
SETS
  T_States={q1,q2,q3,q4,q5,q6,q7,q8,q9}
VARIABLES stateT
INVARIANT stateT ∈T_States
INITIALISATION  stateT:=q1
OPERATIONS
 aa =
  PRE   stateT=q1 THEN
   SELECT stateT = q1 THEN  stateT:∈{q2,q7}
   END
  END;
 bb=
  PRE   stateT∈{q2,q7} THEN
   SELECT  stateT = q2 THEN  stateT:∈{q3,q4}
    WHEN   stateT = q7 THEN  stateT:=q8
   END
  END;
 cc=
  PRE   stateT∈{q3,q8} THEN
   SELECT  stateT = q3 THEN  stateT:=q5
    WHEN   stateT = q8 THEN  stateT:=q9
   END
  END;
 dd=
  PRE   stateT=q4 THEN
   SELECT  stateT = q4 THEN  stateT:=q6
   END
  END
END
```

(b) M2

FIGURE 4.2 – Machines B associées aux STEs T_1 et T_2 (Figure 4.1)

```
ABSTRACT_CONSTANTS
    RR
PROPERTIES
    RR ∈ F_States ↔ T_States ∧
    RR = {(p10 ↦q1),
          (p11 ↦ q2),(p11 ↦q7),
          (p12 ↦q3),(p12 ↦q8),
          (p13 ↦q4),
          (p14 ↦q5),(p14 ↦q9),
          (p15 ↦q6)}
```

FIGURE 4.3 – Relation entre les variables de M1 et M2

4.3.3 Composition d'étiquettes

Traduction en substitutions généralisées

Nous définissons une correspondance entre l'ensemble des compositions d'étiquettes et l'ensemble des substitutions généralisées sous forme d'une application Φ. L'ensemble, SUB, des compositions d'étiquettes ayant été défini par induction (Déf. 4.2.12, p. 63) l'application Φ est naturellement récursive. La traduction définie par Φ est immédiate. La traduction de la composition d'étiquettes vide ϵ est la substitution généralisée, **skip**. La traduction d'une étiquette e est la substitution généralisée associée à e, que nous notons également e. Lorsque e doit être explicitée, nous lui faisons correspondre la substitution généralisée obtenue par la traduction du STE considéré. Aux constructeurs *seq*, *alt*, *opt* et *loop* nous associons respectivement les constructeurs

; (séquence), [] (substitution de choix borné), $\Rightarrow$ (substitution gardée) et $\mathcal{W}$ (boucle), nous nous restreignons aux boucles «pour». Le tableau 4.1 montre la définition de Φ.

Composition d'étiquettes (sub)	Substitutions généralisées ($\Phi(sub)$)
ϵ	$skip$
e	e_1
sub_1 seq sub_2	$\Phi(sub_1); \Phi(sub_2)$
sub_1 alt sub_2	$\Phi(sub_1) [] \Phi(sub_2)$
opt p sub_1	$p \Rightarrow \Phi(sub_1)$
$loop$ p n sub_1	$\mathcal{W}(n > 1, \Phi(sub_1), p, n)$

TABLE 4.1 – Correspondance entre composition d'étiquettes et substitutions généralisées

La définition de Φ a une double conséquence :
- comme SUB est défini inductivement $\Phi(SUB)$ peut être également définie inductivement avec $skip$ et e comme substitutions de base et par des règles identiques à celles de la construction inductive de SUB en remplaçant les constructeurs seq, alt, opt et $loop$ respectivement par ;, [], $\Rightarrow$ et $\mathcal{W}$. Les propriétés de $\Phi(SUB)$ pourront donc se démontrer par induction.
- les propriétés prouvées pour les éléments de $\Phi(SUB)$ peuvent être déduites pour les éléments de SUB.

Les propositions suivantes expriment des propriétés sur les substitutions généralisées représentant l'images par Φ de compositions d'étiquettes.

Propriétés

Proposition 4.3.1. *Toute substitution généralisée image par Φ d'une composition d'étiquettes peut s'écrire sous la forme $sub_1 [] ... [] sub_m$ où chaque sub_i est de la forme $g_1 \Rightarrow e_1; ...; g_n \Rightarrow e_n$ ou $g \Rightarrow skip$. Cette forme est appelée la forme canonique de la substitution généralisée.*

Démonstration. La preuve se fait par induction structurelle sur la substitution généralisée.
- Pour les cas de base :
  - si la substitution est $skip$, on a le résultat suivant
    $skip \equiv true \Rightarrow skip$
  - si la substitution est e, on a le résultat suivant
    $e \equiv true \Rightarrow e$
- Pour les cas d'induction, on suppose que sub et sub' vérifient l'hypothèse d'induction et l'on distingue les cas suivants :
  - Séquencement de compositions d'étiquettes.
    $$\begin{aligned} sub; sub' \equiv\ & [sub_1 [] ... [] sub_n]; [sub'_1 [] ... [] sub'_m] && \text{(par hypothèse d'induction)} \\ \equiv\ & sub_1; sub'_1 [] ... [] sub_1; sub'_m [] && \text{(application de } R8 \text{ et } R9 \text{ Tab. 2.3)} \\ & ... [] \\ & sub_n; sub'_1 [] ... [] sub_n; sub'_m \end{aligned}$$
    La propriété est vérifiée à cause de la forme des sub_i et sub'_i
  - Choix entre des compositions d'étiquettes.
    $$\begin{aligned} sub [] sub' \equiv\ & [sub_1 [] ... [] sub_n] [] [sub'_1 [] ... [] sub'_m] && \text{(par hypothèse d'induction)} \\ \equiv\ & sub_1 [] ... [] sub_n [] sub'_1 [] ... [] sub'_m && \text{(par associativité de } []) \end{aligned}$$
  - Composition d'étiquettes gardée.

$p \Rightarrow sub \equiv \quad p \Rightarrow [sub_1 \,[]\, ... \,[]\, sub_n]$ (par hypothèse d'induction)
$\equiv \quad p \Rightarrow sub_1 \,[]\, ... \,[]\, p \Rightarrow sub_n$ (application de $R10$ Tab. 2.3)

Pour chaque sub_i on a :
$p \Rightarrow sub_i \equiv \quad p \Rightarrow (g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n)$
$\equiv \quad (p \wedge g_1) \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$ (application de $R11$ et $R12$ Tab. 2.3)

– Itération d'une composition d'étiquettes. Toute composition d'étiquettes $\mathcal{W}(n > 1, sub_1, p, n)$ peut s'écrire sous la forme $\underbrace{sub_1; ...; sub_1}_{n\ fois}$.

Nous prouvons les résultats attendus par induction, puisque cette substitution est composée de substitutions utilisant la séquence (;).

CQFD. □

Le lemme suivant est utilisé pour démontrer la proposition 4.3.2.

Lemme 4.3.1. *Toute substitution généralisée sous la forme* $g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$ *où* $e_i \widehat{=} P_i|B_i$ *peut s'écrire de la forme* $P|B$ *où :*

- $P \equiv (g_1 \Rightarrow P_1) \wedge [g_1 \Rightarrow B_1](g_2 \Rightarrow P_2) \wedge ... \wedge [g_1 \Rightarrow B_1; ...; g_{n-1} \Rightarrow B_{n-1}](g_n \Rightarrow P_n)$
- $B \equiv g_1 \Rightarrow B_1; ...; g_n \Rightarrow B_n$

Démonstration. On effectue la démonstration par récurrence sur n

1. Cas de base. Si $n = 1$, on a
   $g_1 \Rightarrow P_1|B_1 \equiv g_1 \Rightarrow P_1|g_1 \Rightarrow B_1$ (application de $R1$ Tab. 2.3)
   le résultat est donc vrai pour 1.
2. Cas général. Hypothèse de récurrence : on suppose la forme vrai pour $n-1$
   $g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_{n-1} \Rightarrow e_{n-1}$
   $\equiv \quad ((g_1 \Rightarrow P_1) \wedge [g_1 \Rightarrow B_1](g_2 \Rightarrow P_2) \wedge ... \wedge [g_1 \Rightarrow B_1; ...; g_{n-2} \Rightarrow B_{n-2}](g_{n-1} \Rightarrow P_{n-1})) \,|\, (g_1 \Rightarrow B_1; ...; g_{n-1} \Rightarrow B_{n-1})$
   $\equiv \quad P'|B'$

   On démontre pour n :
   $g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_{n-1} \Rightarrow e_{n-1}; g_n \Rightarrow e_n$
   $\equiv \quad (P'|B'); (g_n \Rightarrow P_n|B_n)$ (application de l'hypothèse de récurrence)
   $\equiv \quad (P'|B'); (g_n \Rightarrow P_n|g_n \Rightarrow B_n)$ (application $R1$ Tab. 2.3)
   $\equiv \quad P'|(B'; (g_n \Rightarrow P_n|g_n \Rightarrow B_n))$ (application $R2$ Tab. 2.3)
   $\equiv \quad P'|([B'](g_n \Rightarrow P_n)|(B'; g_n \Rightarrow B_n))$ (application $R3$ Tab. 2.3)
   $\equiv \quad (P' \wedge [B'](g_n \Rightarrow P_n))|(B'; g_n \Rightarrow B_n)$ (application $R4$ Tab. 2.3)
   $\equiv \quad P|B$ avec P et B sous la forme attendue

CQFD. □

Proposition 4.3.2. *Toute substitution généralisée sous forme canonique peut s'écrire sous la forme* $P|B$ *où* $P \equiv P_1 \wedge ... \wedge P_n$ *et* $B \equiv B_1 \,[]\, ... \,[]\, B_n$.

Démonstration. Soit $sub_1 \,[]\, sub_2 \,[]\, ... \,[]\, sub_n$ la forme canonique de la substitution. Nous effectuons une preuve par récurrence sur n.

1. Cas de base. Pour $n = 1$,
   – Si $sub_1 = g \Rightarrow skip$, alors $sub_1 = True|g \Rightarrow skip$.
   – Si $sub_1 = g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$, alors $sub_1 = P_1|B_1$ (d'après le lemme 4.3.1).
   $sub_1 = P_1|B_1$ c'est démontré
2. Cas général. Hypothèse de récurrence : on suppose la propriété vraie pour $n-1$.

$$
\begin{array}{lll}
 & sub_1 \,[]\, sub_2 \,[]\, ... \,[]\, sub_n & \\
\equiv & P_1|B_1 \,[]\, ... \,[]\, P_{n-1}|B_{n-1} \,[]\, P_n|B_n & \text{(Lem. 4.3.1)} \\
\equiv & (P'|B') \,[]\, (P_n|B_n) & \text{(Hypothèse de récurrence)} \\
\equiv & P'|(B' \,[]\, (P_n|B_n)) & (R5 \text{ Tab. 2.3}) \\
\equiv & P'|(P_n|(B' \,[]\, B_n)) & (R6 \text{ Tab. 2.3}) \\
\equiv & P' \wedge P_n|(B' \,[]\, B_n) & (R4 \text{ Tab. 2.3})
\end{array}
$$

CQFD. □

4.4 Relations entre STEs et raffinement B

Dans ce paragraphe nous mettons en évidence des relations entre STEs. Pour établir ces relations, nous utilisons des constructions B qui utilisent la traduction des STEs en B et les clauses **REFINES** et **INCLUDES** de B. La correspondance entre STEs se fait par une mise en relation des états et des étiquettes. Nous considérons successivement les cas suivants :

- Raffinement B faisant intervenir deux STEs sur un même alphabet et une relation entre les états.
- Raffinement B faisant intervenir deux STEs sur des alphabets différents, une relation entre les états et une relation entre les étiquettes.
- Raffinement B faisant intervenir un STE et plusieurs STEs, une relation entre les états et une relation entre les étiquettes.
- Raffinement B entre deux STEs gardés.

Pour chacun de ces cas, nous établissons une proposition.

4.4.1 Relation entre deux STEs sur un même alphabet

Nous commençons par présenter la construction B utilisée. Nous caractérisons, ensuite, le raffinement B entre deux STEs par la proposition 4.4.1 et nous comparons cette caractérisation aux relations de simulation introduites par Milner [Mil89], van Glabbeek et Bloom [BIM95].

Construction B

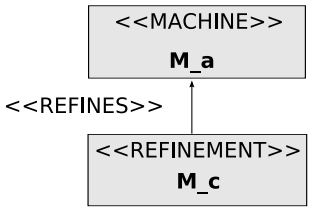


FIGURE 4.4 – Construction B

Nous considérons deux STEs $T_1 = (A,\ S_1,\ s_0^1,\ F1,\ \rightarrow_1)$ et $T_2 = (A,\ S_2,\ s_0^2,,\ F2,\ \rightarrow_2)$ dont les transitions sont étiquetées par les éléments d'un même alphabet A et une relation R de S_1 vers S_2. Nous construisons les spécifications B, M_a et M_c associées respectivement à T_1 et T_2. M_a est une machine abstraite B et M_c est un raffinement (au sens de B) de M_a (Figure 4.4). M_a et M_c sont obtenues par traductions respectives de T_1 et T_2, en appliquant les règles de traduction définies dans 4.3.1 : les variables $stateT_1$ et $stateT_2$ modélisent les états des STEs apparaissant respectivement dans T_1 et T_2, la relation R entre les états des deux STEs est définie dans M_c, de même que l'assertion $(stateT_1 \mapsto stateT_2) \in R$ qui est l'invariant de collage.

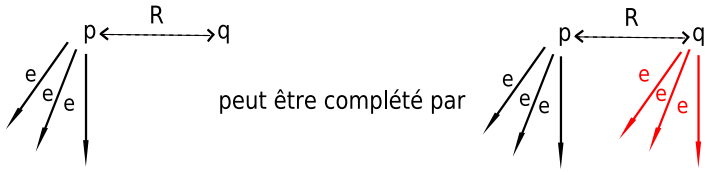
Dans la suite nous dirons que T_1 est le STE abstrait (représenté par la machine abstraite B, M_a), que T_2 est le STE concret (représenté par le raffinement B, M_c) et que T_2 raffine T_1 si et seulement si M_c est un raffinement de M_a.

Raffinement B

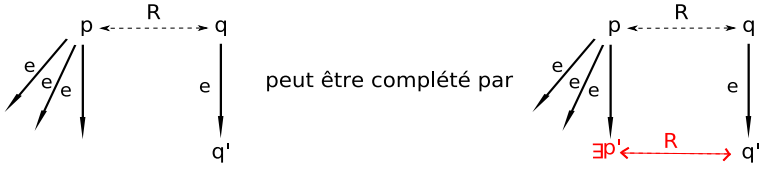
La proposition suivante met en évidence des propriétés nécessaires au raffinement d'un STE par un autre STE.

Proposition 4.4.1. *Si T_2 raffine T_1 par rapport à R, alors on a les trois propriétés suivantes :*

1. $(s_0^1, s_0^2) \in R$
2. $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R\ \wedge\ p \xrightarrow{e}_1 \Rightarrow q \xrightarrow{e}_2]$



3. $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$
$[(p,q) \in R \wedge q \xrightarrow{e}_2 q' \wedge p \xrightarrow{e}_1 \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p',q') \in R]]$



Démonstration. La preuve d'un raffinement en B consiste principalement à démontrer des obligations de preuve pour l'initialisation et pour chaque opération apparaissant dans M_a et M_c. Soient e une opération et $P_a|B_a$, $P_c|B_c$ les définitions respectives de e dans M_a et M_c. $I_a \equiv stateT_1 \in S_1$ et $I_c \equiv stateT_2 \in S_2\ \wedge\ (stateT_1 \mapsto stateT_2) \in R$ sont les invariants respectifs de M_a et M_c. L'obligation de preuve relative à l'initialisation est la formule $[Init_c]\neg[Init_a]\neg I_c$. L'obligation de preuve relative à l'opération e est la formule $I_a\ \wedge\ I_c\ \wedge\ P_a \Rightarrow P_c\ \wedge\ [B_c]\neg[B_a]\neg I_c$ qui peut être décomposée en les deux formules $I_a\ \wedge\ I_c\ \wedge\ P_a \Rightarrow P_c$ et $I_a\ \wedge\ I_c\ \wedge\ P_a \Rightarrow [B_c]\neg[B_a]\neg I_c$. Remarquons que l'invariant I_a joue simplement un rôle de typage pour la variable $stateT_1$, il en est de même pour la partie $stateT_2 \in S_2$ de l'invariant I_c. Ainsi, ils peuvent être ignorés dans les obligations de preuves, nous allons donc considérer les trois obligations de preuve suivantes :

1. $\phi_1 \equiv [Init_c]\neg[Init_a]\neg(stateT_1 \mapsto stateT_2) \in R$
2. $\phi_2 \equiv (stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow P_c$
3. $\phi_3 \equiv (stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow [B_c]\neg[B_a]\neg(stateT_1 \mapsto stateT_2) \in R$.

Nous allons montrer qu'à chaque formule ϕ_1, ϕ_2 et ϕ_3 correspondent les conditions associées aux trois items (1), (2) et (3) de la proposition.

1. ϕ_1, on a $Init_c \equiv stateT_2 := s_0^2$ et $Init_a \equiv stateT_1 := s_0^1$ d'où

$$\begin{aligned}\phi_1 \equiv\ & [stateT_2 := s_0^2]\neg[stateT_1 := s_0^1]\neg(stateT_1 \mapsto stateT_2) \in R \\ \equiv\ & (s_0^1 \mapsto s_0^2) \in R\end{aligned}$$

2. ϕ_2, on a $P_a \equiv stateT_1 \in pre(e, T_1)$ et $P_c \equiv stateT_2 \in pre(e, T_2)$. ϕ_2 s'écrit donc

$$\phi_2 \ \equiv\ (stateT_1 \mapsto stateT_2) \in R \ \wedge\ stateT_1 \in pre(e, T_1) \Rightarrow stateT_2 \in pre(e, T_2)$$

Dans ϕ_2 $stateT_1$ et $stateT_2$ sont des variables libres qui peuvent être interprétées comme des variables universellement quantifiées, en les renommant respectivement en p et q et en utilisant le fait que $stateT_1 \in pre(e, T_1) \equiv stateT_1 \xrightarrow{e}_1$ et $stateT_2 \in pre(e, T_2) \equiv stateT_2 \xrightarrow{e}_2$, on obtient l'item (2) de la proposition.

$$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(p, q) \in R \ \wedge\ p \in pre_{T_1}(e)] \Rightarrow q \in pre_{T_2}(e)$$

3. ϕ_3, les définitions de B_a et B_c sont :
   - $B_a \equiv a_1 \Rightarrow b_1 \,[]\, ... \,[]\, a_n \Rightarrow b_n$
     $\equiv stateT_1 = p_1 \Rightarrow stateT_1 := p'_1 \,[]\, ... \,[]\, stateT_1 = p_n \Rightarrow stateT_1 := p'_n$
     avec $\xrightarrow{e}_1 = \{(p_1, p'_1), ..., (p_n, p'_n)\}$
   - $B_c \equiv c_1 \Rightarrow d_1 \,[]\, ... \,[]\, c_m \Rightarrow d_m$
     $\equiv stateT_2 = q_1 \Rightarrow stateT_2 := q'_1 \,[]\, ... \,[]\, stateT_2 = q_m \Rightarrow stateT_2 := q'_m$
     avec $\xrightarrow{e}_2 = \{(q_1, q'_1), ..., (q_m, q'_m)\}$

En remplaçant B_a, ϕ_3 s'écrit :

$$\begin{aligned}\phi_3 \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] \\ & \neg[a_1 \Rightarrow b_1 \,[]\, ... \,[]\, a_n \Rightarrow b_n]\neg(stateT_1 \mapsto stateT_2) \in R] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] && (\mathcal{WP}\ 5) \\ & \neg([a_1 \Rightarrow b_1]\neg(stateT_1 \mapsto stateT_2) \in R \wedge \\ & ... \wedge \\ & [a_n \Rightarrow b_n]\neg(stateT_1 \mapsto stateT_2) \in R)] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] && (\mathcal{WP}\ 6) \\ & \neg(a_1 \Rightarrow [b_1]\neg(stateT_1 \mapsto stateT_2) \in R \wedge \\ & ... \wedge \\ & (a_n \Rightarrow [b_n])\neg(stateT_1 \mapsto stateT_2) \in R)] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] && (\text{Introduire } \neg) \\ & a_1 \ \wedge\ \neg([b_1]\neg(stateT_1 \mapsto stateT_2) \in R) \vee \\ & ... \vee \\ & a_n \ \wedge\ \neg([b_n]\neg(stateT_1 \mapsto stateT_2) \in R)] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] \\ & \textstyle\bigvee_{j=1}^{n}(a_j \ \wedge\ \neg([b_j]\neg(stateT_1 \mapsto stateT_2) \in R)) \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] \\ & \textstyle\bigvee_{j=1}^{n}(stateT_1 = p_j \ \wedge\ \neg([stateT_1 := p'_j]\neg(stateT_1 \mapsto stateT_2) \in R)) \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] && (\mathcal{WP}\ 1\ ) \\ & \textstyle\bigvee_{j=1}^{n}(stateT_1 = p_j \ \wedge\ (p'_j \mapsto stateT_2) \in R)) \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] \\ & (\exists (p, p') \in \xrightarrow{e}_1)[stateT_1 = p \ \wedge\ (p' \mapsto stateT_2) \in R]] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] \\ & (\exists p \in S_1)\ (\exists\ p' \in S_1)\ [stateT_1 = p \ \wedge\ p \xrightarrow{e}_1 p' \ \wedge\ (p' \mapsto stateT_2) \in R] \\ \equiv\ & (stateT_1 \mapsto stateT_2) \in R \ \wedge\ P_a \Rightarrow [B_c] && (\{q \backslash StateT_1\}) \\ & (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge\ (p' \mapsto stateT_2) \in R]\end{aligned}$$

En remplaçant B_c, on obtient :

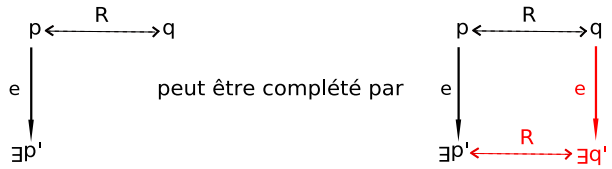
$$\begin{array}{lll}
\phi_3 \equiv & (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow & \\
& [c_1 \Rightarrow d_1 \ [] \ ... \ [] \ c_m \Rightarrow d_m] & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow & (\mathcal{WP}\ 5) \\
& [c_1 \Rightarrow d_1](\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] \ \wedge & \\
& ... \ \wedge & \\
& [c_m \Rightarrow d_m](\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & \bigwedge_{i=1}^{m}((stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow [c_i \Rightarrow d_i] & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & \bigwedge_{i=1}^{m}((stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ c_i \Rightarrow [d_i] & (\mathcal{WP}\ 6) \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & (\forall i \in [1..m])[(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ c_i \Rightarrow [d_i] & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & (\forall i \in [1..m])[(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ stateT_2 = q_i \Rightarrow & \\
& [stateT_2 := q_i'] & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R] & \\
\equiv & (\forall i \in [1..m])[(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ stateT_2 = q_i \Rightarrow & (\mathcal{WP}\ 1) \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q_i') \in R] & \\
\equiv & (\forall (q, q') \in \xrightarrow{e}_2)\ [(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ stateT_2 = q \Rightarrow & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q') \in R] & \\
\equiv & (\forall q \in S_2)\ (\forall q' \in S_2)\ [(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge & \\
& stateT_2 = q \ \wedge \ q \xrightarrow{e}_2 q' \Rightarrow & \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q') \in R] & \\
\equiv & (\forall q' \in S_2)\ [(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ stateT_2 \xrightarrow{e}_2 q' \Rightarrow & (\{q \backslash StateT_2\}) \\
& (\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q') \in R] &
\end{array}$$

On démontre, ainsi, le troisième item de la proposition.

CQFD. □

Corollaire 4.4.1. *Si T_2 raffine T_1 par rapport à R, alors on peut déduire la propriété suivante :*

$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\exists p' \in S_1)$
$[((p, q) \in R \ \wedge \ p \xrightarrow{e}_1 p') \Rightarrow (\exists q' \in S_2)\ [(p', q') \in R \ \wedge \ q \xrightarrow{e}_2 q']]$



Démonstration. Distinguons deux cas :

- Si non $p \xrightarrow{e}_1$ la partie gauche de l'implication est fausse, l'implication est donc vraie, la formule est trivialement vraie.
- si $p \xrightarrow{e}_1$, soient $e \in A$, $p \in S_1$ et $q \in S_2$ quelconque tel que $(p, q) \in R$, d'après l'item (2) de la proposition 4.4.1
  $(p, q) \in R \ \wedge \ p \xrightarrow{e}_1 \Rightarrow q \xrightarrow{e}_2$, c'est-à-dire qu'il existe $q' \in S_2$ tel que $q \xrightarrow{e}_2 q'$

d'après l'item (3) de la proposition 4.4.1 on a
$[(p,q) \in R \ \wedge \ q \xrightarrow{e}_2 q' \ \wedge \ p \xrightarrow{e}_1 \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p' \ \wedge \ (p',q') \in R]]$
c'est-à-dire qu'il existe $p' \in S_1$ tel que $p \xrightarrow{e}_1 p' \ \wedge \ (p',q') \in R$ en définitive, il existe $p' \in S_1$ et $q' \in S_2$ vérifiant $p \xrightarrow{e}_1 p'$, $(p',q') \in R$ et $q \xrightarrow{e}_2 q'$ ce qui démontre le corollaire.

□

Exemple 4.4.1. La Figure 4.5 montre un exemple de raffinement d'une étiquette *ee* d'un STE T_1 par un STE T_2, les états de T_1 sont dénotés p_i alors que ceux de T_2 sont dénotés q_i, la relation R est représentée partiellement par des doubles flèches pointillées reliant les états des deux STEs. Le texte des spécifications B est dans la section A.2 p. 165. Pour prouver le raffinement, 16 POs ont été générées dont 6 ont été prouvées de manière interactive (voir section A.2).

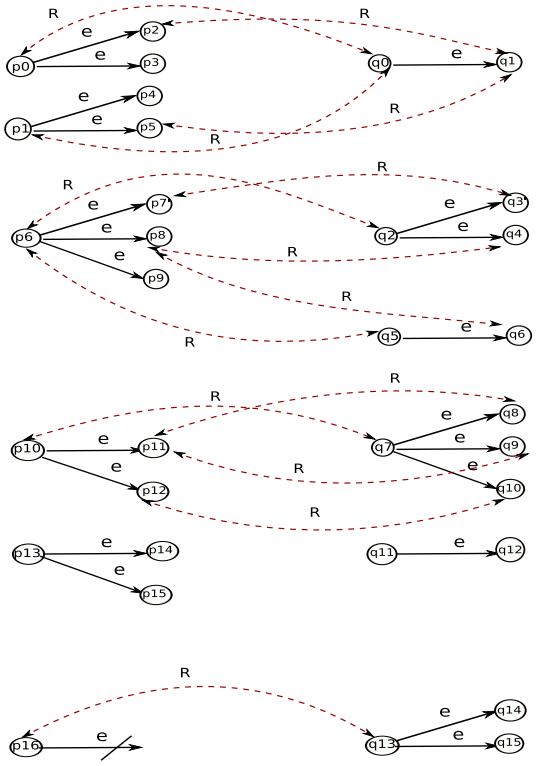


FIGURE 4.5 – Raffinement d'une étiquette *ee*

Relation entre raffinement B et simulation

Nous allons présenter maintenant certains cas où le raffinement B entre deux STEs implique une relation de simulation. Nous commençons par introduire ce premier corollaire, dans lequel nous prenons comme hypothèse le fait que le STE abstrait est déterministe.

Corollaire 4.4.2. *Si T_2 raffine T_1 par rapport à R tel que si T_1 est déterministe alors $T_1 \preccurlyeq_R T_2$.*

Démonstration. La formule du corollaire 4.4.1 est :
$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\exists p' \in S_1)$

$$[((p,q) \in R \ \wedge \ p \xrightarrow{e}_1 p') \Rightarrow (\exists q' \in S_2)\ [(p',q') \in R \ \wedge \ q \xrightarrow{e}_2 q']]$$

Comme T_1 est déterministe, pour tout état p de S_1 et pour tout e de A il existe au plus une transition de la forme $p \xrightarrow{e}_1 p'$. La quantification existentielle sur la variable p' peut être remplacée par une quantification universelle, on a donc

$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall p' \in S_1)$

$$[((p,q) \in R\ \wedge\ p \xrightarrow{e}_1 p') \Rightarrow (\exists q' \in S_2)\ [(p',q') \in R\ \wedge\ q \xrightarrow{e}_2 q']]$$

comme de plus $(s_0^1, s_0^2) \in R$, on a bien $T_1 \preccurlyeq_R T_2$. □

Dans le but de comparer la relation de raffinement entre deux STEs et les relations de ready-simulation et de bisimulation, nous introduisons la propriété suivante J concernant la relation R considérée :

$$(a)\quad J : (\forall e \in A)\ (p \in S_2)\ (q \in S_1)\ [(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 \Rightarrow p \xrightarrow{e}_1]$$

D'un point de vue pratique pour tenir compte en B de la propriété (a) il suffit d'ajouter dans la clause **ASSERTIONS** le prédicat défini par $P_c \Rightarrow P_a$ (P_a et P_c étant les préconditions respectives de l'opération e dans les machines B, M_a et M_c), l'ajout de ce prédicat génère l'obligation de preuve suivante

$$I_a\ \wedge\ I_c \Rightarrow (P_c \Rightarrow P_a)$$

que l'on peut réduire à la forme (a).

La proposition suivante établit la relation entre raffinement B et simulation.

Proposition 4.4.2. *Soient T_1 et T_2 deux STEs, R une relation de S_1 vers S_2 et J : $(\forall e \in A)\ (p \in S_2)\ (q \in S_1)\ [(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 \Rightarrow p \xrightarrow{e}_1]$ une assertion.*

- *Si T_2 raffine T_1 par rapport à R et R vérifie J, alors $T_2 \precsim_{R^{-1}} T_1$.*
- *Si T_2 raffine T_1 par rapport à R, R vérifie J et T_1 est déterministe, alors $T_1 \approx_R T_2$.*

Démonstration.

1. Comme T_2 raffine T_1, le troisième item de la proposition 4.4.1 s'écrit :
   $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$

   $$[(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 q'\ \wedge\ p \xrightarrow{e}_1 \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p',q') \in R]]$$

   J s'écrit :

   $$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 \Rightarrow p \xrightarrow{e}_1]$$

   De ces deux formules on déduit
   $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$

   $$[(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 q' \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p',q') \in R]]$$

   qui est équivalent à
   $(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$

   $$[(q,p) \in R^{-1}\ \wedge\ q \xrightarrow{e}_2 q' \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p',q') \in R^{-1}]]$$

   de plus d'après le premier item de la proposition 4.4.1 qui peut s'écrire $(s_0^2, s_0^1) \in R^{-1}$. On a donc bien $T_2 \preccurlyeq_{R^{-1}} T_1$.
   Le second item de la proposition 4.4.1 s'exprime par

   $$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R\ \wedge\ (\exists p' \in S_1)\ p \xrightarrow{e}_1 p' \Rightarrow\ (\exists q' \in S_2)\ q \xrightarrow{e}_2 q']$$

   ou

$$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(q,p) \in R^{-1}\ \wedge\ (\exists p' \in S_1)\ p \xrightarrow{e}_1 p' \Rightarrow\ (\exists q' \in S_2)\ q \xrightarrow{e}_2 q']$$

À partir de cette formule et de la condition $T_2 \preccurlyeq_{R^{-1}} T_1$, nous établissons que $T_2 \precsim_{R^{-1}} T_1$.

2. Si T_1 est déterministe, on a $T_1 \preccurlyeq_R T_2$ (d'après le corollaire 4.4.2). Comme de plus $T_2 \preccurlyeq_{R^{-1}} T_1$ on a bien $T_1 \approx_R T_2$.

CQFD. □

Exemple 4.4.2. On reprend les deux spécifications M2 et M1 (Figure 4.2 p. 67) associées respectivement aux STEs T_2 et T_1 (Figure 4.1 p. 63). On construit le raffinement M2_r à partir de M_2 qui raffine M1 par rapport à la relation définie dans la Figure 4.3 et en ajoutant les assertions ci-dessous (Figure 4.6). On peut alors conclure que $T_2 \precsim_{R^{-1}} T_1$.

```
ASSERTIONS
    (stateT=T1 ⇒ stateF =F10) ∧ (stateT∈{T2,T7}⇒ stateF =F11) ∧
    (stateT∈{T3,T8}⇒stateF =F12) ∧ (stateT=T4 ⇒stateF =F13)
```

FIGURE 4.6 – Assertion dans le raffinement M2

Dans la traduction des STEs en B (Sec. 4.3.1 p. 65), pour chaque étiquette nous avons défini une opération avec des préconditions, une alternative à cette traduction est de ne pas considérer les préconditions pour les opérations (cela revient à avoir des conditions vraies). Sous cette hypothèse nous avons la proposition suivante.

Proposition 4.4.3. *Si T_2 raffine T_1 par rapport à R et si dans la traduction des STEs les opérations associées aux étiquettes n'ont pas de préconditions alors $T_2 \preccurlyeq_{R^{-1}} T_1$.*

Démonstration. Considérons les formules ϕ_1, ϕ_2 et ϕ_3 de la preuve de la proposition 4.4.1. ϕ_1 est inchangée et peut s'exprimer par $(s_0^2, s_0^1) \in R^{-1}$, ϕ_2 est trivialement vraie, ϕ_3 peut se simplifier (en tenant compte que $P_a \equiv vrai$) en

$$(\forall e \in A)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$$
$$[(p,q) \in R\ \wedge\ q \xrightarrow{e}_2 q' \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p',q') \in R]]$$

Cette formule avec la condition $(s_0^2, s_0^1) \in R^{-1}$ exprime que $T_2 \preccurlyeq_{R^{-1}} T_1$. □

4.4.2 Relation entre deux STEs avec des étiquettes différentes

Nous présentons d'abord la construction B utilisée qui fait intervenir deux STEs, ensuite nous montrons les propriétés du raffinement en terme de relation entre les deux STEs. Puis, nous introduisons la définition de raffinement de chemins et nous comparons les propriétés du raffinement avec cette nouvelle notion.

Construction B

Nous considérons deux STEs $T_1 = (A_1,\ S_1,\ s_0^1,\ F1,\ \rightarrow_1)$ et $T_2 = (A_2,\ S_2,\ s_0^2,,\ F2,\ \rightarrow_2)$ tels que $A_1 \cap A_2 = \emptyset$, une relation R de S_1 vers S_2 et une application χ de A_1 vers $\mathcal{E}(A_2)$. Nous construisons les spécifications B, M_a et M_c traductions respectives de T_1 et T_2. Nous construisons, ensuite, le raffinement M_c' qui raffine M_a et inclut M_c. M_c' est obtenue par traduction de R et de χ. M_c' comporte toutes les opérations de M_a et chaque opération est définie par une composition d'étiquettes grâce à l'application χ. L'initialisation de M_c' est celle de M_c. L'invariant de M_c' est un invariant de collage qui lie les variables de M_a et M_c. Il est de la forme :$(stateT_1 \mapsto stateT_2) \in R$.

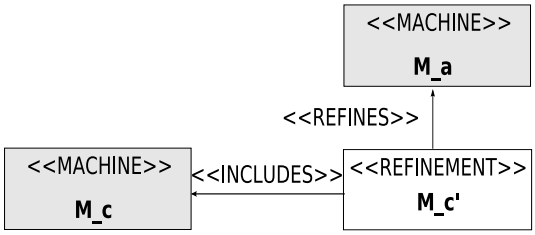


FIGURE 4.7 – Construction B

La construction B est donnée par la Figure 4.7. Si le raffinement B, M'_c est démontré correct, on dit que T_2 raffine T_1 par rapport à R et χ.

Dans cette construction, nous utilisons la spécification M'_c qui va nous permettre de vérifier, par la suite certaines propriétés entre M_a et M_c. M'_c est une spécification B particulière, nous lui associons la sémantique opérationnelle suivante.

Définition 4.4.1.

Soit M'_c une spécification B obtenue par traduction d'une relation R et d'une application χ entre les spécifications B associées aux deux STEs T_1 et T_2. La sémantique opérationnelle associée à M'_c est définie par un ensemble de chemins qui peut être vide, comme suit :
$\{c,\ c \in Chemin(T_2) \wedge c_2 = (q_1 \xrightarrow{b_{11}}_2 q_{21}...q_{n1} \xrightarrow{b_{n1}}_2 q_2)...(q_m \xrightarrow{b_{1m}}_2 q_{2m}...q_{nm} \xrightarrow{b_{nm}}_2 q_{m+1}) \wedge q_1 = s_0^2 \ \wedge\ \forall i \in [1..m]\ \exists a_i \in A_1[q_i \xrightarrow{b_{1i}}_2 q_{2i}...q_{ni} \xrightarrow{b_{ni}}_2 q_{i+1} \in Chemin(\chi(a_i), T_2)]\}$

Raffinement B

On introduit les deux lemmes suivants qui vont servir à la preuve de la proposition 4.4.4.

Lemme 4.4.1. *Soit la substitution* $sub = g_1 \Rightarrow e_1; ...; g_n \Rightarrow e_n$ *où* $e_i \mathrel{\widehat{=}} P_i|B_i$ *et* $B_i = (stateT = q_{i1} \Rightarrow stateT := q'_{i1}\ [] ...\ []\ stateT = q_{ik_i} \Rightarrow stateT := q'_{ik_i})$ *et soit le prédicat* R*, alors* sub *peut s'écrire sous la forme* $P|B$ *et le prédicat* $[B]R$ *est donné par la formule suivante :*

$$(\forall q_2 \in S)...(\forall q_{n+1} \in S)\ ((g_1(stateT)\ \wedge\ g_2(q_2)\ \wedge\ ....\ \wedge\ g_n(q_n)) \wedge stateT \xrightarrow{e_1} q_2...q_n \xrightarrow{e_n} q_{n+1} \Rightarrow [stateT := q_{n+1}]R)$$

Démonstration. D'après le lemme 4.3.1 sub peut s'écrire sous la forme $P|B$ avec $B \equiv g_1 \Rightarrow B_1; ...; g_n \Rightarrow B_n$.

Démontrons la deuxième partie du lemme par récurrence sur n.

1. Cas de base $n = 1$, on a $[B]R \equiv [g_1 \Rightarrow B_1]R$ et on pose $e_1 = e$.
   on remplace B_1 par $stateT = q_1 \Rightarrow stateT := q'_1\ [] ...\ []\ stateT = q_k \Rightarrow stateT := q'_k$.
   $[g_1 \Rightarrow (stateT = q_1 \Rightarrow stateT := q'_1$
   $[] ...\ []\ stateT = q_k \Rightarrow stateT := q'_k)]R$
   $\equiv (g_1(stateT)\ \wedge\ stateT = q_1 \Rightarrow [stateT := q'_1]R)[]$ ($R12$ Tab. 2.3)
   $...[]$
   $(g_1(stateT)\ \wedge\ stateT = q_k \Rightarrow [stateT = q'_k]R)$
   $\equiv (g_1(stateT)\ \wedge\ stateT = q_1 \Rightarrow [stateT := q'_1]R) \wedge$ ($\mathcal{WP}5$ Tab. 2.2 p. 12)
   $... \wedge$
   $(g_1(stateT)\ \wedge\ stateT = q_k \Rightarrow [stateT = q'_k]R)$

$\equiv(\forall(q,q') \in \overset{e}{\rightarrow})[g_1(stateT) \ \wedge \ stateT = q \Rightarrow [stateT := q']R]$
$\equiv(\forall q \in S) \ (\forall q' \in S)[g_1(stateT) \ \wedge \ stateT = q \ \wedge \ q \overset{e}{\rightarrow} q'$
$\Rightarrow [stateT := q']R]$
$\equiv(\forall q' \in S)[g_1(stateT) \ \wedge \ stateT \overset{e}{\rightarrow} q' \Rightarrow [stateT := q']R]$ $(\{q \backslash StateT\})$
Cette formule établit le résultat pour $n = 1$

2. Cas général. Hypothèse de récurrence : on suppose que le résultat est vrai pour tout $k < n$.
$[g_1 \Rightarrow B_1; ...; g_{n-1} \Rightarrow B_{n-1}; g_n \Rightarrow B_n]R$
$\equiv[g_1 \Rightarrow B_1; ...; g_{n-1} \Rightarrow B_{n-1}]([g_n \Rightarrow B_n]R)$ ($\mathcal{WP}$8 Tab. 2.2 p. 12)
$\equiv[g_1 \Rightarrow B_1; ...; g_{n-1} \Rightarrow B_{n-1}] \ (\forall q_{n+1} \in S)$ (Hyp. R. pour $n = 1$)
$[g_1(stateT) \ \wedge \ stateT \overset{e_n}{\rightarrow} q_{n+1} \Rightarrow [stateT := q_{n+1}]R]$
$\equiv(\forall q_2 \in S)...(\forall q_n \in S)[g_1(stateT) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_{n-1}(q_{n-1}) \wedge$ (Hyp. R. pour $n-1$)
$stateT \overset{e_1}{\rightarrow} q_2 \ \wedge \ ... \ \wedge \ q_{n-1} \overset{e_{n-1}}{\rightarrow} q_n \Rightarrow [stateT := q_n]]$
$((\forall q_{n+1} \in S)[g_1(stateT) \ \wedge \ stateT \overset{e_n}{\rightarrow} q_{n+1} \Rightarrow [stateT := q_{n+1}]R])$
$\equiv(\forall q_2 \in S)...(\forall q_n \in S)$ ($R13$ et $\mathcal{WP}1$)
$(g_1(stateT) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_{n-1}(q_{n-1})) \wedge$
$stateT \overset{e_1}{\rightarrow} q_2 \ \wedge \ ... \ \wedge \ q_{n-1} \overset{e_{n-1}}{\rightarrow} q_n \wedge$
$((\forall q_{n+1} \in S)[g_1(q_n) \ \wedge \ q_n \overset{e_n}{\rightarrow} q_{n+1} \Rightarrow [stateT := q_{n+1}]R])$
$\equiv(\forall q_2 \in S)...(\forall q_{n+1} \in S)[(g_1(stateT) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_n(q_n)) \wedge$
$stateT \overset{e_1}{\rightarrow} q_2 \ \wedge \ ... \ \wedge \ q_n \overset{e_n}{\rightarrow} q_{n+1} \Rightarrow [stateT := q_{n+1}]R]$
Cette formule exprime la propriété au rang n et démontre ainsi le lemme.

□

Lemme 4.4.2. *Soit sub une composition d'étiquettes sur un STE T. La substitution $\Phi(sub)$ est composée des appels d'opération $e_i \ \widehat{=} \ P_i|B_i$ avec $B_i = (stateT = q_{i1} \Rightarrow stateT := q'_{i1} \ [] \ ... \ [] \ stateT = q_{ik_i} \Rightarrow stateT := q'_{ik_i})$ et $P_i = stateT \in pre(e_i, T)$. La précondition P de $\Phi(sub)$ est équivalente à $P \equiv stateT \in pre(sub, T)$*

Démonstration. On considère trois cas :

1. Si $\Phi(sub) = g \Rightarrow skip$ alors $P = True$ c'est-à-dire on a $stateT \in S$
2. Si $sub = g_1 \Rightarrow e_1; ...; g_n \Rightarrow e_n$, alors on a :
$P \equiv \bigwedge_{j=1}^{n}([g_1 \Rightarrow B_1; ...; g_{j-1} \Rightarrow B_{j-1}](g_j \Rightarrow P_j))$
$\equiv \bigwedge_{j=1}^{n}((\forall q_1 \in S)...(\forall q_j \in S)$ (Lem. 4.3.1)
$[(g_1(stateT) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_{j-1}(q_{j-1})) \wedge$
$stateT = q_1 \wedge q_1 \overset{e_1}{\rightarrow} q_2 \ \wedge \ ... \ \wedge \ q_{j-1} \overset{ej-1}{\rightarrow} q_j$
$\Rightarrow [stateT := q_j](g_j \Rightarrow P_j)])$
$\equiv \bigwedge_{j=1}^{n}((\forall q_2 \in S)...(\forall q_j \in S)$ ($\mathcal{WP}$ 1)
$[(g_1(stateT) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_{j-1}(q_{j-1})) \wedge$
$stateT \overset{e_1}{\rightarrow} q_2 \ \wedge \ ... \ \wedge \ q_{j-1} \overset{ej-1}{\rightarrow} q_j$
$\Rightarrow (g_j(q_j) \Rightarrow q_j \overset{e_j}{\rightarrow})]$
$\equiv stateT \in pre(sub, T)$ (Def. 4.2.13)
3. La précondition d'une substitution $\Phi(sub)$, tel que sa forme canonique est $\Phi(sub_1) \ [] \ ... \ [] \ \Phi(sub_m)$, est définie par :
$P = \bigwedge_{i=1}^{m} P_i \equiv \bigwedge_{i=1}^{m} stateT \in pre(sub_i, T)$
$\equiv stateT \in \bigcap_{i=1}^{m}(pre(sub_i, T))$

$\equiv stateT \in pre(sub, T)$ (Def. 4.2.13)

CQFD. □

La proposition suivante met en évidence des propriétés nécessaires de deux STEs pour que le raffinement B correspondant soit satisfait.

Proposition 4.4.4. *Soient T_1 et T_2 deux STEs, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$, si T_2 raffine T_1 par rapport à R et χ, on a les propriétés suivantes :*

1. $(s_0^1, s_0^2) \in R$
2. $(\forall p \in S_1)\ (\forall q \in S_2)\ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A_2))$
$$[sub = \chi(e)\ \wedge\ (p,q) \in R\ \ \wedge\ p \xrightarrow{e}_1 \Rightarrow q \in pre(sub, T_2)]$$
3. $(\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)\ \ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A_2))\ (\forall c \in Chemin(sub, T_2))$
$[sub = \chi(e)\ \wedge\ (p,q) \in R\ \ \wedge\ origine(c) = q\ \wedge\ extremite(c) = q'\ \wedge\ p \xrightarrow{e}_1 \Rightarrow$
$$(\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p', q') \in R]]$$

Démonstration. La preuve consiste à démontrer les obligations de preuve pour l'initialisation et pour chaque opération e apparaissant dans M_a et M_c'.

Soient $I_a \equiv stateT_1 \in S_1$, $I_c \equiv stateT_2 \in S_2$ et $I_c' \equiv (stateT_1 \mapsto stateT_2) \in R$ les invariants respectifs de M_a, M_c et M_c'. Soient $Init_a$, $Init_c$ et $Init_c'$ respectivement les initialisations de M_a, M_c et M_c'. Soient $P_a|B_a$ et $P_c'|B_c'$ les définitions respectives de l'opération e dans M_a et M_c'.

Les obligations de preuve relatives au raffinement de l'initialisation et de l'opération e sont les formules ϕ_1' et ϕ_2' suivantes :

- $\phi_1' \equiv [Init_c; Init_c']\neg[Init_a]\neg I_c'$
- $\phi_2' \equiv I_a\ \wedge\ I_c\ \wedge\ I_c'\ \wedge\ P_a \Rightarrow P_c'\ \wedge\ [B_c']\neg[B_a]\neg I_c'$

Dans ϕ_1' la substitution $Init_c'$ est définie par *skip*, alors on obtient une formule équivalente à la formule ϕ_1 de la démonstration de la proposition 4.4.1. Ce qui démontre le premier item de la proposition.

Analysons la deuxième formule ϕ_2'. Nous avons $P_c' \equiv True$, I_a et I_c sont des invariants de typage, ϕ_2' se réduit à

$\phi_2' \equiv (stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow [B_c']\neg[B_a]\neg((stateT_1 \mapsto stateT_2) \in R)$

Rappelons qu'on a :

- $B_a \equiv a_1 \Rightarrow b_1\ [\!]\ ...\ [\!]\ a_n \Rightarrow b_n$
$\equiv stateT_1 = p_1 \Rightarrow stateT_1 := p_1'\ [\!]\ ...\ [\!]\ stateT_1 = p_n \Rightarrow stateT_1 := p_n'$
- $B_c' = \Phi(sub)$ s'écrit sous la forme canonique $\Phi(sub_1)\ [\!]\ ...\ [\!]\ \Phi(sub_n)$

ϕ_2' a la même forme que la formule ϕ_3 dans la démonstration de la proposition 4.4.1, la seule différence est que B_c a été remplacé par B_c'. En remplaçant B_a par sa définition, on a

$\phi_2' \equiv (stateT_1 \mapsto stateT_2) \in R \wedge P_a \Rightarrow [B_c']\neg[a_1 \Rightarrow b_1\ [\!]\ ...\ [\!]\ a_n \Rightarrow b_n]\neg((stateT_1 \mapsto stateT_2) \in R)$

En appliquant les mêmes transformations qu'à la formule ϕ_3 de la démonstration de la proposition 4.4.1 nous obtenons :

$\phi_2' \equiv (stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow [B_c']((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R])$

Nous allons montrer les items (2) et (3) de la proposition en faisant une démonstration par induction structurelle sur la substitution B_c'.

1. cas de base : on a deux cas $B_c' = g \Rightarrow skip$ et $B_c' = g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$.
   - Si $B_c' = \Phi(sub) = g \Rightarrow skip$, alors

$\phi'_2 \equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow [g \Rightarrow skip]$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \ \wedge \ g(stateT_2) \Rightarrow$ ($\mathcal{WP}$ 3 Tab. 2.2)
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R])$

En remplaçant $stateT_1$ par p et $stateT_2$ par q et en quantifiant universellement p et q, nous obtenons :

$\phi'_2 \equiv (\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R \wedge p \xrightarrow{e}_1 \ \wedge \ g(q) \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p' \wedge (p',q) \in R]]$

Comme $B'_c = g \Rightarrow skip$, $pre(B'_c, T_2) = S_2$ alors le deuxième item de la proposition est trivialement vérifié. De plus, l'ensemble des chemins de $Chemin(sub, T_2)$ sont les chemins d'origine et d'extrémité q tel que $g(q) = True$. Ceci entraîne :

$\phi'_2 \equiv (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall c \in Chemin(sub, T_2))$
$[(p,q) \in R \ \wedge \ p \xrightarrow{e}_1 \ \wedge \ origine(c) = q \ \wedge \ extremite(c) = q \Rightarrow$
$(\exists p' \in S_1)\ [p \xrightarrow{e}_1 p' \ \wedge \ (p',q) \in R]]$

Ce qui démontre le troisième item de la proposition pour $B'_c = g \Rightarrow skip$.

– Si $B'_c = \Phi(sub) = g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$. En remplaçant B'_c, on obtient :

$\phi'_2 \equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow [g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n]$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow [P | g_1 \Rightarrow B_2; ...; g_2 \Rightarrow B_n]$ (Lem. 4.3.1 p. 69)
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow P \ \wedge \ [g_1 \Rightarrow B_2; ...; g_2 \Rightarrow B_n]$ ($\mathcal{WP}$ 4 Tab. 2.2)
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow P \ \wedge$ (Lem. 4.4.1 p. 77)
$((\forall q_2 \in S_2)...(\forall q_{n+1} \in S_2)\ ((g_1(stateT_2) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_n(q_n)) \wedge$
$stateT_2 \xrightarrow{e}_1 q_2 \ \wedge ... \wedge \ q_n \xrightarrow{e}_1 q_{n+1} \Rightarrow [stateT_2 := q_{n+1}])$
$((\exists\ p' \in S_1)\ (stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto stateT_2) \in R))$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow P \ \wedge$ ($\mathcal{WP}$ 1 Tab. 2.2)
$((\forall q_2 \in S_2)...(\forall q_{n+1} \in S_2)\ [(g_1(stateT_2) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_n(q_n)) \wedge$
$stateT_2 \xrightarrow{e}_1 q_2 \ \wedge \ ... \ \wedge \ q_n \xrightarrow{e}_1 q_{n+1} \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q_{n+1}) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow stateT_2 \in pre(sub, T_2) \ \wedge$ (Lem. 4.4.2 p. 78)
$((\forall q_2 \in S_2)...(\forall q_{n+1} \in S_2)\ [(g_1(stateT_2) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_n(q_n)) \wedge$
$stateT_2 \xrightarrow{e}_1 q_2 \ \wedge \ ... \ \wedge \ q_n \xrightarrow{e}_1 q_{n+1} \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q_{n+1}) \in R])$

Cette formule peut être décomposée en deux formules. La première formule est

$$(stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow stateT_2 \in pre(sub, T_2)$$

qui est équivalente au deuxième item de la proposition à un renommage près.

La deuxième formule est la suivante.

$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow$
$((\forall q_2 \in S_2)...(\forall q_{n+1} \in S_2)\ [(g_1(stateT_2) \ \wedge \ g_2(q_2) \ \wedge \ .... \ \wedge \ g_n(q_n)) \wedge$
$stateT_2 \xrightarrow{e}_1 q_2 \ \wedge \ ... \ \wedge \ q_n \xrightarrow{e}_1 q_{n+1} \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q_{n+1}) \in R])$
$\equiv (stateT_1 \mapsto stateT_2) \in R \ \wedge \ P_a \Rightarrow$ (Def. 4.2.14 p. 64)
$(\forall q_{n+1} \in S_2)\ (\forall c)\ (c = CheminVal(sub, T_2) \wedge$
$stateT_2 = origine(c) \ \wedge \ q_{n+1} = extremite(c)) \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p' \ \wedge \ (p' \mapsto q_{n+1}) \in R])$

$\equiv(\forall c \in Chemin(sub, T_2))\ (\forall q_{n+1} \in S_2)\ ((stateT_1 \mapsto stateT_2) \in R \wedge$
$P_a\ \wedge\ (stateT_2 = origine(c)\ \wedge\ q_{n+1} = extremite(c)) \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto q_{n+1}) \in R]))$

Cette formule est équivalente au troisième item de la proposition à un renommage près.

2. Cas général. $B'_c = \Phi(sub)$ de forme canonique $\Phi(sub_1)\ [\!]\ ...\ [\!]\ \Phi(sub_n)$.

$\phi'_2 \equiv(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow$
$[\Phi(sub_1)\ [\!]\ ...\ [\!]\ \Phi(sub_n)]$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R])$
$\equiv(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow$ (Prop. 4.3.2 p. 69)
$[P_1\ \wedge\ ...\ \wedge\ P_n | B_1\ [\!]\ ...\ [\!]\ B_n]$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R])$
$\equiv(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow$ ($\mathcal{WP}$ 4,5 Tab. 2.2)
$P_1\ \wedge\ ...\ \wedge\ P_n\ \wedge$
$([B_1]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))\ \wedge$
$...\ \wedge$
$([B_n]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))$
$\equiv(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow$ (Lem. 4.4.2 p. 78)
$stateT_2 \in pre(sub, T_2)\ \wedge$
$([B_1]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))\ \wedge$
$...\ \wedge$
$([B_n]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))$

Cette formule se décompose en deux. La première permet de prouver le deuxième item de la proposition.

$$(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow stateT_2 \in pre(sub, T_2)$$

La deuxième est la suivante.

$(stateT_1 \mapsto stateT_2) \in R\ \wedge\ P_a \Rightarrow$
$([B_1]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))\ \wedge$
$...\ \wedge$
$([B_n]((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto stateT_2) \in R]))$
$\equiv\bigwedge_{i=1}^{n}((\forall c \in Chemin(sub_i, T_2))\ (\forall q_{n+1} \in S_2)\ ((stateT_1 \mapsto stateT_2) \in R \wedge$ (Hyp. d'induction)
$P_a\ \wedge\ (stateT_2 = origine(c)\ \wedge\ q_{n+1} = extremite(c)) \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto q_{n+1}) \in R])))$
$\equiv(\forall c \in Chemin(sub, T_2))\ (\forall q_{n+1} \in S_2)\ ((stateT_1 \mapsto stateT_2) \in R \wedge$ (Def. 4.2.14 p. 64)
$P_a\ \wedge\ (stateT_2 = origine(c)\ \wedge\ q_{n+1} = extremite(c)) \Rightarrow$
$((\exists\ p' \in S_1)\ [stateT_1 \xrightarrow{e}_1 p'\ \wedge\ (p' \mapsto q_{n+1}) \in R]))$

Cette formule est équivalente au troisième item de la proposition.

□

Corollaire 4.4.3. *Si T_2 raffine T_1 par rapport à R et χ, alors on peut déduire la propriété suivante :*

$(\forall p \in S_1)\ (\forall q \in S_2)\ (\exists p' \in S_2)\ \ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A_2))$
$[sub = \chi(e)\ \wedge\ (p, q) \in R\ \ \wedge\ p \xrightarrow{e}_1 p' \Rightarrow$
$(\exists q' \in S_1)\ (\exists c \in Chemin(sub, T_2))\ [origine(c) = q \wedge extremite(c) = q' \wedge (p', q') \in R]]$

Démonstration. La démonstration de ce corollaire est analogue à celle du corollaire 4.4.1, c'est pourquoi nous ne la détaillons pas. □

Le corollaire suivant est un cas particulier de la proposition 4.4.4 lorsque χ est une application de A_1 vers A_2, on parle alors de renommage d'étiquettes.

Corollaire 4.4.4. *Si T_2 raffine T_1 par rapport à une relation R et un renommage d'étiquettes χ, alors on a les propriétés suivantes :*

1. $(s_0^1, s_0^2) \in R$
2. $(\forall e \in A_1)\ (\forall e' \in A_2)\ (\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R\ \wedge\ e' = \chi(e)\ \wedge\ p \xrightarrow{e}_1] \Rightarrow q \xrightarrow{e'}_2$
3. $(\forall e \in A_1)\ (\forall e' \in A_2)\ (\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)$
   $[(p,q) \in R\ \wedge\ e' = \chi(e)\ \wedge\ q \xrightarrow{e'}_2 q'\ \wedge\ p \xrightarrow{e}_1] \Rightarrow (\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p', q') \in R]$

Relation entre raffinement B et raffinement de chemins

Nous introduisons la définition suivante de raffinement de chemins.

Définition 4.4.2 (Raffinement de chemins)**.**

Soient $c_1 = p_1 \xrightarrow{a_1}_1 p_2 ... p_m \xrightarrow{a_m}_1 p_{m+1}$ et c_2 deux chemins respectivement dans T_1 et T_2 avec $p_1 = s_0^1$, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$. Le chemin c_2 raffine le chemin c_1 par rapport à une relation R et une application χ ssi c_2 est de la forme $(q_1 \xrightarrow{b_{11}}_2 q_{21} ... q_{n1} \xrightarrow{b_{n1}}_2 q_2) ... (q_m \xrightarrow{b_{1m}}_2 q_{2m} ... q_{nm} \xrightarrow{b_{nm}}_2 q_{m+1})$ tels que :

- $q_1 = s_0^2$
- $\forall i \in [1..m]\ (q_i \xrightarrow{b_{1i}} q_{2i} ... q_{ni} \xrightarrow{b_{ni}} q_{i+1} \in Chemin(\chi(a_i), T_2))$
- $\forall i \in [1..m]\ (q_i, p_i) \in R$

(voir Figure 4.8).

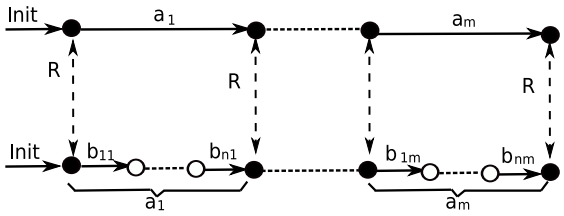


FIGURE 4.8 – Raffinement de chemins

Un chemin c_2 dans T_2, qui raffine un chemin c_1 dans T_1 par rapport à une relation R et une application χ, est un chemin dans lequel chaque transition étiquetée par a_i de c_1 a été remplacées par une séquence de transitions (un chemin) dans T_2 qui sont étiquetées par une séquence d'étiquettes appartenant à l'image de a_i par χ.

Proposition 4.4.5. *Soient T_1 et T_2 deux STEs, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$, si T_2 raffine T_1 par rapport à R et χ et si T_1 est déterministe, alors pour tout chemin c_1 dans T_1 il existe un chemin c_2 dans T_2 tel que c_2 raffine le chemin c_1 par rapport à R et χ.*

Démonstration. Si T_2 raffine T_1 par rapport à R et χ et si T_1 est déterministe, alors nous avons les propriétés suivantes :

1. $(s_0^1, s_0^2) \in R$

2. $(\forall p \in S_1)\ (\forall q \in S_2)\ (\forall p' \in S_2)\ \ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A_2))$
   $[sub = \chi(e)\ \wedge\ (p,q) \in R\ \ \wedge\ p \xrightarrow{e}_1 p' \Rightarrow$
   $(\exists q' \in S_1)\ (\exists c \in Chemin(sub, T_2))\ [origine(c) = q\ \wedge\ extremite(c) = q'\ \wedge\ (p', q') \in R]]$
   Cet item se déduit directement à partir du corollaire 4.4.3 p. 81 et que T_1 est déterministe.

Cette proposition est une conséquence directe de ces propriétés. □

Dans la traduction des STEs en B (section 4.3.1 p. 65), pour chaque étiquette nous avons défini une opération avec des préconditions, une alternative à cette traduction est de ne pas considérer les préconditions pour les opérations (cela revient à avoir des conditions vraies). Sous cette hypothèse nous avons la proposition suivante.

Proposition 4.4.6. *Soient T_1 et T_2 deux STEs, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$. Si T_2 raffine T_1 par rapport à R et χ et si dans la traduction des STEs les opérations associées aux étiquettes n'ont pas de préconditions, alors on a les propriétés suivantes :*

1. $(s_0^1, s_0^2) \in R$
2. $(\forall p \in S_1)\ (\forall q \in S_2)\ (\forall q' \in S_2)\ \ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A_2))\ (\forall c \in Chemin(sub, T_2))$
   $[sub = \chi(e)\ \wedge\ (p,q) \in R\ \wedge\ origine(c) = q\ \wedge\ extremite(c) = q' \Rightarrow$
   $(\exists p' \in S_1)\ [p \xrightarrow{e}_1 p'\ \wedge\ (p', q') \in R]]$

Démonstration. Considérons les formules ϕ_1' et ϕ_2' de la preuve de la proposition 4.4.4. ϕ_1' est inchangée et peut s'exprimer par $(s_0^1, s_0^2) \in R$, ϕ_2' peut se simplifier en tenant compte que $P_a \equiv vrai$ et $q \in pre(sub, T_2) \equiv vrai$ au deuxième item de cette proposition. □

Corollaire 4.4.5. *Soient T_1 et T_2 deux STEs, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$, si T_2 raffine T_1 par rapport à R et χ et si dans la traduction des STEs les opérations associées aux étiquettes n'ont pas de préconditions, alors pour tout chemin c_2 dans T_2 et appartenant à C_2 tel que $C_2 = \{c, c = (q_1 \xrightarrow{b_{11}}_2 q_{21} ... q_{n1} \xrightarrow{b_{n1}}_2 q_2) ... (q_m \xrightarrow{b_{1m}}_2 q_{2m} ... q_{nm} \xrightarrow{b_{nm}}_2 q_{m+1})\ \wedge q_1 = s_0^2\ \wedge \forall i \in [1..m]\ \exists a_i \in A_1 [q_i \xrightarrow{b_{1i}}_2 q_{2i} ... q_{ni} \xrightarrow{b_{ni}}_2 q_{i+1} \in Chemin(\chi(a_i), T_2)]\}$, il existe un chemin c_1 dans T_1 tel que c_2 raffine c_1.*

Démonstration. Ce corollaire est une conséquence directe de la proposition 4.4.6. □

Remarque 4.4.1. L'ensemble des chemins C_2 est l'ensemble des chemins associé à la spécification M_c' (Def. 4.4.1)

4.4.3 Relation entre un STE et plusieurs STEs

Nous présentons la construction B utilisée. Ensuite, nous définissons le raffinement B entre un STE et plusieurs STEs.

Construction B

Nous considérons les STEs $T_0 = (A_0,\ S_0,\ s_0^0,\ F_0,\ \rightarrow_0)$ et $T_i = (A_i, S_i, s_0^i, F_i, \rightarrow_i)$, $1 \leq i \leq n$. dont les alphabets A_i sont deux à deux disjoints, une relation $R \subseteq S_0 \times (S_1 \times ... \times S_k)$ et une application χ de A_0 vers $\mathcal{E}(\bigcup_{i=1}^{n} A_i)$. Nous construisons les spécifications B, M_a, M_{c1},... et M_{cn} associées respectivement à T_0, T_1,... et T_n avec $stateT_i$ comme variable. M_c' qui raffine M_a et inclut M_{c1},... et M_{cn} est obtenu par traduction de R et de χ. M_c' comporte toutes les opérations de M_a définies par l'application χ à partir des opérations des spécifications M_{c1},... ,M_{cn}. L'invariant de collage de M_c' est de la forme $(stateT \mapsto (stateT_1 \mapsto ...(stateT_{n-1} \mapsto stateT_n)...) \in R$. Nous dirons que les STEs T_1, ..., T_n raffinent T_0 par rapport à R et χ, si le raffinement M_c' est vérifié en B. Cette construction B est décrite dans la Figure 4.9.

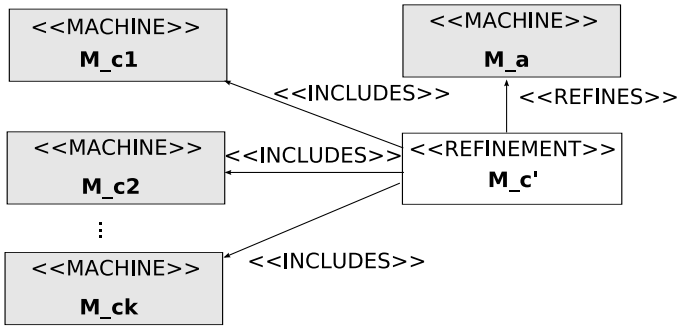


FIGURE 4.9 – Construction B

Raffinement B

Proposition 4.4.7. *Soient $T_0, ..., T_n$ $n+1$ STEs et $T = (A,\ S,\ s^0,\ F,\ \rightarrow)$ le produit libre de $T_1, ..., T_n$. Si les STEs $T_1, ..., T_n$ raffinent T_0 par rapport à une relation R et une application χ, alors les trois propriétés suivantes sont vérifiées pour toute étiquette e appartenant à A_0 et toute composition d'étiquettes $sub = \chi(e)$:*

1. $(s_0^0, (s_0^1, ..., s_0^n)) \in R$
2. $(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ (\forall e \in A_0)\ (\forall sub \in \mathcal{E}(A))$
   $[sub = \chi(e)\ \wedge\ (p, (q_1, ..., q_n)) \in R\ \ \wedge\ p \overset{e}{\rightarrow}_1 \Rightarrow (q_1, ..., q_n) \in pre(sub, T)]$
3. $(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ (\forall (q'_1, ..., q'_n) \in S)\ (\forall e \in A_0)\ (\forall sub \in \mathcal{E}(A))\ (\forall c \in Chemin(sub, T))$
   $[sub = \chi(e)\ \wedge\ (p, (q_1, ..., q_n)) \in R\ \ \wedge\ origine(c) = (q_1, ..., q_n)\ \wedge$
   $extremite(c) = (q'_1, ..., q'_n)\ \wedge\ p \overset{e}{\rightarrow}_0 \Rightarrow (\exists p' \in S_0)\ [p \overset{e}{\rightarrow}_0 p'\ \wedge\ (p', (q'_1, ..., q'_n)) \in R]]$

Démonstration. La démonstration de cette proposition est analogue à celle de la proposition 4.4.4, c'est pourquoi nous ne la détaillons pas.

□

4.4.4 Raffinement entre deux STEs gardés

Proposition 4.4.8. *Soient $T_1 = (A_1,\ G_1,\ D_1,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ et $T_2 = (A_2,\ G_2,\ D_2,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ deux STEs gardés, R une relation de S_1 vers S_2 et χ une application de A_1 vers $\mathcal{E}(A_2)$, si T_2 raffine T_1 par rapport à R et χ, on a les propriétés suivantes :*

1. $(s_0^1, s_0^2) \in R$
2. $(\forall e \in A)\ (\forall p \in S_1)\ (\forall x \in D_1)\ (\forall y \in D_2)$
   $[(p, q) \in R\ \wedge$
   $(p, x) \in pre(e, T_1)$
   $\Rightarrow (q, y) \in pre(e, T_2)]$
3. $(\forall e \in A)\ (\forall g \in G_1)\ (\forall p \in S_1)\ (\forall x \in D_1)\ (\forall y \in D_2)\ (\forall q \in S_2)\ (\forall q' \in S_2)$
   $[(p, q) \in R\ \wedge$
   $(p, x) \in pre(e, T_1)\ \wedge$
   $q \overset{[g(x)]e}{\rightarrow}_2 q'$
   $\Rightarrow (\exists p' \in S_1)(\exists g' \in G_2)[p \overset{[g'(y)]e}{\longrightarrow}_1 p'\ \wedge\ (p', q') \in R]]$

Démonstration. La démonstration de cette proposition peut se faire de manière analogue à celle de la proposition 4.4.1.

□

4.5 Travaux connexes

Plusieurs travaux se sont intéressés à définir la sémantique du raffinement B en utilisant des relations entre STEs. Citons [BJK00] et [LB03]. Cette manière de décrire le raffinement a deux principaux avantages : (i) la possibilité d'effectuer la vérification de raffinements par d'autres outils que les prouveurs de théorèmes (ii) la possibilité de combiner les techniques des prouveurs de théorèmes avec les techniques de « model-checking » et ainsi de vérifier de nouvelles propriétés qui ne sont pas supportées dans la méthode B. Comme toute approche se basant sur les techniques de « model-checking », ces deux approches sont limitées lorsqu'il a une explosion combinatoire de l'ensemble des états. Cependant, Bert *et al.* [BPS05] se sont intéressés à la description du comportement des spécifications B en se basant sur leurs spécifications abstraites antérieures, à l'aide des systèmes de transitions symboliques. Le but est de donner une vue graphique des spécifications B qui aide à mieux les comprendre et non de vérifier la relation de raffinement.

D'autres travaux [Dun03, DC05] se sont intéressés à étendre la relation de raffinement B en la complétant par l'ajout de nouvelles conditions. Ces dernières permettent de définir des relations plus élaborées entre les spécifications B. Dans [Dun03], les auteurs proposent un ensemble d'obligation de preuve pour supporter le raffinement « backward ». Dans [DC05], Dunne *et al.* ont comme objectif de supporter différents types de raffinement de processus

Les approches citées supportent les spécifications en B classique [LB03, Dun03], en B événementiel [BJK00, BPS05] ou dans les deux formalismes [BLLS08]. Notons également que dans ces différentes approches, les liens **INCLUDES** et **SEES** ne sont pas pris en considération. Dans ce qui suit, nous présentons brièvement les trois premiers travaux que nous venons de citer [BJK00, BPS05, LB08].

4.5.1 Les travaux de [BJK00]

Les auteurs s'intéressent à la relation de raffinement B en rapport avec la sémantique opérationnelle. Ils définissent une sémantique opérationnelle des spécifications B événementiel, en utilisant des systèmes de transitions interprétés. Ces derniers sont caractérisés par une fonction d'interprétation associée aux états.

Seul un type particulier de raffinement, appelé raffinement modulaire, est supporté par cette approche. Le raffinement modulaire est caractérisé par une relation de collage modulaire entre les états de deux systèmes de transitions associés aux spécifications abstraite et concrète, et par un ensemble de relations qui lient ces deux systèmes de transitions. La notion de raffinement modulaire est comparée à la relation de ready-simulation.

Le but de cette approche est de combiner les techniques de prouveur de théorème aux techniques de « model-checking » pour effectuer des vérifications. La vérification est ainsi faite en deux étapes : au niveau syntaxique par le prouveur de théorème et au niveau opérationnel par le model-checker.

4.5.2 L'approche *GeneSyst*

Dans [BC00, PS04, BPS05], Bert *et al.* présentent une méthode et un outil, appelé *GeneSyst* pour construire des systèmes de transitions symboliques (SLTS) à partir de spécifications B. Le SLTS construit donne une vue graphique et représente tous les comportements du modèle B. La construction d'un SLTS dépend de la preuve d'un ensemble de formules permettant de vérifier certaines conditions sur les évènements de la spécification considérée. Pour définir les liens entre la spécification B et le STLS correspondant, les auteurs se basent sur les traces associées à une

spécification B et aux chemins associés aux systèmes de transitions symboliques correspondant et ils montrent leur équivalence.

L'approche prend en considération la représentation du comportement d'un raffinement en introduisant la notion de hiérarchie dans les systèmes de transitions. La structure générale de la représentation des comportements d'un système abstrait, ainsi que les noms des états abstraits apparaissent dans le système de transition associé au raffinement de ce système abstrait. En effet, les états du système de transition associés au raffinement sont construits par la projection des états du système de transitions en tenant compte de l'invariant de collage. Ensuite, la relation de transition entre les états est construite en se basant sur le comportement abstrait. Les auteurs démontrent que les chemins associés à un système de transition obtenu par la projection du système de transition abstrait sont égaux aux traces générées par le raffinement. Le point fort de cette approche est qu'elle est applicable à des systèmes infinis. La vérification d'un raffinement par rapport à sa spécification abstraite, en se basant sur leurs STEs, n'est pas considérée, néanmoins la construction du système de transition raffiné se base sur les propriétés du raffinement.

4.5.3 *ProB*

ProB [LB03, LB08] est à la fois un animateur et un « model-checker » pour la méthode B. Il a été développé pour visualiser le comportement dynamique d'une machine. Ce « model-checker » peut dans le cas d'ensembles finis de petite taille, explorer de façon exhaustive l'ensemble des états pouvant être atteints par une machine B et ainsi valider cette machine. Même lorsque la recherche ne peut se faire de façon exhaustive, il est possible, grâce à un model-checker, de découvrir des contre-exemples de violation d'un invariant. Cet outil peut être vu comme un complément aux démonstrateurs de théorèmes.

ProB a été développé initialement pour le B classique et il a été étendu ensuite et porté sur la plate-forme Rodin pour supporter le B événementiel [BLLS08], *ProB* ne supporte pas l'utilisation des substitutions préconditionnées ni les liens de composition **INCLUDES** et **SEES**.

ProB supporte la vérification automatique du raffinement entre spécifications B [LB05]. Le raffinement considéré par *ProB* est différent du raffinement B, car ProB ne prend pas en compte l'invariant de collage, la vérification concerne seulement le raffinement de traces. Pour cela, une sémantique de traces est associée aux spécifications B. Rappelons qu'une machine M est un raffinement de traces d'une machine N si n'importe quelle trace de N est une trace de M (si une trace est possible dans le système concret elle est possible dans le système abstrait). Le rôle du *ProB* est d'essayer de trouver un contre-exemple pour le raffinement, c'est-à-dire de trouver une séquence d'appels d'opération qui est permise dans un raffinement mais qui n'est pas permise dans la spécification abstraite correspondante. Techniquement parlant, un algorithme a été proposé qui parcourt les états des deux systèmes, en construisant une structure de collage R entre eux. Dans le cas de succès de l'algorithme, R lie chaque état initial concret à un ensemble d'états abstraits, puis la condition de simulation est vérifiée pour chaque couple d'états liés. Le raffinement de trace entre les deux STEs est ainsi vérifié.

4.6 Conclusion

L'étude présentée dans ce chapitre diffère des travaux citées Section 4.5, puisque nous ne considérons pas les spécifications B en général, mais nous considérons des spécifications particulières obtenues par la traduction des STEs. L'objectif de notre travail est d'identifier les propriétés des relations entre STEs pour un ensemble de schémas possibles. Dans certains schémas, nous rajoutons des conditions supplémentaires afin de modéliser des relations plus fortes

entre les STEs. La clause **INCLUDES** est utilisée conjointement avec la clause **REFINES** dans les schémas étudiés, ce qui nous a permis de modéliser des relations entre STEs non considérées par les approches citées auparavant. La preuve des proposions introduites dans ce chapitre est basée sur l'analyse des obligations de preuve générées par le prouveurs de B pour chaque schéma considéré.

5

Assemblage de deux interfaces

Sommaire

5.1 Introduction 89
5.2 Description des interfaces 91
5.2.1 Diagrammes UML 91
5.2.2 Formalisation 93
5.2.3 Traduction en B 93
5.3 Compatibilité entre deux interfaces 94
5.3.1 Construction B 94
5.3.2 Raffinement et compatibilité entre interfaces 96
5.4 Adaptation 100
5.4.1 Forme générale d'un adaptateur en B 101
5.4.2 Correspondances entre opérations 102
5.4.3 Compatibilité entre l'adaptateur et les deux interfaces 111
5.5 Détection des incompatibilités, diagnostic et correction 114
5.5.1 Typologie des erreurs 114
5.5.2 Correction de l'adaptateur 115
5.5.3 Stratégies de correction 118
5.5.4 Étude de cas 118
5.6 Travaux connexes 124
5.7 Conclusion 126

5.1 Introduction

Un composant offre, par l'intermédiaire d'interfaces fournies (PI), les services ou les opérations qu'il réalise et vice versa un composant exprime, via des interfaces requises (RI), les services ou les opérations dont il a besoin. Nous pouvons connecter deux interfaces fournie et requise, appartenant à deux composants différents, si la première propose toutes les fonctionnalités permettant de réaliser les besoins de la seconde. On dit alors que les interfaces des composants considérés sont compatibles.

Dans le contexte de la réutilisation de composants préfabriqués, une interface requise d'un composant n'a que peu de chance de correspondre exactement à l'interface fournie d'un autre composant développé par une tierce partie. Une réponse logicielle à ce problème est le développement d'adaptateurs, chargés d'assurer la connexion des interfaces à assembler.

Comme nous avons pu le constater dans le chapitre 3, il existe de nombreux travaux qui se sont intéressés à l'adaptation logicielle et en particulier à l'adaptation de deux interfaces. Par ailleurs, la spécification formelle des interfaces et la preuve de leur compatibilité aux niveaux syntaxique et sémantique, en utilisant la méthode formelle B, ont été étudiées dans [CHS06, HHS06]. Ils expriment la compatibilité entre deux interfaces en terme de raffinement B. En d'autres termes, ils font la preuve que l'interface fournie correspond à un raffinement correct de l'interface requise et par conséquent, que les composants peuvent être connectés.

Notre approche d'assemblage de deux interfaces [MLS06] est basée sur les travaux de Chouali *et al.* [CHS06] que nous étendons en tenant compte en plus, du niveau protocole et en traitant le problème d'adaptation, lorsque les deux interfaces considérées ne sont pas compatibles. La suite de notre travail consiste en (i) la mise en évidence des différentes sortes d'adaptations, supportées par notre approche ; (ii) l'étude des conditions de compatibilité entre deux interfaces avec et sans adaptation ; (iii) la proposition d'une méthode d'analyse, de diagnostic et de correction de la spécification des adaptateurs.

Notre approche a les caractéristiques suivantes :

- nous nous plaçons au niveau conception et spécification des systèmes,
- nous utilisons les deux formalismes UML et B : UML pour ses diagrammes graphiques et B comme langage formel outillé,
- nous décrivons les composants par leurs interfaces requises ou fournies,
- nous prenons en compte trois niveaux d'interopérabilité : syntaxique, sémantique et protocole,
- nous prenons en considération le diagnostic et la correction des erreurs qui apparaissent lors de la construction de l'adaptateur,

Notre approche se décompose en les sept étapes suivantes :

Etape 1. Description des interfaces. L'interface requise et l'interface fournie des composants à assembler sont décrites par des diagrammes UML. La description des interfaces est transformée en B.

Etape 2. Vérification de la compatibilité. La vérification s'effectue sur les modèles B associés aux interfaces permettant d'utiliser les outils de preuves associés. Si la preuve réussit le processus s'arrête et on déduit que les deux interfaces sont compatibles, sinon on passe à l'étape 3 pour construire un adaptateur.

Etape 3. Construction de l'adaptateur. Un modèle B associé à l'adaptateur est construit à partir des règles de correspondances entre les deux interfaces. Ces règles sont des correspondances entre les modèles des données et entre les opérations des deux interfaces.

Etape 4. Vérification de la correction de l'adaptateur.

Etape 5. Détection des incompatibilités. En cas d'échec de la vérification, on détecte un ensemble d'erreurs.

Etape 6. Analyse et diagnostic d'erreurs. Elle s'effectue à partir de l'étape précédente et d'une typologie des erreurs liée à notre cadre de travail.

Etape 7. Proposition de correction. Les corrections consistent à modifier des règles de correspondances et par conséquent à corriger l'adaptateur. Si aucune adaptation n'est possible entre les interfaces alors il vaut mieux choisir une autre interface fournie.

La Figure 5.1 illustre les différentes étapes de notre approche.

Nous détaillons dans ce chapitre les sept étapes de notre approche. Nous commençons tout d'abord par donner la description des interfaces considérées dans la section 5.2 et la définition formelle de la relation de compatibilité, qui est vérifiée entre deux interfaces à l'aide de la méthode

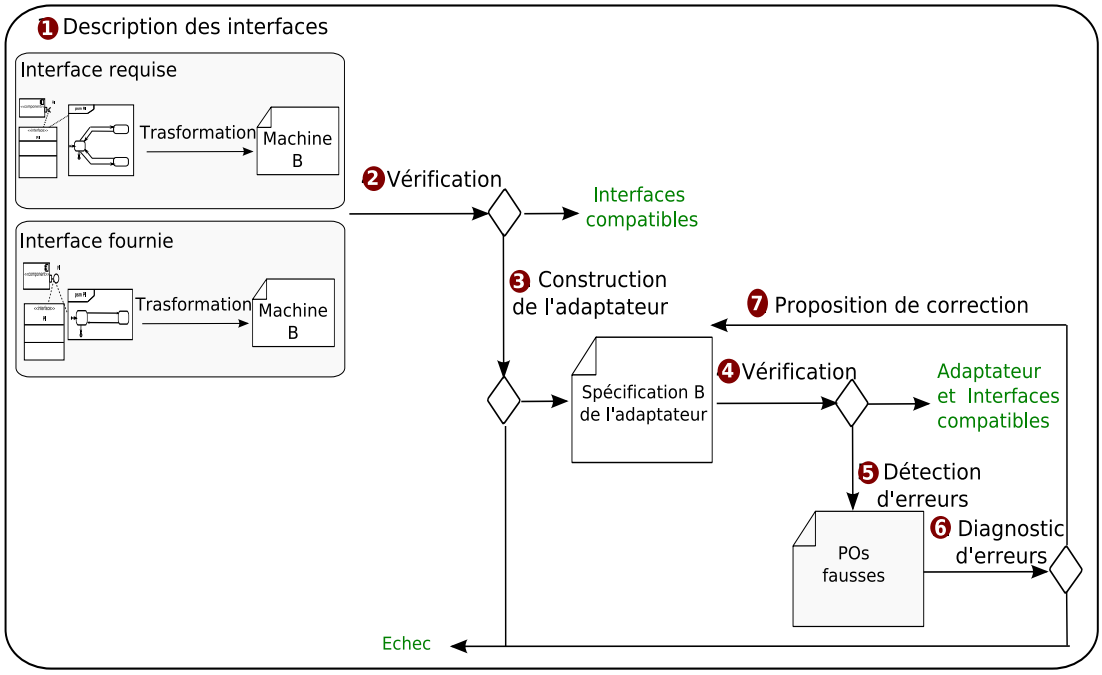


FIGURE 5.1 – Vue d'ensemble de notre approche d'assemblage de deux interfaces

B, dans la section 5.3. Nous détaillons après dans la section 5.4 la forme générale de l'adaptateur et les différents cas d'adaptation identifiés ainsi que la définition formelle de la relation de compatibilité de l'adaptateur avec les deux interfaces à connecter. Ensuite, nous introduisons dans la section 5.5, la deuxième étape pour guider la construction de l'adaptateur. Cette étape est basée sur la détection des incompatibilités, le diagnostic et la correction. Enfin, nous discutons dans la section 5.6 des travaux relatifs à l'adaptation et basés sur la méthode B. Tout au long de ce chapitre les différentes notions sont illustrées par des études de cas : des interfaces de lampes et de lecteurs de CDs.

5.2 Description des interfaces

Nous considérons une spécification des interfaces qui couvre trois aspects : syntaxique, sémantique et protocole. Nous distinguons deux types d'interface requise et fournie. La description des interfaces est donnée par des diagrammes UML. Nous formalisons ces diagrammes qui sont traduits systématiquement en une spécification B.

5.2.1 Diagrammes UML

Dans le formalisme UML, il existe deux sortes d'interfaces qui peuvent être associées à un composant : les interfaces requises et les interfaces fournies. La description de ces interfaces se fait de la même manière indépendamment de leur type.

Les aspects syntaxique, sémantique et protocole sont pris en compte dans les spécifications UML 2.0 [OMG05] de la façon suivante :

– Au niveau syntaxique, les diagrammes de classes expriment les modèles de données et le type des opérations.

- Les diagrammes d'états-transitions du protocole (PSM) sont utilisés pour décrire les protocoles. On prend comme hypothèse que les protocoles des interfaces considérées ne présentent pas de blocage.
- Au niveau sémantique, les contraintes OCL sont utilisées pour exprimer les préconditions et les postconditions des opérations ainsi que les propriétés invariantes des composants. Ces contraintes sont associées aux diagrammes de classes et aux PSM. Les pré/postconditions peuvent porter sur les paramètres des opérations considérées et sur les attributs de l'interface.

En plus de ces éléments UML, nous utilisons des diagrammes de structure composite qui décrivent totalement ou partiellement l'architecture du système en termes des composants et des interfaces à connecter.

Exemple 5.2.1 (Description de l'interface $RI\_GRlight$)**.** Nous considérons un composant C qui requiert l'interface $RI\_GRlight$, correspondant à un composant lampe. Ce composant peut avoir d'autres interfaces requises et fournies qui peuvent être traitées de manière identique à $RI\_GRlight$. La Figure 5.2 donne la description UML de cette interface.

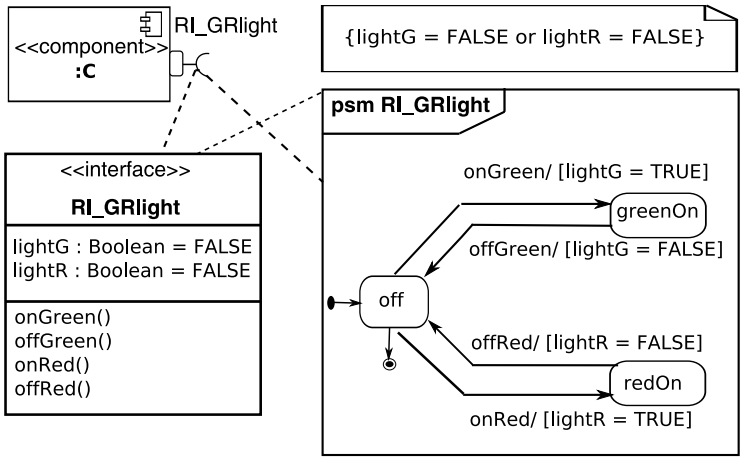


FIGURE 5.2 – Interface $RI\_GRlight$

L'interface $RI\_GRlight$ possède deux variables $lightG$ et $lightR$ de type $Boolean$. La valeur de $lightG$ (respectivement $lightR$) est $TRUE$ si la lampe verte (respectivement rouge) est allumée sinon elle est $FALSE$.

Le composant lampe possède les trois états suivants :

- off lorsqu'aucune lampe n'est allumée
- $greenOn$ lorsque la lampe verte est allumée
- $redOn$ lorsque la lampe rouge est allumée

La lampe possède les opérations $onGreen$, $offGreen$, $onRed$ et $offRed$ qui réalisent précisément les tâches suivantes :

- $onGreen$ pour allumer la lampe verte
- $offGreen$ pour éteindre la lampe verte
- $onRed$ pour allumer la lampe rouge
- $offRed$ pour éteindre la lampe rouge

Les propriétés invariantes de l'interface $RI\_GRlight$ expriment que la lampe verte et la lampe rouge ne peuvent pas être allumées simultanément.

5.2.2 Formalisation

Nous formalisons la description des interfaces, donnée par des diagrammes UML, en considérant les trois niveaux séparément, comme suit.

Définition 5.2.1. Une interface est définie par un triplet :

$$Interface = \langle Interface_{sig}, Interface_{sem}, Interface_{pro} \rangle$$

- Au niveau syntaxique : ($Interface_{sig}$) comprend les données accessibles d'un composant, c-à-d le nom et le type des attributs et les opérations munies de profil.
- Au niveau sémantique : ($Interface_{sem}$) comporte les préconditions et les postconditions des opérations sous la forme ($PreCond, PostCond$) et la description des propriétés d'un composant par un invariant.
- Au niveau protocole : ($Interface_{pro}$) décrit l'ordre dans lequel sont effectués les appels des opérations. Il est défini par un STE $Interface_{pro} = (A,\ S,\ s_0,\ F,\ \rightarrow)$ où A est un ensemble d'opérations requises ou fournies, selon l'interface considérée.

Dans tout le chapitre, nous prenons comme hypothèses que (i) la sémantique des opérations en terme de pré/postcondition et le protocole sont décrits de manière indépendante ; (ii) le STE associé à une interface requise est déterministe.

5.2.3 Traduction en B

Nous définissons un ensemble de règles de dérivation de UML vers B pour traduire les spécification des interfaces. Nous suivons les étapes suivantes pour obtenir la machine B associée à une interface donnée définie par $Interface = \langle Interface_{sig}, Interface_{sem}, Interface_{pro} \rangle$ où $Interface_{pro} = (A,\ S,\ s_0,\ F,\ \rightarrow)$. Nous commençons par traduire le STE $Interface_{pro}$, en B, en appliquant la traduction définie dans la section 4.3 du chapitre 4 p. 65. Ensuite, nous transformons les autres parties de l'interface comme suit :

- Les variables associées à l'interface sont déclarées dans la clause **VARIABLES** et typées dans la clause **INVARIANT**. Il en est de même pour la variable de contrôle associée au protocole.
- Les ensembles et les constantes de l'interface sont déclarés dans, respectivement, les clauses **SETS** et **CONSTANTS** et définis dans la clause **PROPERTIES**.
- Les propriétés invariantes associées à l'interface sont traduites en une expression B qui apparaît dans l'invariant.
- Dans la clause **ASSERTIONS**, on ajoute la propriété suivante, qui nous permet de s'assurer qu'il n'y a pas de blocage dans le STE $Interface_{pro}$:
$$(\forall q)\ (\exists e)\ [q \xrightarrow{e} \ \vee\ q \in F]$$
- Chaque opération e de l'interface est traduite en B par $e = P' \wedge P''|B'||B''$, P', B' sont obtenus par la traduction du STE $Interface_{pro}$ en B. P'' et B'' sont obtenus principalement par la traduction des pré/postconditions associées à l'opération e, en se basant sur les règles de dérivation proposées dans la thèse de Ledang [Led02] (Chapitre 7, page 111).
  - P'' est composé de la conjonction des prédicats de typage de l'opération e et de la traduction de la précondition de e en une expression B.
  - La substitution B'' est de la forme $X : (Post)$ où X est la liste des variables impliquées dans la postcondition et $Post$ est un prédicat dérivé à partir de la postcondition. Le prédicat $Post$ porte sur les variables X et sur les variables $X\$0$ (obtenues par la traduction des opérations $@pre$), on le note aussi $Post(X\$0, X)$.

Remarque 5.2.1. La traduction d'une interface UML en B, produit une machine B qui doit être prouvée. La preuve de cette machine n'est pas assurée. En particulier, l'invariant doit être préservé par les opérations de la machine. Lorsque ce n'est pas le cas, il est nécessaire de modifier la machine B.

Exemple 5.2.2 (Traduction de l'interface *RI_GRlight*)**.** La Figure 5.3 présente la machine B associée à l'interface *RI_GRlight* générée à partir de la Figure 5.2. Pour la vérification de ce modèle, 5 obligations de preuve sont générées et sont prouvées automatiquement. Cette machine a été obtenue en appliquant les règles de traduction. Pour prouver cette machine, nous avons dû renforcer les préconditions des opérations on_green et on_red par respectivement les conditions lightR = FALSE et lightG = FALSE.

```
MACHINE
   RI_GRlight
SEES
    ctx
ABSTRACT_CONSTANTS
    Light_StatesRIGR
PROPERTIES
    Light_StatesRIGR ⊆ Light_States ∧
    Light_StatesRIGR = {off, greenOn, redOn}
VARIABLES
   lightG , lightR , riGRState
INVARIANT
   lightG ∈ 𝔹 ∧
   lightR ∈ 𝔹 ∧
   riGRState ∈ Light_StatesRIGR ∧
   ((lightG = FALSE) ∨ (lightR = FALSE))
INITIALISATION
   lightG , lightR , riGRState := FALSE, FALSE, off
OPERATIONS
   onGreen =
   PRE riGRState = off ∧ lightR = FALSE THEN
      SELECT riGRState = off ∧ lightR = FALSE THEN
         lightG := TRUE || riGRState := greenOn
      END
   END;
   offGreen =
   PRE riGRState = greenOn THEN
      SELECT riGRState = greenOn THEN
         lightG := FALSE || riGRState := off
      END
   END;

   onRed =
   PRE riGRState = off ∧ lightG = FALSE THEN
      SELECT riGRState = off ∧ lightG = FALSE THEN
         lightR := TRUE || riGRState := redOn
      END
   END;

   offRed =
   PRE riGRState = redOn THEN
      SELECT riGRState = redOn THEN
         lightR := FALSE || riGRState := off
      END
   END
END
```

FIGURE 5.3 – La machine RI_GRlight

5.3 Compatibilité entre deux interfaces

Nous exprimons la compatibilité entre deux interfaces en termes d'un raffinement B. En nous basant sur la traduction précédemment définie des interfaces, nous définissons la relation de compatibilité considérée.

Dans notre approche, nous nous basons sur les travaux de Chouali *et al.* [CHS06] et nous les étendons en prenant en considération le niveau protocole. Cependant, nous nous restreignons à un modèle de données relativement simple, composé d'un ensemble d'attributs.

5.3.1 Construction B

Nous disposons d'une interface requise et d'une interface fournie. Nous traduisons l'interface requise en une machine abstraite B en utilisant la traduction définie précédemment.

Nous traduisons l'interface fournie en un raffinement B qui comporte les principales clauses B de la traduction de l'interface UML fournie en B. De plus, le raffinement vérifie les conditions suivantes :

- il raffine la spécification associée à l'interface requise,
- il comporte la relation R', définissant la relation entre les variables de contrôle (comme cela est décrit dans la section 4.3.2).
- il comporte un invariant de collage de la forme $I'_p \wedge I''_p$ tels que :
  - I'_p lie les variables du contrôle associées aux protocoles des deux interfaces (section 4.3.2 p. 66). I'_p s'exprime par le prédicat $(stateR \mapsto stateP) \in R'$.
  - I''_p permet de lier les variables associées aux deux interfaces. À chaque variable de l'interface requise correspond une variable de l'interface fournie ; cette partie de l'invariant est définie par une conjonction d'égalités, nous nommons R'' cette relation. Les variables mises en relation par R'' doivent être de types compatibles.
- il comporte l'assertion J_F dans la clause **ASSERTIONS**, qui exprime que si le STE associé au protocole de l'interface requise est dans un état final alors le STE associé au protocole de l'interface fournie est aussi dans un état final.

$$J_F \ : \ (stateR \in RIFinalStates \Rightarrow stateP \in PIFinalStates)$$

Dans la suite nous utilisons la construction B précédente afin de vérifier la compatibilité de deux interfaces. Nous dénotons aussi par RI et PI les spécifications associées aux interfaces RI et PI. La définition suivante exprime la compatibilité en termes du raffinement B.

Définition 5.3.1. Soient PI et RI deux interfaces munies des mêmes noms d'opérations. Soit R une relation qui lie les variables et les états des protocoles des deux interfaces. Les interfaces PI et RI sont dites compatibles aux niveaux syntaxique, sémantique et protocole par rapport à R si et seulement si la spécification B, PI est un raffinement B de RI (*cf.* Figure 5.4).

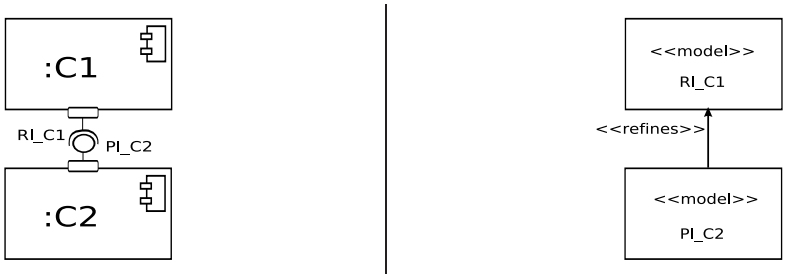


FIGURE 5.4 – Compatibilité entre C_1 et C_2

Exemple 5.3.1 (Compatibilité entre les interfaces $RI\_GRlight$ et $PI\_GRlight$)**.** Pour la réalisation de l'interface requise décrite dans l'exemple 5.2.1, on considère le composant *light* qui offre l'interface $PI\_GRlight$. Cette interface fournie a le même modèle de donnée et les mêmes opérations que l'interface $RI\_GRlight$ mais pas le même protocole (*cf.* Figure 5.5). La vérification de la compatibilité entre ces deux interfaces est prouvée en B, puisque l'interface fournie offre un protocole plus flexible que celui requis.

La transformation en B, pour l'interface $PI\_GRlight$, est faite de la même manière que pour l'interface $RI\_GRlight$. On construit le raffinement B $newPI\_GRlight$ qui est généré à partir de la machine $PI\_GRlight$. Nous lui ajoutons la clause **REFINES** RI_GRlight et un invariant de collage. Ce qui nous permet de vérifier la compatibilité entre ces deux interfaces. Pour la

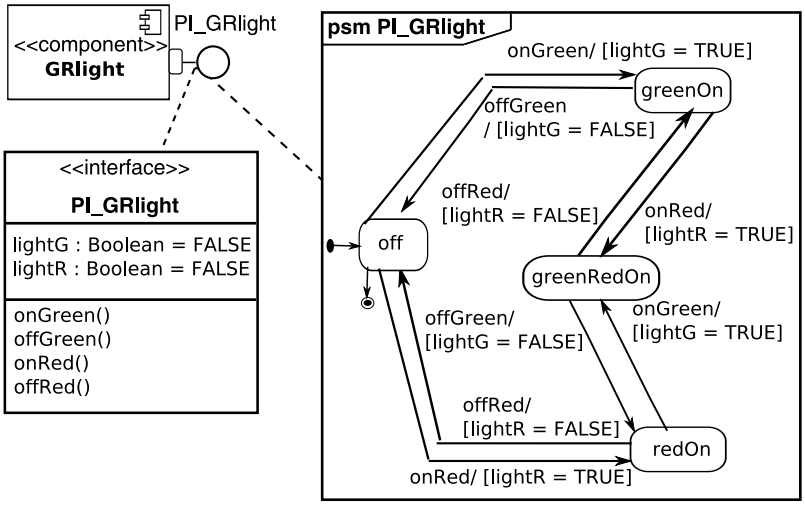


FIGURE 5.5 – Interface $PI\_GRlight$

vérification de ce raffinement, 42 obligations de preuve sont générées et sont prouvées automatiquement. La Figure 5.6 présente l'entête et l'invariant du raffinement B associé à l'interface $newPI\_GRlight$. La spécification complète est donnée dans l'annexe B Figure B.1 p. 171.

REFINEMENT
newPI_GRlight
REFINES
RI_GRlight
SEES
ctx
VARIABLES
lightG ,
lightR ,
piGRState
INVARIANT
piGRState ∈ Light_States ∧
piGRState ∈ {off, greenOn, redOn, greenRedOn}∧
lightG ∈ℙ ∧
lightR ∈ℙ ∧
piGRState = riGRState

FIGURE 5.6 – Extrait du raffinement B associé à l'interface $newPI\_GRlight$

5.3.2 Raffinement et compatibilité entre interfaces

La proposition suivante exprime que l'on peut effectuer les vérifications de la compatibilité des interfaces sur les trois niveaux séparément. Cette propriété est due à la construction B et à l'hypothèse qu'au niveau des opérations, le protocole et la sémantique sont décrits de manière indépendante. La compatibilité entre deux interfaces est exprimée comme la conjonction de la compatibilité entre les niveaux syntaxique, sémantique et protocole.

Proposition 5.3.1. *Soient* $RI = \langle RI_{sig}, RI_{sem}, RI_{pro} \rangle$ *et* $PI = \langle PI_{sig}, PI_{sem}, PI_{pro} \rangle$ *deux interfaces à connecter. Si la spécification B de PI raffine la spécification B de RI par rapport à un invariant* I_p *alors la compatibilité entre les interfaces PI et RI est définie séparément selon le*

niveau considéré : syntaxique, sémantique ou protocole et elle est définie par la conjonction de la compatibilité entre ces trois niveaux.

Démonstration. Soient RI et PI deux spécifications B associées respectivement aux interfaces RI et PI. Soient I_r et I_p les invariants respectifs dans RP et RI. I_p est composé de la conjonction suivante $I_p = I'_p \wedge I''_p \wedge I'''_p$ tel que :

- I'_p est un invariant de collage qui permet de lier les variables de contrôle.
- I''_p est un invariant de collage qui lie les données des deux interfaces.
- I'''_p permet le typage des variables de l'interface requise et la variable de contrôle et éventuellement permet de définir des propriétés invariantes sur ces variables.

La vérification syntaxique des spécifications RI et PI est faite séparément. Les vérifications de la compatibilité sémantique et protocole sont faites par les POs.

Soient e une opération, $P_r|K_r$, $P_p|K_p$ les définitions respectives de e dans RI et PI. Les préconditions P_r sont composées de la conjonction $P_r \equiv P'_r \wedge P''_r$ où P'_r définit la précondition sur la variable protocole de RI et P''_r définit la précondition sur les variables de RI. P_p est défini de la même manière.

La substitution K_r est définie par $K_r = K'_r||K''_r$ où K'_r définit la postcondition sur la variable protocole de RI et K''_r définit la postcondition sur les variables de RI. K_p est défini de la même manière.

Les POs relatives à l'opération e sont de la forme :
$\phi_1 = I_p \wedge I_r \wedge P_r \Rightarrow P_p$ et $\phi_2 = I_p \wedge I_r \wedge P_r \Rightarrow [K_p]\neg[K_r]\neg I_p$.
En considérant seulement les invariants de collage, ces formules s'écrivent :

$$\begin{aligned}
\phi'_1 = &\quad I'_p \wedge I''_p \wedge P'_r \wedge P''_r \Rightarrow P'_p \wedge P''_p \\
\equiv &\quad (I'_p \wedge P'_r \Rightarrow P'_p) \wedge (I''_p \wedge P''_r \Rightarrow P''_p) \\
\phi'_2 = &\quad I'_p \wedge I''_p \wedge P'_r \wedge P''_r \Rightarrow [K'_p||K''_p]\neg[K'_r||K''_r]\neg(I'_p \wedge I''_p) \\
\equiv &\quad (I'_p \wedge P'_r \Rightarrow [K'_p]\neg[K'_r]\neg I'_p) \wedge (I''_p \wedge P''_r \Rightarrow [K''_p]\neg[K''_r]\neg I''_p)
\end{aligned}$$

Puisque les descriptions sémantique et syntaxique d'une interface sont indépendantes, on peut décomposer les formules ϕ'_1 et ϕ'_2 comme suit.

- D'un point de vue protocole, il faut prouver :
  - $I'_p \wedge P'_r \Rightarrow P'_p$
  - $I'_p \wedge P'_r \Rightarrow [K'_p]\neg[K'_r]\neg I'_p$
- D'un point de vue sémantique, il faut prouver :
  - $I''_p \wedge P''_r \Rightarrow P''_p$
  - $I''_p \wedge P''_r \Rightarrow [K''_p]\neg[K''_r]\neg I''_p$

Ainsi, on démontre qu'on peut séparer la vérification de la compatibilité entre le niveau sémantique et protocole pour chaque opération. Pour l'initialisation, nous obtenons le même résultat en procédant de manière identique. □

Syntaxique

Il y a plusieurs définitions de la compatibilité au niveau syntaxique. Dans notre contexte, nous considérons la compatibilité de signature la plus simple qui est l'équivalence à un renommage près (appelé « exact match » dans [ZW95]). La vérification syntaxique concerne deux aspects : la vérification de la compatibilité entre les attributs des interfaces et la vérification de la compatibilité entre les opérations.

Si la spécification associée à une interface RI est un raffinement de la spécification associée à PI par rapport à un invariant I_r, alors on a les propriétés suivantes :

- Les deux spécifications B ont les mêmes noms d'opérations avec les mêmes signatures. Ceci est une contrainte syntaxique imposée entre les opérations d'une spécification B et les opérations de son raffinement.
- Les attributs de l'interface RI ne sont pas plus restrictifs que ceux de PI. Chaque attribut de RI doit avoir une contrepartie dans PI mais pas nécessairement l'inverse. C'est-à-dire que PI peut contenir des attributs qui ne sont pas nécessaires pour réaliser RI. Cette propriété est vérifiée par construction de l'invariant de collage qui est défini par un ensemble d'égalités qui associe à chaque variable de l'interface requise une variable de l'interface fournie. Les variables en relation doivent être de types compatibles. Dans le cas contraire, une erreur syntaxique est générée pour l'invariant de collage.

Sémantique

La sémantique d'une interface est exprimée par un invariant et par les préconditions et les postconditions de ses opérations.

La compatibilité au niveau sémantique entre deux interfaces consiste à vérifier des relations entre les pré/postconditions associées aux opérations considérées en se basant sur des conditions d'appariement des spécifications particulières. Si les conditions entre les pré/postconditions des opérations sont vérifiées alors l'invariant de l'interface requise est vérifié par l'interface fournie.

La proposition suivante met en évidence des propriétés nécessaires, au niveau sémantique, au raffinement d'une interface requise par une interface fournie.

Proposition 5.3.2 (Compatibilité au niveau sémantique)**.** *Si la spécification associée à une interface PI est un raffinement de la spécification associée à RI par rapport à un invariant I_p, alors les conditions d'appariement des spécifications vérifiées sont les suivantes.*

1. *Pour l'initialisation* $Init_p \Rightarrow Init_r$
2. *Pour chaque opération op définie par* $(PreCond_r, PostCond_r)$ *et* $(PreCond_p, PostCond_p)$ *respectivement dans RI et PI :*
   (a) $PreCond_r \Rightarrow PreCond_p$
   (b) $PreCond_r \wedge PostCond_p \Rightarrow PostCond_r$ *qui peut s'écrire* $PostCond_p \Rightarrow (PreCond_r \Rightarrow PostCond_r)$

Démonstration. Soient PI et RI deux interfaces, X_p et X_r leurs attributs respectifs, *op* une opération, $Init_p(X_p)$ et $Init_r(X_r)$ les initialisations respectives dans PI et RI. L'obligation de preuve associée au raffinement de l'initialisation, est :

$$Init_p(X_p) \Rightarrow \exists X_r\ [Init_r(X_r)\ \wedge\ X_p = X_r]$$
$$\equiv\quad Init_p(X) \Rightarrow Init_r(X)$$

Les obligations de preuve associées à *op* sont :

- $I''_p\ \wedge\ P''_r \Rightarrow P''_p$
- $I''_p\ \wedge\ P''_r \Rightarrow [K''_p]\neg[K''_r]\neg I''_p$

où I''_p est l'invariant de collage défini par un ensemble d'égalités entre les attributs des deux interfaces et noté $X_p = X_r$. P''_r, K''_r, P''_p et K''_p sont notés respectivement $K_r(X_r)$, $P_r(X_r)$, $P''_p(X_p)$ et $K''_p(X_p)$. Les deux POs s'écrivent alors :

(2-a) $\quad (X_p = X_r)\ \wedge\ P''_r(X_r) \Rightarrow P''_p(X_p)$
$\quad\equiv\quad P''_r(X) \Rightarrow P''_p(X))$
(2-b) $\quad (X_p = X_r)\ \wedge\ P''_r(X_r) \Rightarrow [K''_p]\neg[K''_r]\neg(X_p = X_r)$

$\equiv \quad (X_p = X_r) \ \wedge \ P''_r(X_r) \Rightarrow [X_p : Post_p]\neg[X_r : Post_r]\neg(X_p = X_r)$
$\equiv \quad (X_p = X_r) \ \wedge \ P''_r(X_r) \Rightarrow [@ \ X'_p([X_p\$0, X_p := X_p, X'_p]Post_p \Rightarrow X_p := X'_p)]$
$\neg[@ \ X'_r([X_r\$0, X_r := X_r, X'_r]Post_r \Rightarrow X_r := X'_r)]\neg(X_p = X_r)$
$\equiv \quad (X_p = X_r) \ \wedge \ P''_r(X_r) \Rightarrow \forall \ X'_p([X_p\$0, X_p := X_p, X'_p]Post_p \Rightarrow$
$[X_p := X'_p]\neg(\forall \ X'_r([X_r\$0, X_r := X_r, X'_r]Post_r \Rightarrow [X_r := X'_r]))\neg(X_p = X_r))$
$\equiv \quad \forall \ X'_p((X_p = X_r) \ \wedge \ P''_r(X_r) \ \wedge \ ([X_p\$0, X_p := X_p, X'_p]Post_p) \Rightarrow$
$\exists X'_r \ ([X_r\$0, X_r := X_r, X'_r]Post_r \ \wedge \ (X'_p = X'_r)))$
$\equiv \quad (X_p = X_r) \ \wedge \ P''_r(X_r) \ \wedge \ Post_p(X_p, X'_p) \Rightarrow Post_r(X_r, X'_r) \ \wedge \ (X'_p = X'_r)$
$\equiv \quad P''_r(X) \ \wedge \ Post_p(X, X') \Rightarrow Post_r(X, X')$
CQFD. □

Les conditions d'appariement décrites dans la proposition sont un affaiblissement des conditions d'appariement de « Plug-In Match » définies par Zaremski et Wing [ZW97]. En effet, la relation entre les préconditions est définie de la même manière. Par contre, la relation entre les postconditions est moins forte puisque dans le « Plug-In Match » elle s'écrit : $PostCond_p \Rightarrow PostCond_r$

Protocole

En terme de protocole, une opération requise est un message envoyé et une opération fournie est un message reçu. Nous disposons donc d'un protocole associé à l'interface requise qui envoie seulement des messages et un autre protocole associé à l'interface fournie qui reçoit seulement des messages. Nous prenons comme hypothèse que l'interface requise décide de l'ordre des appels des opérations fournies. Par conséquent, dès qu'une opération requise est envoyée, elle doit être acceptée par l'interface fournie. Par contre une interface fournie peut offrir une opération que l'autre interface ne peut pas envoyer.

Dès qu'on parle de vérifications d'une communication entre deux protocoles, on considère souvent la vérification d'absence de blocage dans la communication. La propriété d'absence de réceptions non spécifiées est aussi fréquemment prise en compte.

Nous introduisons la notion de collaboration entre deux STEs [YS97]. Nous donnons ensuite, les deux définitions d'absence de blocage et d'absence de réceptions non spécifiées dans une collaboration entre deux STEs, l'un est associé à un protocole requis et l'autre à un protocole fourni. Il est à noter que nous avons repris ces deux dernières définitions à partir de [YS97] et de [BZ83] et nous les avons adapté à notre contexte (protocole requis et protocole fourni). Dans la proposition 5.3.3, nous montrons que la relation de compatibilité vérifiée à l'aide du raffinement B implique bien les deux propriétés d'absence de blocage et d'absence de réceptions non spécifiées dans la communication de deux STEs.

Définition 5.3.2 (Collaboration entre deux STEs)**.** Une collaboration entre deux STEs $T_1 = (A, \ S_1, \ s_1^0, \ F_1, \ \rightarrow_1)$ et $T_2 = (A, \ S_2, \ s_2^0, \ F_2, \ \rightarrow_2)$ est une suite finie de transitions de la forme $((p_i, \ q_i), \ e_i, \ (p_{i+1}, \ q_{i+1}))$ où $0 \leq i \leq n$, $(p_i, \ q_i) \in S_1 \times S_2$, $e_i \in A$ et $(p_{i+1}, \ q_{i+1}) \in S_1 \times S_2$ telle que

- $p_0 = s_0^1 \ \wedge \ q_0 = s_0^2$
- $\forall i \in [1..n] \ (p_i \xrightarrow{e_i}_1 p_{i+1} \ \wedge \ q_i \xrightarrow{e_i}_2 q_{i+1})$

L'ensemble des collaborations entre T_1 et T_2 est noté par $Collab(T_1, \ T_2)$.

Définition 5.3.3 (Absence de blocage)**.** Soient T_1 un STE associé à un protocole requis, T_2 un STE associé à un protocole fournie et $Collab(T_1, \ T_2)$ l'ensemble des collaborations entre eux. On dit qu'il n'existe pas de blocage dans l'ensemble $Collab(T_1, \ T_2)$ si et seulement si pour tout élément de $Collab(T_1, \ T_2)$ de la forme $((p_i, \ q_i), \ e_i, \ (p_{i+1}, \ q_{i+1}))$ où $0 \leq i \leq n$, on a

- $p_{n+1} \in F_1 \Rightarrow q_{n+1} \in F_2$
- $(q_{n+1} \in F_2 \ \wedge \ \forall\, e\ q_{n+1} \overset{e}{\not\rightarrow}_2) \Rightarrow p_{n+1} \in F_1$

Définition 5.3.4 (Absence de réceptions non spécifiées)**.** Soient T_1 un STE associé à un protocole fournie, T_2 un STE associé à un protocole requis et $Collab(T_1,\ T_2)$ l'ensemble des collaborations entre eux. L'ensemble des collaborations $Collab(T_1,\ T_2)$ vérifie la propriété de réceptions non spécifiées si et seulement si pour tout α_n appartenant à $Collab(T_1,\ T_2)$ et de la forme $((p_i,\ q_i),\ e_i,\ (p_{i+1},\ q_{i+1}))$ où $0 \leq i \leq n$, la propriété suivante est vérifiée :

$$(\forall e_{n+1} \in A)\ (\forall p \in S_1)\ [p_{n+1} \xrightarrow{e_{n+1}}_1 p \Rightarrow \exists q\ q_{n+1} \xrightarrow{e_{n+1}}_2 q]$$

La proposition suivante met en évidence des propriétés nécessaires au raffinement d'une interface requise par une interface fournie, au niveau protocole.

Proposition 5.3.3 (Compatibilité au niveau protocole)**.** *Soient PI et RI deux interfaces, $RI_{pro} = (A,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ un STE déterministe associé à RI, $PI_{pro} = (A,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ un STE associé à PI, R une relation de S_1 vers S_2 et $J_F : \forall p\ \forall q\ [p \in F_1 \Rightarrow q \in F_2]$ une assertion. Si l'interface PI raffine l'interface RI avec R comme relation entre les protocoles et R vérifie J_F, alors on peut déduire que l'ensemble de collaborations entre eux ne contient ni blocage ni réceptions non spécifiées.*

Démonstration. Si PI raffine RI alors on a les propriétés suivantes :

1. $RI_{pro} \preccurlyeq_R PI_{pro}$ (d'après le corollaire 4.4.2 p. 74)
2. $(\forall p \in S_1)\ (\forall q \in S_2)\ [(p,q) \in R \ \wedge \ p \in F_1 \Rightarrow q \in F_2]$ (Cette propriété est équivalente à l'obligation de preuve associée à l'assertion J_F)

La proposition se déduit directement à partir de ces deux propriétés. □

5.4 Adaptation

En cas d'échec de la vérification de la compatibilité entre deux interfaces, l'adaptation est nécessaire pour remédier à ce problème. Dans notre approche, on considère l'assemblage de deux composants C_1 et C_2 dont les interfaces sont respectivement RI_C1 et PI_C2. L'adaptation entre ces deux interfaces consiste à construire un nouveau composant qui joue le rôle d'intermédiaire en offrant une interface fournie qui réalise RI_C1, en utilisant PI_C2 (voir Figure 5.7). La construction de l'adaptateur se fait en B à partir des spécifications B des interfaces fournie et requise, permettant la vérification de l'adaptateur.

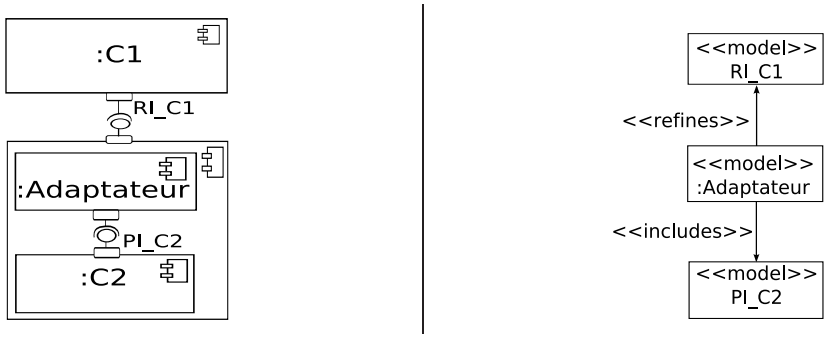


FIGURE 5.7 – Adaptateur

Un adaptateur est défini par deux types de correspondances entre les interfaces :

– Correspondances entre les données. Elles sont définies par un ensemble de règles permettant de lier les attributs des deux interfaces considérées et leur protocole. Nous n'avons pas étudié la problématique de l'adaptation des données de deux interfaces, bien que ce soit un préalable à l'adaptation. Si X_r et X_p sont respectivement les données correspondantes aux interfaces requise et fournie, nous prenons comme hypothèse que X_p peut s'adapter à X_r en utilisant une fonction F telle que $X_r = F(X_p)$.
– Correspondances entre les opérations. Dans notre contexte ces correspondances correspondent à associer à chaque opération requise un ou plusieurs opérations fournies (*cf.* Section 5.4.2).

La spécification de l'adaptateur est obtenue par la traduction des correspondances entre les données et les opérations. L'étape principale de la construction de l'adaptateur est l'identification des règles de correspondances, qui requiert l'intervention du concepteur. Compte tenu du contexte étudié (assemblage d'une interface requise avec une autre fournie), les règles de correspondances sont définies par (i) une application qui lie les variables de l'interface requise à une fonction des variables de l'interface fournie et (ii) une application qui lie chaque opération requise à une composition d'opérations fournies.

Dans ce qui suit, nous donnons la forme générale d'une spécification B associée à l'adaptateur. Ensuite, nous proposons plusieurs cas d'adaptations, en se basant sur la manière de faire les correspondances entre les deux interfaces considérées, permettant ainsi de guider le processus d'adaptation. Nous illustrons certains cas d'adaptation par des exemples.

5.4.1 Forme générale d'un adaptateur en B

```
REFINEMENT adapter
REFINES RI
INCLUDES PI
PROMOTES op
VARIABLES var
ABSTRACT_CONSTANTS R
PROPERTIES prop
/* définition de la relation R*/
INVARIANT inv
/*typage des données*/
/* Invariant de collage*/
 (stateR ↦ stateP) ∈ R
ASSERTIONS
 stateR ∈ RIFinalStates ⇒ stateP ∈ PIFinalStates
INITIALISATION init
OPERATIONS
 /*la définition des opérations requise*/
 out1 ⟵ op_r1(in1, in2) =
  PRE pre1
  THEN
    body1 /* appels des opérations de PI*/
  END
 op_r2=
  PRE pre2
  THEN
    body2 /* appels des opérations de PI*/
  END
END
```

FIGURE 5.8 – Forme générale d'une spécification adaptateur

Soient RI et PI deux machines B associées à deux interfaces non compatibles. L'assemblage de

RI et PI nécessite donc une adaptation. La Figure 5.8 montre la spécification B *adapter* réalisant une telle adaptation.

La clause **REFINES** s'applique à l'interface RI et la clause **INCLUDES** s'applique à l'interface PI. Remarquons que la construction B du raffinement *adapter* correspond au schéma étudié dans la section 4.4.1 p. 70.

Dans la clause **INVARIANT**, nous effectuons les correspondances des données entre les deux interfaces. Ces correspondances sont exprimées par des formules de la logique de premier ordre.

Dans la clause **OPERATIONS**, nous établissons des correspondances entre les opérations des deux interfaces, telles que chaque opération de RI soit définie par une composition d'opérations de PI

Dans la clause **ASSERTIONS**, nous introduisons l'assertion J_F qui exprime que si le STE associé au protocole de l'interface requise est dans un état final alors le STE associé au protocole de l'interface fournie est aussi dans un état final.

$$J_F \ : \ (stateR \in RIFinalStates \Rightarrow stateP \in PIFinalStates)$$

Nous dirons que l'adaptateur défini par les correspondances précédentes est vérifié si et seulement si le raffinement B *adapter* est prouvé en B.

5.4.2 Correspondances entre opérations

Pour toute opération de l'interface requise, il est nécessaire de trouver une opération de l'interface fournie la réalisant. Lorsque ce n'est pas possible, il faut faire appel à l'adaptation. Dans notre approche l'adaptation consiste à faire correspondre une opération requise à une composition d'opérations fournies.

Le tableau 5.1 présente un récapitulatif des différentes compositions basiques que nous avons identifiées. Chaque opération requise est exprimée par une substitution B faisant intervenir des opérations fournies. Les cas simples qui ne nécessitent pas d'adaptation, c'est-à-dire lorsque une opération requise s'exprime simplement par une opération fournie sont aussi décrits dans ce tableau. La liste établie qui n'est pas exhaustive, a été conçue en utilisant les travaux cités au chapitre 3 [YS97, SR02, PdAHSV02, BCP06, TA06, CPS08] en tenant compte de notre contexte (assemblage des interfaces fournie et requise).

Si l'on considère la partie protocole des opérations, la mise en correspondance d'une opération requise avec une substitution B composée d'appels d'opération fournie, est interprétée comme dans la section 4.4.2 p. 76 où l'on fait correspondre les étiquettes à des compositions d'étiquettes.

Dans la suite nous détaillons quelques cas de base d'adaptation, en les illustrant par des exemples. Nous donnons également des résultats concernant la sémantique (cas de séquencement et du choix d'opérations fournies)

Correspondance directe

Si une opération fournie est compatible aux niveaux syntaxique, sémantique et protocole avec une opération requise, alors on n'a pas besoin de les adapter. En B lorsque l'on a deux opérations *op*, une dans PI l'autre dans RI, compatibles sur les trois niveaux, on peut utiliser la clause **PROMOTES** qui permet de promouvoir l'opération de l'interface fournie. Les opérations promues font alors partie de l'interface de l'adaptateur.

Correspondance	Niveau d'adaptation			Construction B
	Sig.	Pro.	Sém.	
Correspondance directe				**PROMOTES** op
Renommage	×			$op\_r1 \mathrel{\widehat{=}} op\_p1$
Choix d'opérations requises	×	×	×	$op\_r1 \mathrel{\widehat{=}} op\_p1$ $op\_r2 \mathrel{\widehat{=}} op\_p1$
Réarrangement des paramètres	×			$op\_r1(xx, yy) \mathrel{\widehat{=}} op\_p1(yy, xx)$
Synthèse des paramètres	×			$op_r1 \mathrel{\widehat{=}}$ $@xx \cdot (xx = expr \Rightarrow op\_p1(xx))$
Séquencement d'opérations fournies	×	×	×	$op\_r1 \mathrel{\widehat{=}} op\_p1; op\_p2$
Choix d'opérations fournies	×	×	×	$op\_r1 \mathrel{\widehat{=}} (p_1 \Rightarrow op_p1) [] (p_2 \Rightarrow op_p2)$
Boucle d'opérations fournies	×	×	×	$op\_r1 \mathrel{\widehat{=}}$ $\mathcal{W}(n > 1, op\_p1, p, n)$
Restriction de PI	×			
Aucune correspondance d'opérations requises	×		×	Échec d'adaptation **INCLUDES** PI, newPI

TABLE 5.1 – Différents cas de correspondances entre opérations

Différences syntaxiques

Ce type d'adaptation consiste à faire correspondre une opération requise à une opération fournie syntaxiquement différente. Nous distinguons les différences syntaxiques suivantes : noms différents, avec les mêmes paramètres mais dans un ordre différent ou avec moins de paramètres.

- **Renommage.** Lorsque les opérations fournie et requise se correspondent parfaitement sauf pour les noms, nous effectuons un renommage pour rendre les opérations compatibles. Cela consiste à exprimer op_r1 en fonction de op_p1 par $op\_r1 \mathrel{\widehat{=}} op\_p1$. On peut appliquer cette adaptation à deux opérations requises op_r1 et op_r2 en utilisant la même opération fournie. Les deux correspondances sont définies par $op\_r1 \mathrel{\widehat{=}} op\_p1$ et $op\_r2 \mathrel{\widehat{=}} op\_p1$, cette adaptation est appelée choix d'opération requise.
- **Réarrangement des paramètres.** Cette adaptation consiste à effectuer une permutation des paramètres de l'opération fournie (ou requise) de telle façon que les deux opérations soient compatibles. Elle s'applique aux fonctions avec paramètres et requiert que le concepteur spécifie la permutation, le concepteur est en général guidé par le type des paramètres.
- **Synthèse des paramètres.** Il se produit parfois la nécessité de définir des paramètres de l'opération fournie, c'est au concepteur que revient cette tâche. Cette adaptation se fait en B par la définition suivante :

  $$op\_r1 \mathrel{\widehat{=}} @xx \cdot (xx = expr \Rightarrow op\_p1(xx))$$

  Notons que op_r1 est une opération qui peut avoir plusieurs arguments et que op_p1 peut posséder plus d'un argument, *expr* est une expression générale pouvant dépendre de diverses données.

Exemple 5.4.1 (Adaptation des interfaces *PI_SMlight* et *RI_GRlight*)**.** Pour la réalisation de l'interface requise décrite dans l'exemple 5.2.1, on considère le composant *SMlight*. Dans la Figure 5.9, on donne la description de son interface fournie *PI_SMlight*. Ce composant correspond à une lampe qu'on peut allumer ou éteindre. La couleur de la lampe est donnée à

chaque allumage. La spécification B de l'interface $PI\_SMlight$ est présentée dans l'annexe B Figure B.3, p. 173.

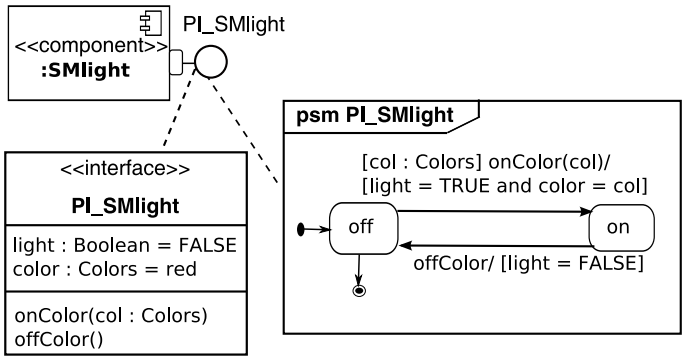


FIGURE 5.9 – Interface $PI\_SMlight$

L'adaptation requise pour réaliser les opérations on_green et on_red est décrite par la Figure 5.10.

```
onGreen =
    ANY col WHERE col = green THEN
        onColor(col)
    END;

onRed =
    ANY col WHERE col = green THEN
        onColor(col)
    END;
```

FIGURE 5.10 – Opérations onGreen() et onRed()

Cette adaptation associe à l'opération fournie onColor, une fois l'opération requise onGreen et une autre fois l'opération requise onRed. Cette adaptation illustre la synthèse du paramètre de l'opération change et aussi le choix d'opération requise entre onGreen et onRed. La spécification B complète est présentée dans la Figure B.8 p. 176

Séquencement d'opérations fournies

À une opération de l'interface requise nous faisons correspondre une séquence de deux ou de plus de deux opérations de l'interface fournie, cela est défini dans le tableau 5.1 par la relation

$$op\_r1 \mathrel{\widehat{=}} op\_p1; op\_p2$$

Au niveau sémantique, cette relation doit se traduire par des conditions d'appariements entre la précondition et la postcondition de op_r1 d'une part et la précondition et la postcondition de la séquence $op\_p1; op\_p2$ d'autre part.

La proposition suivante exprime les préconditions de la séquence $op\_p1; op\_p2$ en terme des préconditions et postconditions de op_p1 et op_p2.

Proposition 5.4.1. *Soient op_p1 et op_p2 deux opérations d'une interface PI, $(pre_{p1}, post_{p1})$ et $(pre_{p2}, post_{p2})$ les préconditions et les postconditions associées respectivement à op_p1 et op_p2. Les précondition et postcondition associées à la séquence $op\_p1; op\_p2$ sont :*

1. $pre_{p1;p2}(X_p) = pre_{p1}(X_p) \wedge (\forall\ Y)\ (post_{p1}(X_p, Y) \Rightarrow pre_{p2}(Y))$
2. $post_{p1;p2}(X_p\$0, X_p) = (\forall\ Y)\ (post_{p1}(X_p\$0, Y) \wedge post_{p2}(Y, X_p))$

Démonstration.

$op\_p1\ ;\ op\_p2$
$\equiv\ pre_{p1}(X_p)|(X_p : post_{p1}(X_p\$0, X_p)); pre_{p2}(X_p)|(X_p : post_{p1}(X_p\$0, X_p))$
$\equiv\ pre_{p1}(X_p) \wedge [X_p : post_{p1}(X_p\$0, X_p)]pre_{p2}(X_p)|$ (R2, R3 et R4 Tab. 2.3)
$(X_p : post_{p1}(X_p\$0, X_p); X_p : post_{p1}(X_p\$0, X_p))$

Nous considérons la précondition de cette substitution et nous lui appliquons les transformations suivantes jusqu'à obtenir le premier item de la proposition à un renommage près.

$pre_{p1}(X_p) \wedge [X_p : post_{p1}(X_p\$0, X_p)]pre_{p2}(X_p)$
$\equiv\ pre_{p1}(X_p) \wedge [@X'([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \Rightarrow X_p := X')]pre_{p2}(X_p)$
$\equiv\ pre_{p1}(X_p) \wedge \forall X'\ ([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \Rightarrow [X_p := X']pre_{p2}(X_p))$ ($\mathcal{WP}6$ et $\mathcal{WP}7$)
$\equiv\ pre_{p1}(X_p) \wedge \forall X'\ (post_{p1}(X_p, X') \Rightarrow pre_{p2}(X'))$ ($\mathcal{WP}1$ Tab. 2.2)

Nous partons du corps de la substitution, $op\_p1; op\_p2$, et nous lui appliquons des propriétés associées aux substitutions B.

$(X_p : post_{p1}(X_p\$0, X_p); X_p : post_{p2}(X_p\$0, X_p))$
$\equiv\ @X'([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \Rightarrow X_p := X');$
$@X''([X_p\$0, X_p := X_p, X'']post_{p2}(X_p\$0, X_p) \Rightarrow X_p := X'')$
$\equiv\ (\forall X')\ (\forall X'')\ (([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \wedge$ ($\mathcal{WP}7$, R11, R13)
$[X_p := X'; X_p\$0, X_p := X_p, X'']post_{p2}(X_p\$0, X_p)) \Rightarrow X_p := X'; X_p := X'')$
$\equiv\ (\forall X')\ (\forall X'')\ (([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \wedge$ ($\mathcal{WP}8$ et $\mathcal{WP}1$)
$[X_p\$0, X_p := X', X'']post_{p2}(X_p\$0, X_p)) \Rightarrow X_p := X'')$
$\equiv\ (\forall X')\ (\forall X'')\ (([X_p\$0 := X_p]post_{p1}(X_p\$0, X') \wedge$ ($\mathcal{WP}1$)
$[X_p := X'']post_{p2}(X', X_p)) \Rightarrow [X_p := X'']))$
$\equiv\ (\forall X'')\ (\forall X')\ (([X_p\$0, X_p := X_p, X''](post_{p1}(X_p\$0, X') \wedge post_{p2}(X', X_p)))$
$\Rightarrow X_p := X'')$
$\equiv\ X_p\ :\ ((@Y)\ (post_{p1}(X_p\$0, Y) \wedge post_{p2}(Y, X_p)))$
$\equiv\ X_p\ :\ (post_{p1;p2}(X_p\$0, X_p))$

CQFD. □

La réalisation d'une opération requise par une composition d'opérations fournies est en général à la charge du concepteur, mais celui-ci peut être guidé par le protocole ou par la sémantique. Par exemple, supposons que l'on veuille réaliser l'opération requise op_r1 par une opération fournie op_p2 et que les opérations op_r1 et op_p2 vérifient les conditions d'appariement sémantique et que le protocole de l'interface fournie exige que l'appel de l'opération op_p2 doit être précédé par l'appel de l'opération op_p1. Si les conditions d'appariement sont vérifiées entre op_r1 et $op\_p1; op\_p2$ alors on peut faire correspondre l'opération op_r1 au séquencement $op\_p1; op\_p2$.

Un autre cas peut se produire lorsqu'on échoue à trouver dans l'interface fournie une seule opération qui réalise l'opération requise op_r1 (c-à-d il n'existe aucune opération fournie qui vérifie les contraintes d'appariement avec op_r1). Par contre, la combinaison de plusieurs opérations fournies en séquence permet de réaliser l'opération requise. Pour guider la recherche des opérations fournies, on peut citer deux cas [MA03, Hem05] :

- La réalisation de l'opération requise peut être faite en deux étapes par le séquencement de deux opérations fournies telle que la sortie de la première opération fournie est l'entrée de la deuxième opération fournie.
- L'opération requise peut être décomposée en deux ou plusieurs opérations fournies indépendantes (les opérations fournies modifient des attributs différents). Il s'agit de savoir comment décomposer l'opération requise en deux opérations indépendantes, ce qui est

connu sous le nom « Splice program ». Il s'agit ensuite de chercher à réaliser les deux sous problèmes avec les opérations de l'interface fournie. La séquence des opérations fournies indépendantes peut se faire dans n'importe quel ordre.

Exemple 5.4.2 (Adaptation des interfaces PI_Mlight et $RI\_GRlight$)**.** On considère le composant *Mlight* pour la réalisation de l'interface requise décrite dans l'exemple 5.2.1. En particulier, on s'intéresse à son interface fournie PI_Mlight qui est décrite par la Figure 5.11. Ce composant correspond à une lampe qu'on peut allumer ou éteindre. La couleur de la lampe peut être modifiée lorsque la lampe est éteinte.

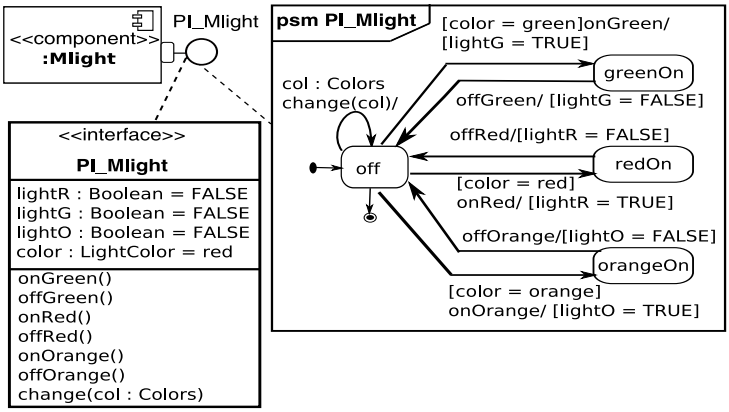


FIGURE 5.11 – Interface PI_Mlight

L'adaptation requise pour réaliser l'opération onGreen est décrite en Figure 5.12.

```
onGreen =
  BEGIN
    ANY col WHERE col = green THEN
       change(col)
    END;
    onGreen1
  END;
```

FIGURE 5.12 – Opération onGreen()

Cette adaptation associe à l'opération requise onGreen les deux opérations fournies change et onGreen1. Cette adaptation illustre la synthèse du paramètre de l'opération change combinée à la séquence avec l'opération onGreen1. La spécification B complète est présentée dans l'annexe B Figure B.7, p 176.

Choix d'opérations fournies

Cette adaptation fait correspondre une opération requise à un choix de deux ou de plus de deux opérations fournies. Contrairement au cas du séquencement, une seule opération est exécutée, de plus le choix peut être indéterministe ou déterministe. Dans le dernier cas, on doit spécifier des conditions sur les paramètres de l'opération considérée ou sur les attributs de l'interface fournie, qui déterminent quelle opération va être exécutée.

La proposition suivante détermine la préconcondition et la postcondition associées à un choix de la forme $(p_1 \Rightarrow op\_p1) \ [] \ (p_2 \Rightarrow op\_p2)$.

Proposition 5.4.2. *Soient* $(pre_{p1}, post_{p1})$ *et* $(pre_{p2}, post_{p2})$ *les pré/postconditions associées respectivement à* op_p1 *et* op_p2. *Soient* p_1 *et* p_2 *deux prédicats sur* X_p, *on les note aussi* $p_1(X_p)$ *et* $p_2(X_p)$. *Les pré/postconditions associées au choix* $(p_1 \Rightarrow op\_p1) \ [] \ (p_2 \Rightarrow op\_p2)$ *sont :*

- $pre_{p1[]p2} = (p_1(X_p) \Rightarrow pre_{p1}(X_p)) \wedge (p_2(X_p) \Rightarrow pre_{p2}(X_p))$
- $post_{p1[]p2} = (p_1(X_p\$0) \ \wedge \ post_p(X_p\$0, X_p)) \vee (p_2(X_p\$0) \ \wedge \ post_{p2}(X_p\$0, X_p))$

Démonstration.

$$
\begin{array}{ll}
 & (p_1(X_p) \Rightarrow op\_p1) \ [] \ (p_2(X_p) \Rightarrow op\_p2) \\
\equiv & (p_1(X_p) \Rightarrow pre_{p1}(X_p) | (X_p : post_{p1}(X_p\$0, X_p))) \ [] \\
 & (p_2(X_p) \Rightarrow pre_{p2}(X_p) | (X_p : post_{p2}(X_p\$0, X_p))) \\
\equiv & ((p_1(X_p) \Rightarrow pre_{p1}(X_p)) \ \wedge \ (p_2(X_p) \Rightarrow pre_{p2}(X_p))) | \quad \text{(R7, R5 et R4 Tab. 2.3)} \\
 & (p_1(X_p) \Rightarrow (X_p : post_{p1}(X_p\$0, X_p))) \ [] \\
 & (p_2(X_p) \Rightarrow (X_p : post_{p2}(X_p\$0, X_p)))
\end{array}
$$

La précondition de cette substitution est équivalente au premier item de la proposition. On applique au corps de cette substitution des transformations comme suit :

$$
\begin{array}{ll}
 & (p_1(X_p) \Rightarrow (X_p : post_{p1}(X_p\$0, X_p))) \ [] \ (p_2(X_p) \Rightarrow (X_p : post_{p2}(X_p\$0, X_p))) \\
\equiv & (p_1(X_p) \Rightarrow ((@X') \ ([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \Rightarrow X_p := X')) \ [] \\
 & (p_2(X_p) \Rightarrow (@X'') \ ([X_p\$0, X_p := X_p, X'']post_{p2}(X_p\$0, X_p) \Rightarrow [X_p := X'')) \\
\equiv & (p_1(X_p) \Rightarrow ((\forall X') \ ([X_p\$0, X_p := X_p, X']post_{p1}(X_p\$0, X_p) \Rightarrow [X_p := X'])) \ [] \quad (\mathcal{WP}7 \text{ Tab. 2.2}) \\
 & (p_2(X_p) \Rightarrow (\forall X'') \ ([X_p\$0, X_p := X_p, X'']post_{p2}(X_p\$0, X_p) \Rightarrow [X_p := X'])) \\
\equiv & (\forall X') \ ([X_p\$0, X_p := X_p, X'](p_1(X_p\$0) \wedge post_{p1}(X_p\$0, X_p)) \Rightarrow [X_p := X']) \ [] \quad \text{(R12)} \\
 & (\forall X'') \ ([X_p\$0, X_p := X_p, X''](p_2(X_p\$0) \wedge post_{p2}(X_p\$0, X_p)) \Rightarrow [X_p := X'']) \\
\equiv & (\forall X') \ ([X_p\$0, X_p := X_p, X']((p_1(X_p\$0) \wedge post_{p1}(X_p\$0, X_p)) \vee \\
 & (p_2(X_p\$0) \wedge post_{p2}(X_p\$0, X_p))) \Rightarrow [X_p := X']) \\
\equiv & X_p : (p_1(X_p\$0) \wedge post_{p1}(X_p\$0, X_p) \vee p_2(X_p\$0) \wedge post_{p2}(X_p\$0, X_p)) \\
\equiv & X_p : post_{p1[]p2}(X_p\$0, X_p)
\end{array}
$$

CQFD. □

Exemple 5.4.3 (Adaptation des interfaces $RI\_SMlight$ et $PI\_GRlight$)**.** Les descriptions des interfaces $PI\_SMlight$ et $RI\_GRlight$ sont décrites respectivement dans les exemples 5.4.1 et 5.2.1. En inversant leur rôle (l'interface requise devient l'interface fournie et inversement), nous obtenons les interfaces $RI\_SMlight$ et $PI\_GRlight$. L'adaptation requise pour réaliser l'opération onColor est décrite par la Figure 5.13.

```
onColor (col) =
  BEGIN
    SELECT col = green THEN
        onGreen
    WHEN col = red THEN
        onRed
    END
  END;
```

FIGURE 5.13 – Opération onColor()

Cette adaptation associe à l'opération requise onColor une fois l'opération fournie onGreen et une fois l'opération fournie onRed. Le choix de l'opération à prendre dépend des paramètres de l'opération onColor. Notons que si la valeur du paramètre de l'opération onColor peut prendre d'autres couleurs que le vert et le rouge, alors l'interface fournie réalise partiellement l'interface requise. On doit alors chercher un autre composant qui offre une interface fournie qui complète

l'opération requise. Nous pouvons aussi rejeter le composant de l'interface $PI\_SMlight$ et chercher un autre composant. La spécification B complète est présentée dans l'annexe B Figure B.9 p. 177.

Boucle d'opérations fournies

Cette technique d'adaptation consiste à faire correspondre une opération requise à une opération fournie que l'on peut répéter plusieurs fois. À noter que ce type d'adaptation impose que la spécification de l'adaptateur soit une implantation. C'est une contrainte purement syntaxique en B puisque la substitution **WHILE** ne peut être utilisée que dans une implantation. Ceci implique qu'on doit modifier certaines parties dans notre spécification afin de supporter les contraintes syntaxiques d'une implantation (par exemple, on doit modifier la définition de la relation R qui est définie par une constante abstraite).

Les pré/postconditions, associées à la boucle de op_r1, sont obtenues de la même manière que si l'on avait n opérations op_p1 en séquence.

Exemple 5.4.4 (Adaptation des interfaces $PI\_GRlight$ et RI_Blight)**.**

Pour la réalisation de l'interface requise RI_Blight d'un composant $Blight$, on considère le composant $GRlight$ décrit dans l'exemple 5.3.1. Dans la Figure 5.14, on donne la description de l'interface requise RI_Blight. L'interface RI_Blight requiert un composant lampe clignotant en vert (BlinkGreen(nn∈ Nat)) ou en rouge (BlinkRed(nn∈ Nat)) n fois. Initialement elle est éteinte. La spécification B de l'interface RI_Blight est présentée dans l'annexe B Figure B.4, p. 173.

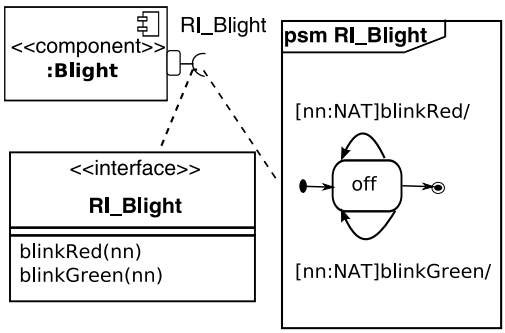


FIGURE 5.14 – Interface RI_Blight

L'adaptation requise pour réaliser l'opération BlinkRed est décrite dans la Figure 5.15.

Cette adaptation associe à l'opération requise BlinkRed une boucle de séquencement des deux opérations fournies onGreen et offGreen. En plus, il faut préciser le nombre d'itérations de la boucle et gérer la variable tt variant de boucle en la décrémentant à chaque itération. La spécification B complète est présentée dans l'annexe B Figure B.11 p. 178.

Restriction d'une interface fournie

Il peut se produire que certaines opérations de l'interface fournie ne soient pas utilisées pour réaliser l'interface requise. Dans ce cas, on peut redéfinir l'interface fournie en éliminant ces opérations, on parle alors de restriction de l'interface fournie.

```
blinkRed(nn) =
  BEGIN
    VAR tt IN
      tt := nn;
      WHILE tt > 1 DO
        onRed;              /* corps de la boucle*/
        offRed;
        tt :=tt−1
      INVARIANT
        tt ∈ 1..10 ∧          /*conditions invariantes */
        tt ≥0 ∧
        piBState = off

      VARIANT
        tt                  /*valeur entière qui décroît */
      END
    END
  END;
```

FIGURE 5.15 – Opération BlinkRed()

Aucune correspondance d'opérations requises

Pour certaines opérations requises, on peut ne pas trouver de correspondances, alors deux solutions se présentent :

- **Fusion de plusieurs interfaces fournies.** Dans le cas où on a réussi à trouver des correspondances pour certaines opérations requises et pas pour d'autres. De plus, si on juge que les opérations sans correspondance peuvent être réalisées indépendamment par un autre composant, alors on peut chercher un autre composant qui fournit les opérations manquantes. Par conséquent, plusieurs interfaces fournies vont réaliser l'interface requise.
  Exemple 5.4.5 (Adaptation des interfaces PI_Slight et $RI\_GRlight$)**.** Pour réaliser l'interface requise $RI\_GRlight$, on considère le composant *Slight* décrit dans la Figure 5.16. Ce composant correspond à une lampe qui a une seule couleur et qu'on peut allumer ou éteindre. La couleur de la lampe est donnée à l'initialisation. La spécification B de l'interface PI_Slight est présentée dans l'annexe B Figure B.5, p. 174.
  L'interface fournie par ce composant n'est pas suffisante pour réaliser l'interface requise. On ne peut réaliser qu'une partie des méthodes requises. On peut proposer d'ajouter un nouveau composant qui va fournir les fonctionnalités qui manquent. Plus précisément, on utilise deux instances du composant *Slight*. La spécification B associée à l'adaptateur est présentée dans l'annexe B Figure B.10 p. 177.
- **Échec de l'adaptation.** Dans le cas contraire, lorsque l'on n'a pas réussi à trouver de correspondances pour un sous-ensemble d'opérations requises pouvant être réalisés séparément, il est nécessaire de remplacer le composant offrant l'interface fournie par un autre composant.
  Exemple 5.4.6 (Échec d'adaptation des interfaces PI_Tlight et $RI\_GRlight$)**.** On considère le composant *Tlight* décrit dans la Figure 5.16. Ce composant correspond à une lampe qu'on peut allumer ou éteindre avec trois couleurs (oranger, vert et rouge) dans un ordre particulier. La spécification B de l'interface PI_Tlight est présentée dans l'annexe B Figure B.6 p. 175.
  L'interface fournie par ce composant ne nous permet pas de réaliser même partiellement l'interface requise $RI\_GRlight$, alors l'adaptation échoue.

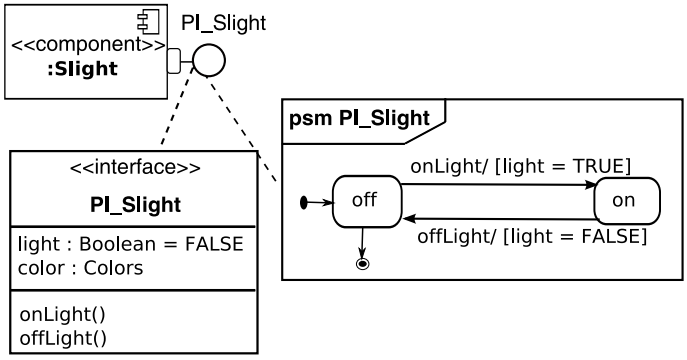


FIGURE 5.16 – Interface PI_Slight

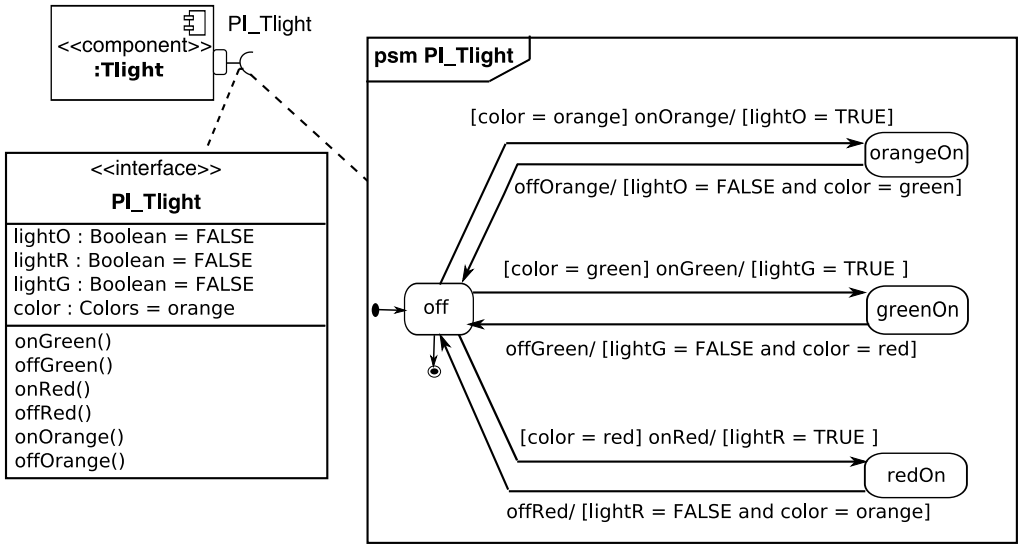


FIGURE 5.17 – Interface PI_Tlight

Combinaisons de plusieurs correspondances

Il est possible de combiner les différentes adaptations résumées dans le tableau 5.1 p. 103. En effet, nous considérons des correspondances entre une opération requise à une composition d'opérations fournies. Nous définissons les compositions d'opérations de la même manière que les compositions d'étiquettes présentées dans la définition 4.2.12 p. 63 sauf que les étiquettes considérées sont des opérations qui décrivent en même temps un ensemble de transitions dans le STE IP_{pro} et pré/postconditions associées à l'opération correspondante et P est l'ensemble de prédicats unaires de profil $(S \cup X_p) \rightarrow Bool$. On note $\mathcal{E}(A_p)$ l'ensemble des compositions d'opérations sur l'ensemble A_p. Ces compositions d'opérations sont traduites en B de la même manière que les compositions d'étiquettes (Sec. 4.3.3 p. 67).

Ainsi, nous définissons les correspondances entre opérations par une application χ de A_r vers $\mathcal{E}(A_p)$ qui associe à chaque opération de l'interface requise une composition d'opérations fournies.

La proposition suivante détermine la préconcondition et la postcondition associées à une composition d'opérations.

Proposition 5.4.3. *Soit sub une composition d'opérations. Les pré/postconditions associées à sub sont :*

1. *Si* $sub \equiv g_1 \Rightarrow e_1; g_2 \Rightarrow e_2; ...; g_n \Rightarrow e_n$ *où* $(pre_i, post_i)$ *sont les pré/postconditions associées à* e_i*, les pré/postconditions associées à sub sont :*
   - *(a)* $pre_{sub}(X_p) \equiv (\forall X_2)...(\forall X_n) \bigwedge_{j=1}^{n}(p_1(X_p) \wedge p_2(X_2) \wedge .... \wedge p_j(X_j) \wedge Post_1(X_p, X_2) \wedge ... \wedge Post_j(X_{j-1}, X_j) \Rightarrow pre_j(X_j))$
   - *(b)* $post_{sub}(X_p\$0, X_p) \equiv (\forall X_2)...(\forall X_n) (p_1(X_p\$0) \wedge p_2(X_2) \wedge .... \wedge p_n(X_n)) \wedge Post_1(X_p\$0, X_2) \wedge ... \wedge Post_n(X_n, X_p)$
2. *Si sub est sous forme canonique* $sub_1 [] ... [] sub_n$ *où* $(pre_i, post_i)$ *sont les pré/postconditions associées à* sub_i*, les pré/postconditions associées à sub sont définies comme suit :*
   - *(a)* $pre_{sub}(X_p) \equiv \bigwedge_{j=1}^{n} pre_i(X_p)$
   - *(b)* $post_{sub}(X_p\$0, X_p) \equiv \bigvee_{j=1}^{n} post_i(X_p\$0, X_p)$

Démonstration. Pour démonter cette proposition, nous nous basons sur les propositions 5.4.1 p. 104 et 5.4.2 p. 107. □

5.4.3 Compatibilité entre l'adaptateur et les deux interfaces

Dans cette section nous étudions l'utilisation des liens de modularité **REFINES** et **INCLUDES** pour vérifier la compatibilité aux niveaux syntaxique, sémantique et protocole entre un adaptateur et les deux interfaces qu'il relie.

Définition 5.4.1. Soient RI et PI deux interfaces respectivement requise et fournie. Soit *Adapt* un adaptateur qui relie les deux interfaces. L'interface RI est compatible avec l'interface PI par rapport à *Adapt*, si la spécification de *Adapt* raffine la spécification B associée à RI et inclut la spécification B associée à PI (*cf.* Figure 5.7).

On introduit la proposition suivante dans laquelle on définit la compatibilité entre l'adaptateur et les deux interfaces comme la conjonction de la compatibilité entre les niveaux syntaxique, sémantique et protocole.

Proposition 5.4.4. *Si la spécification B de Adapt raffine la spécification B de RI et inclut la spécification de PI alors la compatibilité entre les interfaces PI, RI et Adapt est définie séparément selon le niveau considéré syntaxique, sémantique ou protocole et elle est définie par la conjonction de la compatibilité entre ces trois niveaux.*

Démonstration. La démonstration de cette proposition est faite de la même manière que la proposition 5.3.1 p. 96. □

Syntaxique

Dans notre contexte, la compatibilité syntaxique de l'adaptateur avec les deux interfaces considérées est définie par la vérification syntaxique associée aux liens de modularité **INCLUDES** et **REFINES** et aux hypothèses syntaxiques considérées.

La spécification de *Adapt* inclut la spécification de PI, alors on a les propriétés syntaxiques suivantes :

1. Les variables de l'interface fournie peuvent être lues dans l'adaptateur, alors elles doivent être utilisées correctement.
2. L'adaptateur n'a le droit de modifier l'état de l'interface fournie qu'à travers ses opérations.
3. Dès qu'une opération de l'interface fournie apparaît dans la spécification de l'adaptateur, ses paramètres et leur type doivent être corrects.

La spécification de *Adapt* est un raffinement de la spécification RI, on a alors les propriétés syntaxiques suivantes :

- Les spécifications B *Adapt* et RI ont les même noms d'opérations avec les même signatures.
- L'invariant de collage lie les attributs de RI et les attributs de PI en respectant les types de chaque attribut. En plus, on a pris comme hypothèse que chaque attribut de RI doit avoir une contre-partie dans PI en fonction des attributs de PI mais pas nécessairement l'inverse. L'invariant de collage pour les attributs est défini par un ensemble d'égalités qui associent à chaque variable de l'interface requise une fonction de variable de l'interface fournie qui doivent être de types compatibles : $X_r = F(X_p)$.

Protocole

Dans cette section nous étendons la relation de compatibilités entre deux protocoles (Sec. 5.3.2 p. 96) pour considérer une relation de compatibilité en tenant compte de l'adaptateur (ensemble des correspondances).

Nous donnons les définitions de collaboration entre protocoles, à l'aide d'un adaptateur, d'absence de blocage et de réceptions non spécifiées dans une collaboration. Nous montrons, ensuite, que la relation de compatibilité qu'on vérifie à l'aide du raffinement B implique la relation de compatibilité qu'on a définie.

On donne les définitions suivantes.

Définition 5.4.2 (Collaboration entre deux STEs)**.** Soient T_1 et T_2 deux STEs et χ une application de A_1 vers $\mathcal{E}(A_2)$. Une collaboration entre T_1 et T_2, par rapport à la relation de correspondance χ, est une suite finie de forme $((p_i,\ q_i),\ e_i,\ (p_{i+1},\ q_{i+1}))$ où $0 \leq i \leq n$, $e_i \in A_1$, $(p_i, q_i) \in S_1 \times S_2$ et $(p_{i+1},\ q_{i+1}) \in S_1 \times S_2$ telle que :

- $p_0 = s_0^1 \ \wedge \ q_0 = s_0^2$
- $\forall i \in [1..n]\ (p_i \xrightarrow{e_i}_1 p_{i+1})$
- $\forall i \in [1..n](\exists q_{2i})... \ (q_i \xrightarrow{e_{1i}} q_{2i}...q_{mi} \xrightarrow{e_{mi}} q_{i+1} \in Chemin(\chi(e_i), T_2))$.

On note par $Collab(T_1,\ T_2,\ \chi)$ l'ensemble des collaborations entre T_1 et T_2 par rapport χ.

Définition 5.4.3 (Absence de blocage)**.** Soient $T_1 = (A_1,\ S_1,\ s_1^0,\ F_1,\ \rightarrow_1)$ un STE associé à un protocole requis, $T_2 = (A_2,\ S_2,\ s_2^0,\ F_2,\ \rightarrow_2)$ un STE associé à un protocole fourni, $\mathcal{A}$ un adaptateur défini par une application χ et $Collab(T_1,\ T_2,\ \chi)$ l'ensemble des collaborations entre eux. On dit qu'il n'existe pas de blocage dans l'ensemble de collaboration $Collab(T_1,\ T_2,\ \chi)$ si et seulement si pour tout α_n, appartenant $Collab(T_1,\ T_2,\ \chi)$ et de la forme $((p_i,\ q_i),\ e_i,\ (p_{i+1},\ q_{i+1}))$ où $0 \leq i \leq n$, on a

- $p_{n+1} \in F_1 \Rightarrow q_{n+1} \in F_2$
- $(q_{n+1} \in F_2\ \wedge\ \forall\ e\ (q_{n+1} \notin pre(\chi(e), T_2))) \Rightarrow p_{n+1} \in F_1$

Définition 5.4.4 (Absence de réceptions non spécifiées)**.** Soient T_1 un STE associé à un protocole fourni, T_2 un STE associé à un protocole requis, $\mathcal{A}$ un adaptateur défini par une application χ et $Collab(T_1,\ T_2,\ \mathcal{A})$ l'ensemble des collaborations entre eux. L'ensemble des collaborations $Collab(T_1,\ T_2,,\ \chi)$ vérifie la propriété d'absence de réceptions non spécifiées si et seulement si pour tout α_n appartenant $Collab(T_1,\ T_2,\ \mathcal{A})$ et de la forme $((p_i,\ q_i),\ e_i,\ (p_{i+1},\ q_{i+1}))$ où $0 \leq i \leq n$, la propriété suivante est vérifiée :

$(\forall e_{n+1} \in A_1)\ (\forall p \in S_1)\ [p_{n+1} \xrightarrow{e_{n+1}}_1 p \Rightarrow \exists c\ (c \in Chemin(\chi(e_{n+1}), T_2))\ origine(c) = q_{n+1}]$

Proposition 5.4.5 (Compatibilité au niveau protocole)**.** *Soient PI et RI deux interfaces, $RI_{pro} = (A_1,\ S_1,\ s_0^1,\ F_1,\ \rightarrow_1)$ un STE déterministe associé à RI, $PI_{pro} = (A_2,\ S_2,\ s_0^2,\ F_2,\ \rightarrow_2)$ un STE associé à PI et $\mathcal{A}$ un adaptateur entre ces deux interfaces. Si la spécification B de $\mathcal{A}$ raffine l'interface RI et inclut l'interface PI, alors on peut déduire que la communication entre les deux interfaces à l'aide de l'adaptateur $\mathcal{A}$ ne contient ni blocage ni réceptions non spécifiées.*

Démonstration. La spécification de $\mathcal{A}$ est basée sur une relation R de S_1 vers S_2, une assertion $J_F : \forall p\ \forall q\ [p \in F_1 \Rightarrow q \in F_2]$ et une application χ de A_1 vers $\mathcal{E}(A_2)$. Si la spécification de $\mathcal{A}$ raffine l'interface RI et inclut l'interface PI, alors on a les propriétés suivantes :

1. Pour tout chemin c_1 dans T_1 il existe un chemin c_2 dans T_2 tel que c_2 raffine le chemin c_1 par rapport à R et χ. (d'après la proposition 4.4.5 p. 82)
2. $(\forall p \in S_1)\ (\forall q \in S_2)\ [(p, q) \in R\ \wedge\ p \in F_1 \Rightarrow q \in F_2]$ (Cette propriété est équivalente à l'obligation de preuve associée à l'assertion J_F)

La proposition se déduit directement à partir de ces propriétés. □

Sémantique

La compatibilité, au niveau sémantique, entre l'adaptateur et les deux interfaces RI et PI, se base sur la vérification des conditions d'appariement entre les pré/postconditions associées à une opération de l'interface requise et les pré/postconditions associées à une composition d'opérations de l'interface fournie. Ces conditions d'appariement sont définies de la même manière qu'entre les pré/postconditions de deux opérations (Prop. 5.3.2 p. 98).

Proposition 5.4.6 (Compatibilité au niveau sémantique)**.** *Soient PI et RI deux interfaces, $\mathcal{A}$ un adaptateur entre ces deux interfaces et F la fonction entre les attributs des deux interfaces. Soient $Init_p$ et $Init_r$ les initialisations respectivement dans PI et RI, op une opération dans A_1, $(PreCond_r, PostCond_r)$ les pré/postconditions associées à op dans l'interface requise, sub une composition d'étiquettes telle que $sub = \chi(op)$ et $(PreCond_{sub}, PostCond_{sub})$ les pré/postconditions associées à sub. Si la spécification B de $\mathcal{A}$ raffine l'interface RI et inclut l'interface PI, alors les conditions d'appariement des spécifications vérifiées sont les suivantes.*

1. *Pour l'initialisation* $Init_p(X) \Rightarrow Init_r(F(X))$
2. *Pour chaque opération op*
   (a) $PreCond_r(X) \Rightarrow PreCond_{sub}(F(X))$
   (b) $PreCond_r(X) \ \wedge \ PostCond_{sub}(F(X)) \Rightarrow PostCond_r(X)$

Démonstration. Pour démonter cette proposition, nous procédons de la même manière que dans la proposition 5.3.2 p. 98, en tenant compte de la proposition 5.4.3 p. 111. □

5.5 Détection des incompatibilités, diagnostic et correction

Dans cette section, notre objectif est d'apporter des aides à la correction de la spécification de l'adaptateur en cas d'échec de la vérification. La vérification de l'adaptateur dépend des spécifications des deux interfaces à connecter, cependant, nous prenons comme hypothèse que lors de la correction nous ne modifions pas les spécifications associées aux interfaces requise et fournie.

Nous utilisons les résultats de la section 2.6 p. 24 concernant la détection, l'analyse et la correction des erreurs dans une spécification B, en particulier, les résultats associés à un raffinement ou à une implantation, puisque la spécification associée à l'adaptateur est un raffinement ou une implantation.

La détection des erreurs dans la spécification de l'adaptateur se fait en utilisant les POs fausses exactement de la même manière que dans la section 2.6.1 p. 24. Nous disposons des informations supplémentaires sur la forme générale des spécifications (*e.g.* la définition de l'invariant est basée sur une relation R et une fonction F, les opérations sont définies par des appels d'opération) et leur sémantique (*e.g.* les opérations décrivent des pré/postconditions et un ensemble de chemins dans l'interface fournie). Conséquemment, nous avons plus d'informations dans l'étape analyse et diagnostic. Ces informations supplémentaires vont nous permettre d'affiner les corrections proposées dans la section 2.6 p. 24.

Dans ce qui suit, la section 5.5.1 présente une typologie des erreurs spécifiques à notre contexte et la façon de les identifier. La section 5.5.2 présente un ensemble de corrections proposées pour chaque type d'erreur. Ces résultats sont illustrés par l'étude de cas d'un lecteur de CD dans la section 5.5.4 .

5.5.1 Typologie des erreurs

Nous présentons une typologie des erreurs, en nous basant sur le processus de vérification de l'adaptateur et en particulier sur les propriétés qui assurent sa correction et qui peuvent être non vérifiées. En effet, les types d'erreurs présentées ne sont que les types d'incompatibilités possibles entre les interfaces à connecter et l'adaptateur. Également, nous présentons comment chaque cas d'erreur est identifié.

Dans cette typologie, on distingue deux types d'erreurs : des incompatibilités de protocole et des incompatibilités de sémantique. Les erreurs au niveau protocole sont identifiées par les POs fausses dont le but contient des variables de protocole de l'interface fournie ou de l'interface requise ou même seulement des valeurs associées à ces variables. De la même manière, les erreurs au niveau sémantique sont identifiées.

1. Incompatibilité de l'initialisation. Ce type d'erreur est détecté par les POs associées à l'initialisation, en particulier, aux POs de type *po*7.

(a) Incompatibilités au niveau protocole. L'état initial du protocole de l'interface fournie n'est pas lié à l'état initial du protocole de l'interface requise. On note ce type d'erreur par Err_pi.

(b) Incompatibilités au niveau sémantique. Les valeurs des attributs de l'interface requise à l'initialisation ne sont pas liées ou correctement liées aux valeurs initiales des attributs de l'interface fournie. On note ce type d'erreur par Err_si.

2. Opération fournie non déclenchable. Ce type d'erreur peut se produire dans une opération de l'adaptateur. Cette erreur peut être détectée par des POs fausses de types $po8$ et $po11$.

   (a) Incompatibilités au niveau protocole. L'opération fournie est appelée à partir d'un état de protocole où elle ne peut pas être déclenchée. On note ce type d'erreur par Err_po. Ce type d'erreur est appelé réceptions non spécifiées. On se trouve dans le cas où l'interface requise envoie un message qui ne peut pas être reçu par l'interface fournie de l'adaptateur.

   (b) Incompatibilités au niveau sémantique. L'opération fournie est appelée avec des données ou des paramètres ne vérifiant pas sa précondition. C'est-à-dire certaines valeurs de données qu'on veut traiter par l'interface requise ne sont pas considérées par l'opération fournie. On note ce type d'erreur par Err_so.

3. État final du protocole fourni non respecté. Ce type d'erreur est détecté par les POs fausses associées aux assertions de type $po2$, en particulier, la partie de l'assertion qui pose problème est $\mathsf{stateR} \in \mathsf{F\_r} \Rightarrow \mathsf{stateP} \in \mathsf{F\_p}$.
   Si le protocole de l'interface requise est dans un état final alors le protocole de l'interface fournie doit l'être aussi. Dans le cas contraire, on a un état de blocage puisque l'interface fournie reste en attente d'un ou de plusieurs messages de l'interface requise. On note ce type d'erreur par Err_pf.

4. Les incompatibilités générées par la non-conformité entre l'interface requise et l'interface fournie par l'adaptateur. Ce type d'erreur est détecté par les POs fausses associées à des opérations. Elles sont de type $po10$. En particulier, ces POs sont associées à la vérification de la partie de l'invariant $(stateR, stateP) \in R$ pour les erreurs de protocole et à la partie de l'invariant $X_r = F(X_p)$ pour les erreurs de sémantique.

   (a) Incompatibilités au niveau protocole. La réalisation d'une opération requise par une opération fournie mène à un état du protocole de l'interface fournie qui n'est pas conforme avec l'interface requise. On note ce type d'erreur par Err_po'.

   (b) Incompatibilités au niveau sémantique. La réalisation d'une opération requise par une opération fournie mène à des valeurs des attributs de l'interface fournie qui ne sont pas conformes aux valeurs des attributs de l'interface requise. On note ce type d'erreur par Err_so'.

À noter que l'on peut se retrouver simultanément avec des incompatibilités de protocole et de sémantique. Dans la section suivante, nous étudions comment chaque type d'erreur peut être corrigé.

5.5.2 Correction de l'adaptateur

La spécification d' un adaptateur est définie par un ensemble de correspondances. De ce fait, corriger une spécification d'adaptateur revient à modifier ces correspondances. Nous distinguons deux types de corrections : (i) nous modifions les correspondances entre les données. Par exemple,

nous modifions l'invariant de collage et/ou nous modifions les ensembles et les constantes sur lesquels se base l'invariant de collage. (ii) nous modifions les correspondances entre les opérations. Ces dernières modifications sont faites dans les opérations de l'adaptateurs en supprimant des appels d'opération, ou en ajoutant des appels d'opération, *etc.*

Le choix d'une correction est guidé par le type d'erreur identifié, par les valeurs des attributs et de l'état du protocole des deux interfaces juste avant l'erreur ainsi que par les spécifications des interfaces. Pour chaque cas d'erreur défini précédemment, nous proposons un ensemble de corrections possibles. Nous reprenons certaines corrections proposées dans la section 2.6.3 p. 26 et nous proposons des corrections plus spécifiques et plus élaborées. Certaines corrections de la section 2.6.3 sont ignorées puisqu'elles ne sont pas significatives dans notre contexte.

Les corrections proposées doivent vérifier les contraintes de protocole et de sémantique. Elles sont basées sur :

- Les STEs associées aux deux interfaces considérées et en particulier sur la recherche des chemins dans ces deux STEs.
- Les pré/postconditions associées aux opérations des deux interfaces considérées.

Dans ce qui suit, on considère deux interfaces :

- Une interface requise $RI = \langle RI_{sig}, RI_{sem}, RI_{pro} \rangle$ avec $RI_{pro} = (A_r,\ S_r,\ s_0,\ F_r,\ \rightarrow_r)$.
- Une interface fournie $PI = \langle PI_{sig}, PI_{sem}, PI_{pro} \rangle$ avec $PI_{pro} = (A_p,\ S_p,\ s'_0,\ F_p,\ \rightarrow_p)$.

On considère, RI la spécification B associée à l'interface RI avec stateR la variable de contrôle et X_r les attributs de RI. On considère PI la spécification B associée à l'interface PI avec stateP la variable de contrôle utilisée et X_p ses attributs. La spécification *Adapt* est un adaptateur qui assure la connexion entre ces deux interfaces. I_{adapt} est l'invariant de la spécification *Adapt*. R est la relation qui lie les variables de contrôle des deux interfaces et F est la fonction qui lie les attributs des deux interfaces. B_{adapt} comprend les propriétés sur les constantes et les ensembles déclarés dans *Adapt*.

Pour chaque type d'erreur qui peut apparaître dans la spécification de *Adapt*, nous proposons des corrections comme suit.

Incompatibilité de l'initialisation

Les différentes corrections possibles pour l'initialisation qui sont applicables dans ce type d'erreur (aux POs de type *po*7), sont : Modifier I_{adapt} et Modifier B_{adapt}.

Nous affinons ces corrections comme suit.

1. **Modifier B_{adapt}.**
   1.1 *Modifier R.* Si l'erreur est de type Err_*pi*, alors on propose d'ajouter dans la définition de la relation R le couple (s_0, s'_0), s'il n'existe pas et de supprimer tous les couples (s_0, s) avec $s \neq s'_0$.
   1.2 *Modifier la relation entre les attributs.* Si l'erreur concerne le niveau sémantique (Err_*si*), on propose de modifier la définition de la fonction F.
2. **Modifier I_{adapt}.** Dans le cas où l'erreur est de type Err_*si*, on propose de modifier I_{adapt} en modifiant la fonction F.

Opération fournie non déclenchable

On considère, par exemple, une opération $op = K_{adapt}$ de l'adaptateur non vérifiée à cause d'une opération fournie $op_i = P_i|K_i$ qui est appelée dans la définition de op et qui n'est pas déclenchable. Les différentes corrections possibles (associées aux POs fausses de type *po*8 et *po*11), dans ce cas, sont : Modifier I_{adapt}, Modifier B_{adapt} et Modifier K_{adapt}.

Nous détaillons chacune de ces corrections.

1. **Modifier K_{adapt}.**
   1.1 *Insérer des opérations dans K_{adapt}.* On propose de préfixer l'appel de op_i par une composition d'opérations. On considère les deux substitutions sub_1 et sub_2 qui correspondent respectivement à la substitution qui précède et à celle qui suit op_i dans K_{adapt}. On propose de modifier K_{adapt} par $sub_1; sub; op_i; sub_2$ où sub est une composition d'opérations fournies définie par $P|K$ vérifiant les propriétés suivantes :
   - $I_{adapt} \wedge P_r \Rightarrow [sub_1]P$
   - $I_{adapt} \wedge P_r \Rightarrow [sub_1; K]P_i$

   La définition de sub doit vérifier les deux items suivants concernant le protocole et la sémantique.
   - Soit s l'ensemble des états possibles du STE PI_{pro} avant l'appel de op_i dans la spécification de l'adaptateur.
     Soient $s' = pre(op_i, PI_{pro})$ l'ensemble des états à partir desquels op_i est déclenchable avec $W = chemin(PI_{pro}, s,\ s')$ l'ensemble des chemins possibles entre s et s' dans PI_{pro}.
     Si l'ensemble W n'est pas l'ensemble vide, alors on peut déduire sub à partir des chemins de W.
   - Soient X'_p l'ensemble des valeurs possibles des variables X_p avant l'appel de op_i et X''_p l'ensemble des valeurs possibles des variables X_p après l'exécution de sub sur l'ensemble des valeurs X'_p, alors sub doit vérifier les propriétés suivantes $Hyp \wedge X_p \in X'_p \Rightarrow P$ et $Hyp' \wedge X_p \in X''_p \Rightarrow P_i$.

   1.2 *Supprimer l'opération non déclenchable.* On propose de supprimer op_i. Cette correction permet d'éliminer la PO fausse considérée.

   1.3 *Ajouter une garde.*
   - Err_so : Si $X'''_p \subseteq X'_p$ est un ensemble non vide tel qu'on a $X_p \in X'''_p \Rightarrow P_i$ alors on peut corriger K_{adapt} par l'ajout d'une garde $X_p = X'''_p$ avant l'appel de op_i.
   - Err_po : si $s \cap s' \neq \emptyset$, alors on peut corriger K_{adapt} par l'ajout d'une garde avant l'appel de op_i. Cette garde a pour effet le renforcement des hypothèses de la PO élémentaire considérée.

2. **Modifier I_{adapt}.** Si l'erreur est de type Err_so alors on propose de modifier la définition de la fonction F. En particulier, nous modifions l'attribut qui pose problème dans la preuve.
3. **Modifier B_{adapt}.** La correction consiste à ajouter dans la définition de la relation R les couples manquants.

État final du protocole fourni non respecté

Dans ce cas, il y a une seule correction possible qui correspond à modifier B_{adapt}. Elle consiste à modifier la relation R de telle façon que tout état de F_r soit lié uniquement à des états de F_p.

Non-conformité entre l'interface requise et l'interface fournie par l'adaptateur

Les différentes corrections possibles, dans ce cas, sont : Modifier I_{adapt}, Modifier B_{adapt}, Modifier K_{adapt}.

Nous affinons les corrections proposées comme suit :

1. **Modifier K_{adapt}.**
   1.1 *Postfixer K_{adapt}.* On propose d'ajouter à la fin de la définition de K_{adapt} une composition d'opérations fournies sub définie par $P|K$ qui vérifie les propriétés suivantes :

- $I_{adapt} \wedge P_r \Rightarrow [K_{adapt}]P$
- $I_{adapt} \wedge P_r \Rightarrow [K_{adapt}; K]\neg[K_r]\neg I_{adapt}$

Pour guider la recherche de *sub*, on procède comme suit.

- Soient s_p l'ensemble des états du protocole de l'interface fournie juste après l'exécution de K_{adapt} et s_r l'ensemble des états attendus du protocole requis juste après l'exécution de $op \mathrel{\widehat{=}} P_r|K_r$. On propose de chercher tous les chemins $W = chemin(T_p, s_p,\ R(s_r))$. Si W est différent de l'ensemble vide, alors on déduit *sub* à partir des éléments de W.
- Soient X'_p les valeurs des attributs de l'interface fournie après l'exécution de K_{adapt}, X'_r les valeurs attendues des attributs de l'interface requise et X'''_p les valeurs des attributs obtenues après l'exécution de *sub* sur X'_p. Alors *sub* doit vérifier les conditions suivantes : $X'_p \Rightarrow P$ et chaque X''_p est lié au moins à un élément de X'_r.

1.2 *Supprimer des opérations.* On peut proposer de supprimer certains appels d'opération dans K_{adapt}.

1.3 *Ajouter des gardes.* Renforcer certaines gardes ou ajouter des gardes dans K_{adapt} afin d'exclure des états du protocole ou des valeurs des attributs de l'interface fournie non liés avec respectivement des états dans s_r et des valeurs dans X'_r.

2. **Modifier I_{adapt}.** Si l'erreur est de type Err_so', on peut proposer de modifier la définition des fonctions permettant de lier les attributs des deux interfaces. En particulier, nous modifions l'attribut qui pose problème dans la preuve.
3. **Modifier B_{adapt}.** Ajouter dans la définition de la relation R des couples qui lient les éléments de s avec des éléments de s_r.

5.5.3 Stratégies de correction

Nous considérons la stratégie de correction séquentielle, présentée dans le chapitre 2 section 2.6.3 p. 26. Par conséquent, nous traitons les erreurs selon leur type dans l'ordre suivant :

1. Incompatibilité de l'initialisation.
2. Opération fournie non déclenchable. Nous commençons par corriger la première opération non déclenchable.
3. État final du protocole fourni non respecté.
4. Non-conformité entre l'interface requise et l'interface fournie par l'adaptateur.

Nous considérons aussi la stratégie de correction en cascade.

5.5.4 Étude de cas

Nous donnons la description des deux interfaces *RI_player* et *PI_player* qu'on veut connecter au moyen d'un adaptateur. Nous présentons, ensuite, une première version de l'adaptateur entre ces deux interfaces, Adapt_player1, que nous avons construit en nous basant sur la section 5.4. Finalement, nous détaillons les étapes de détection d'erreur, de diagnostic et de correction effectuées sur la spécification Adapt_player1 afin d'obtenir une version finale de l'adaptateur corrigée.

Description de l'interface *RI_player*

La Figure 5.18 donne la description avec des diagrammes UML de cette interface. À l'initialisation du système, il faut insérer de l'argent pour ensuite pouvoir écouter de la musique. Lorsque la musique est en marche, on peut augmenter ou baisser le volume, mettre la lecture du CD en pause, accélérer la lecture et arrêter la lecture. Si la lecteur de CDs est en pause, on peut

reprendre sa lecture. De même, si le lecteur CD est en mode lecture rapide, on peut reprendre la lecture normale.

L'interface *RI_player* possède les variables suivantes :

- *money* indique le crédit courant (l'argent inséré non dépensé). Elle est de type NAT et peut prendre l'une des valeurs de l'ensemble $\{20, 50\}$.
- *volume* est une variable de type NAT et peut prendre l'une des valeurs de $\{1, 2, 3\}$. Elle indique le volume courant.

L'interface *RI_player* possède les états suivants :

- $r1$ lorsque le lecteur de CDs est initialisé.
- r_2 lorsque il y a suffisamment de crédit pour écouter le lecteur CD.
- r_3 lorsque le lecteur de CDs est en lecture.
- r_4 lorsque le lecteur de CDs est en pause.
- r_6 lorsque le lecteur de CDs est en lecture rapide.
- r_5 lorsque le lecteur de CDs est arrêté.

Cette interface a besoin des opérations *pay*, *volume*, *play*, *pause*, *stop* et *speed* pour réaliser les tâches suivantes :

- L'opération *pay* fait croître la variable *coin* selon la monnaie insérée.
- L'opération *volume* permet de modifier le volume.
- L'opération *play* fait marcher le lecteur de CDs.
- L'opération *pause* arrête momentanément la lecteur de la musique.
- L'opération *stop* arrête de lecteur de CDs.
- L'opération *speed* accélère la lecture de la musique.

La machine B, associée à l'interface *RI_player* générée à partir de la Figure 5.18, est présentée dans l'annexe B Figure B.12.

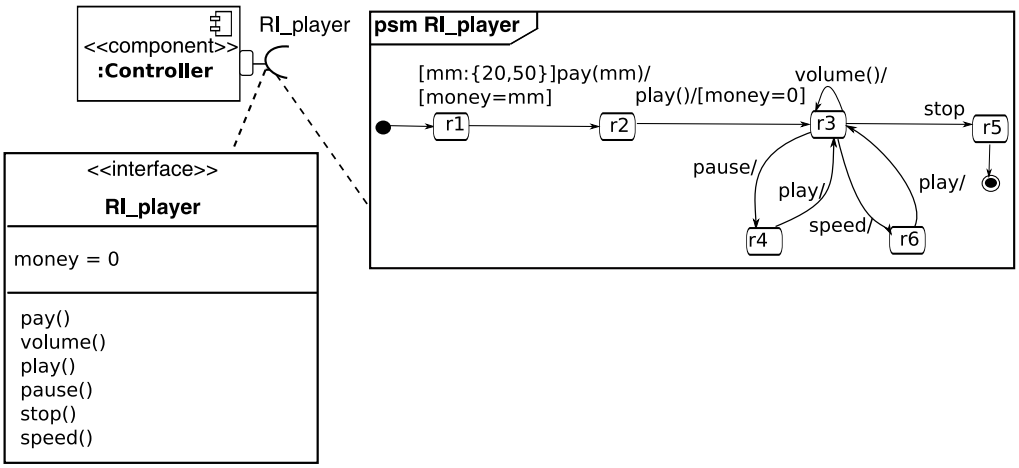


FIGURE 5.18 – Interface *RI_player*

Description de l'interface *PI_player*

Nous considérons une interface fournie *PI_player* qui est sélectionnée pour être connectée avec *RI_player*. Elle est décrite comme suit. Au début, on doit mettre en service le lecteur de CDs. On peut changer de volume tant que le lecteur de CDs est en service. À condition qu'on ait inséré suffisamment de monnaie, on peut sélectionner un CD parmi trois et démarrer sa lecture. Lorsque le lecteur de CDs est en marche, on peut le mettre en pause ou l'arrêter. Si le lecteur de

CDs est en pause, on peut le mettre en lecture rapide ou reprendre sa lecture. Lorsque le lecteur de CDs est en lecture rapide, on peut le mettre en pause. Lorsque le lecteur de CDs est en arrêt, on peut sélectionner un nouveau CD à écouter si on a encore un crédit suffisant, comme on peut mettre hors service le lecteur de CDs. La Figure 5.19 donne une description détaillée de cette interface avec des diagrammes UML.

L'interface PI_player possède les variables suivantes :

- *coin* indique le crédit courant (l'argent inséré non dépensé). Elle est de type NAT et elle peut prendre l'une des valeurs de $\{10, 20, 50\}$.
- *volume* est une variable de type NAT, qui peut prendre l'une des valeurs de $\{1, 2, 3, 4, 5, 6\}$. Elle indique le volume courant.

L'interface PI_player possède les états suivants :

- $p0$ lorsque lecteur de CDs est initialisé.
- $p1$ lorsque lecteur de CDs est mis en service.
- $p2$ lorsque il y a suffisamment de crédit pour écouter le lecteur de CDs.
- $p3$ lorsque un CD est sélectionné.
- $p4$ lorsque le lecteur de CDs est en lecture.
- $p5$ lorsque le lecteur de CDs est en pause.
- $p6$ lorsque le lecteur de CDs est en lecture rapide.
- $p7$ lorsque le lecteur de CDs est arrêté.
- $p8$ lorsque le lecteur est mis hors service.

L'interface PI_player offre les mêmes opérations que l'interface requise sauf l'opération *pay*. De plus, elle offre les opérations suivantes.

- l'opération *on* met le lecteur de CDs en service,
- l'opération off met le lecteur de CDs hors service,
- l'opération *insertcoin* fait la même fonctionnalité que l'opération requise *pay* sauf...
- Les opérations $selectCD1$, $selectCD2$ et $selectCD3$ permettent de sélectionner un CD,

Le modèle B associé à l'interface PI_player générée à partir de la Figure 5.19 est présentée dans l'annexe B Figure B.13.

Première version de l'adaptateur

On propose cette première version d'adaptateur illustrée par la Figure 5.20. La vérification de la spécification Adapt_player1 a échoué et en particulier la vérification de l'initialisation et des opérations : speed, stop et play.

Ainsi, on se retrouve avec un ensemble de POs fausses.

Détection d'erreurs, diagnostic et correction

Nous allons analyser les POs fausses une par une et nous allons proposer des corrections de la spécification Adapt_player1.

- **PO fausse associée à l'assertion**

1. Détection d'erreurs : l'obligation de preuve suivante est fausse.

```
"Local hypotheses" ∧
  stateR∈ RF_States ∧
"Check assertion (stateR∈ RF_States ⇒ stateP∈ PF_States) deduction"
⇒ stateP$1∈ PF_States
```

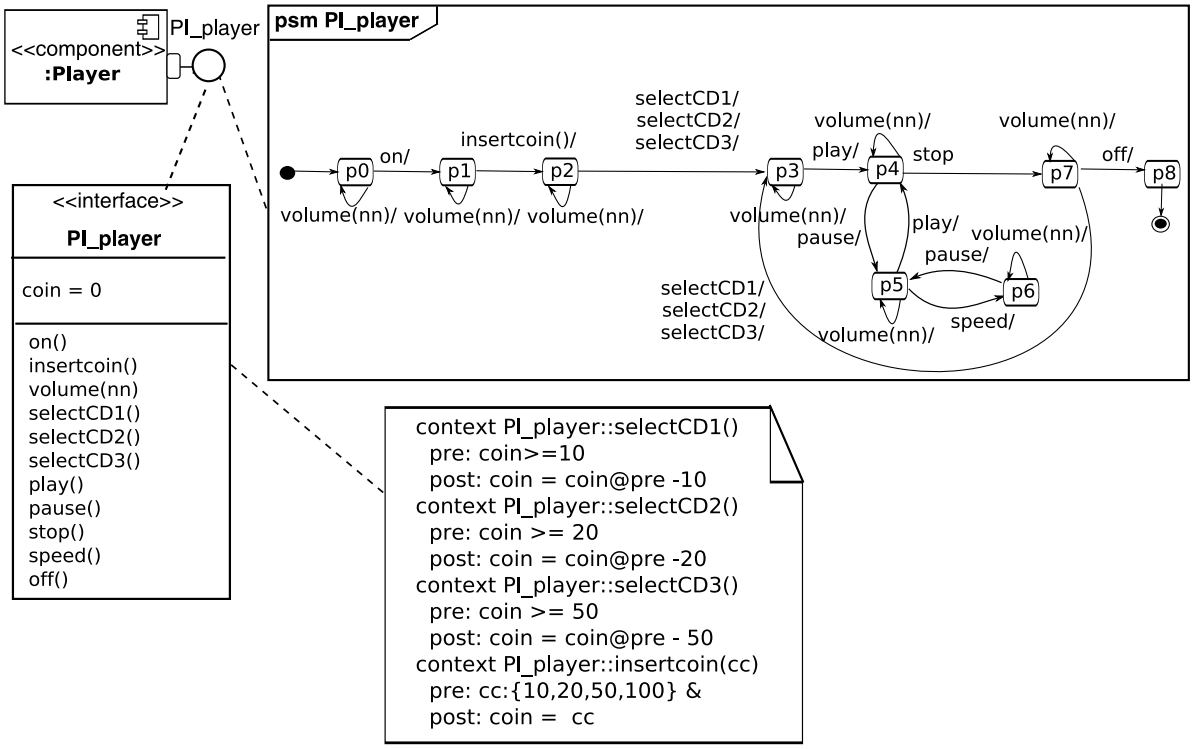


FIGURE 5.19 – Interface PI_player

2. Analyse et diagnostic : c'est une erreur de type Err_pf, en particulier le couple (p7↦r5) pose problème.
3. Correction proposée et retenue : modifier la définition de la relation REL en remplaçant l'élément (p7↦r5) par (p8↦r5). Ainsi, on réussit la vérification de l'assertion.

– **PO fausse associée à l'initialisation**

1. Détection d'erreurs : l'obligation de preuve suivante est fausse.

"Local hypotheses" ∧
p0∈ P_States ∧ 0∈ ℤ ∧
"Check that the invariant (stateP↦stateR∈ REL) is established by the initialisation "
⇒ p0↦r1∈ REL

2. Analyse et diagnostic : c'est une erreur de type Err_pi.
3. Correction proposée et retenue : modifier la définition de la relation REL en lui ajoutant l'élément (p0↦r1). Ainsi, on réussit la vérification de l'initialisation.

– **PO fausse associée à l'opération speed**

1. Détection d'erreurs : l'obligation de preuve suivante est fausse.

"speed preconditions in this component" ∧
stateR = r3 ∧
"Check preconditions of called operation , ∨ While loop construction , ∨ Assert predicates"
⇒ stateP$1 = p5

2. Analyse et diagnostic : c'est une erreur de type Err_po. On a en hypothèse stateR = r3.

```
REFINEMENT
   Adapt_player1
REFINES
   RI_player
 INCLUDES
   PI_player_0
PROMOTES
     volume,stop, speed
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL ∈ P_States ↔  R_States ∧
    REL = {(p1↦r1), (p2↦r2), (p4↦r3),
           (p5↦r4), (p6↦r6), (p7↦r5)}
INVARIANT
    (stateP ↦ stateR ) ∈ REL∧
    money =moneyr
ASSERTIONS
   (stateR ∈ RF_States) ⇒ (stateP ∈ PF_States)
OPERATIONS
   pay(mm) =
   BEGIN
        insertcoin (mm)
   END;

   play=
   BEGIN
       SELECT  stateP = p2 ∧ money = 50 THEN  SelectCD3; play1
       WHEN  stateP = p5 THEN   play1
       WHEN  stateP = p6 THEN   pause1;play1
       END
    END ;
    pause=
    BEGIN
       pause1
    END
END
```

FIGURE 5.20 – Première version de l'adaptateur entre les interfaces PI_player et RI_player

3. Corrections proposées :
   (a) Préfixer l'opération speed avec pause puisque $REL^{-1}(r3) = p4$ et il existe un chemin entre les états $p4$ et $p5$ $chemin(PI\_C_{pro}, p4,\ p5) = \{pause\}$.
   (b) Modifier la définition de la relation REL en lui ajoutant l'élément (p5↦r3).
4. Correction retenue : on choisit la première correction et on réussit la vérification de l'opération speed.

– **POs fausses associées à l'opération play**

1. Détection d'erreurs : l'obligation de preuve suivante est erronée.

```
"play preconditions in this component" ∧
  stateR∈ {r2,r4,r6} ∧
  moneyr∈ {20,50} ∧
"Local hypotheses" ∧
  stateP$1 = p2 ∧
"Check preconditions of called operation, ∨ While loop construction, ∨ Assert predicates"
⇒ 50≤money$1
```

2. Analyse et diagnostic : c'est une erreur de type Err_*so*, c'est la précondition de l'opération SelectCD3 qui a généré l'erreur et c'est la variable money$1 qui pose problème. On a en hypothèse moneyr∈ {20,50} et money$1 = moneyr.
3. Corrections proposées :
   (a) Ajouter la garde moneyr=50 avant l'opération SelectCD3.
   (b) Supprimer l'opération SelectCD3 et la remplacer par l'opération SelectCD2 ou l'opération SelectCD1.
4. Correction retenue : on choisit la deuxième correction et on remplace par l'opération SelectCD2.

La correction retenue nous permet de remplacer une opération non déclenchable par une autre qui est déclenchable mais on ne réussit pas encore la vérification de l'opération play.

1. Détection d'erreur : l'obligation de preuve suivante est fausse.

```
"play preconditions in this component" ∧
  stateR∈ {r2,r4,r6} ∧
  moneyr∈ {20,50} ∧
"Local hypotheses" ∧
  stateP$1 = p2 ∧
  money$1 = 50 ∧
  p4∈ P_States ∧
  money$1−20∈ ℤ ∧
  0≤money$1−20 ∧
  money$1−20≤2147483647 ∧
  (stateR = r6 ∧ money$1−20 = moneyr ⇒ ¬(p4↦r3∈ REL ∧
  (r3∈ RF_States ⇒ p4∈ PF_States) ∧ (p4∈ PF_States ⇒ r3∈ RF_States)) )∧
  (stateR = r4 ∧ money$1−20 = moneyr ⇒ ¬(p4↦r3∈ REL ∧
  (r3∈ RF_States ⇒ p4∈ PF_States) ∧ (p4∈ PF_States ⇒ r3∈ RF_States))) ∧
"Check that the invariant (money = moneyr) is preserved by the operation" ∧
"Check operation refinement"
⇒ money$1−20 = 0
```

2. Analyse et diagnostic : c'est une erreur de type Err_*so′*, c'est la postcondition de l'opération play qui a généré l'erreur et c'est la variable money$1 qui pose problème. On a en hypothèse moneyr∈ {20,50} et money$1=moneyr.
3. Corrections proposées :
   (a) Ajouter la garde suivante moneyr=20 avant l'opération SelectCD2. Pour le cas contraire (*i.e.* moneyr=50), faire appel à l'opération SelectCD3.
   (b) Postfixer l'opération en ajoutant une opération ou une composition d'opération tel que leur précondition $moneyr = 30 \Rightarrow PRE$ et $POST : moneyr = 0$.
   (c) Modifier le lien entre les variables money$1 et moneyr.
4. Correction retenue : on choisit la première correction. Ainsi, on réussit la vérification de l'opération play.

– **PO fausse associée à l'opération stop**

1. Détection d'erreur : l'obligation de preuve suivante est fausse.

```
"stop preconditions in this component" ∧
  stateR = r3 ∧
"Local hypotheses" ∧
```

```
  stateP$1 = p4 ∧
  p7∈ P_States ∧
"Check that the  invariant  (stateP↦stateR∈ REL) is preserved by the operation"
⇒ p7↦r5∈ REL
```

2. Analyse et diagnostic : c'est une erreur de type Err_po'. On a (p7↦r5)∉REL.
3. Corrections proposées
   (a) Postfixer l'opération stop avec off puisque $REL^{-1}(r5) = p8$ et il existe un chemin entre $p7$ et $p8$ $chemin(PI\_C_{pro}, p7,\ p8) = \{off\}$.
   (b) Ajouter le couple (p7↦r5) dans REL.
4. Correction retenue : on choisit la première correction. Si on choisit la deuxième correction, on va se contredire avec la correction effectuée dans la PO fausse associée à l'assertion. Ainsi, on réussit la vérification de l'opération stop.

Après toutes les corrections précédentes, on obtient la spécification correcte de l'adaptateur (voir Figure 5.21). On remarque qu'il existe plusieurs corrections pour chaque cas d'erreur. Par conséquent, on peut trouver plusieurs versions possibles de l'adaptateur.

5.6 Travaux connexes

Nous complétons les travaux connexes à l'adaptation logicielle statique, présentés dans le chapitre 3, par la présentation de quelques travaux [CHS06, LHHS07, CLS07] étroitement liés à notre approche d'adaptation. Ces travaux considèrent la vérification d'interopérabilité entre les interfaces des composants à l'aide de la méthode B et en particulier le problème d'adaptation. Nous allons aussi comparer ces travaux avec notre approche.

Selon notre connaissance, Chouali *et al.* [CHS06] ont été les premiers à utiliser le raffinement B pour la vérification de la compatibilité entre deux interfaces. Les interfaces considérées sont associées à un modèle de données (IDM, Interface Data Model), elles décrivent les opérations de plusieurs classes et des relations entre les classes. La vérification de la compatibilité considère les niveaux syntaxique et sémantique. Au niveau syntaxique, une condition d'appariement, qui est proche de la condition d'appariement « Plug-In » [ZW97], est vérifiée. Ils ne disent pas explicitement qu'ils font de l'adaptation des interfaces, mais ils considèrent deux interfaces avec des IDMs différents et ils définissent comment lier entre eux ces IDMs alors on peut dire qu'ils font de l'adaptation des données. Les différences entre l'approche de [CHS06] et notre approche sont les suivantes :

- nous partons d'une spécification UML des interfaces et nous les traduisons en B.
- Les interfaces que nous considérons sont plus limitées si l'on considère les données et nous ne vérifions pas de relation entre les interfaces comme dans [CHS06].
- nous considérons le niveau protocole en plus des niveaux syntaxique et sémantique.
- nous considérons la description des opérations en terme de pré/postcondition et nous définissons la condition d'appariement entre les pré/postconditions.

Une étude a été menée par Colin *et al.* [CLS07] pour l'adaptation du modèle de données. Pour exprimer et vérifier les correspondances entre les données, les auteurs procèdent par étapes successives à travers des raffinements B successifs. Le processus d'adaptation des modèles B se décompose en trois grandes étapes de raffinement. La première étape est l'adaptation des variables qui prépare la mise en correspondance entre les attributs des deux interfaces. La deuxième étape est l'adaptation des types de données qui correspond au transtypage des données et la dernière

```
REFINEMENT
   RI_player_r
REFINES
    RI_player
 INCLUDES
   PI_player
PROMOTES
     volume
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL ∈ P_States ↔   R_States ∧
    REL = {(p0|−>r1),(p2↦r2),(p4↦r3),
           (p5↦r4),(p6↦r6),(p8|−>r5)}
INVARIANT
    (stateP ↦ stateR ) ∈ REL∧
    money =moneyr
ASSERTIONS
   (stateR ∈ RF_States) ⇒ (stateP ∈ PF_States)
OPERATIONS
   pay(mm) =
   BEGIN
      on;
      insertcoin (mm)
   END;

   play=
   SELECT stateP = p2 ∧ money = 50 THEN SelectCD3;play1
   WHEN stateP = p2 ∧ money = 20 THEN SelectCD2;play1
   WHEN stateP = p5 THEN play1
   WHEN stateP = p6 THEN pause1;play1
   END;

   stop =
   BEGIN
      stop1;off
   END;

   speed =
   BEGIN
      pause1;speed1
   END;

   pause=
   BEGIN
      pause1
   END
END
```

FIGURE 5.21 – Adaptateur entre les interfaces PI_player et RI_player

étape est l'inclusion de l'interface fournie. Notre approche d'adaptation est plus limitée si l'on considère l'adaptation des données puisque nous ne considérons pas le transtypage des données.

Lanoix *et al.* [LHHS07] proposent de généraliser notre approche [MLS06] par la construction d'un adaptateur pour l'assemblage de plusieurs composants qui fournissent ou requièrent des interfaces. L'adaptateur doit réaliser l'ensemble des interfaces requises des composants à l'aide de leurs interfaces fournies. De notre point de vue, cette généralisation revient à la même chose que de vérifier chacune des deux interfaces séparément, la différence est juste d'ordre syntaxique. À notre avis la vérification de la compatibilité au niveau protocole de cette approche est assez limitée. En effet, la vérification du protocole peut être suffisante à condition que le protocole

de chaque interface soit décrit d'une manière indépendante, tandis que souvent la description du protocole d'un composant qui offre et requiert des interfaces est exprimée par l'ensemble des opérations de ses interfaces requises et fournies et il n'est pas significatif d'exprimer le protocole de chaque interface séparément. Le deuxième point faible de cette approche est que la plupart du temps lorsqu'on considère l'assemblage de plusieurs composants, on ne considère pas une relation de correspondance uniquement entre deux interfaces mais on décrit la coopération entre les différentes interfaces des composants.

Dans [LCS07], les auteurs ont présenté une étude plus globale des différents cas d'assemblages possibles dans une architecture à base de composants en considérant les travaux [MLS06, CLS07, LHHS07]. Chaque cas est caractérisé par un schéma d'assemblage particulier.

5.7 Conclusion

Nous avons proposé dans ce chapitre une approche pour l'assemblage de deux interfaces, caractérisée par l'utilisation de la méthode B pour la vérification. Nous avons introduit une approche d'adaptation générative corrective (voir classification en Section 3.4.5, p. 57) où nous commençons par construire une première version de l'adaptateur en nous basant sur des règles de correspondances. Ensuite, nous effectuons les étapes de détection des incompatibilités, de diagnostic et de correction permettant d'aboutir à une spécification d'adaptation corrigée.

D'une part, cette approche nous a permis d'appliquer les résultats présentés dans le chapitre 4 : les schémas présentés dans les sections 4.4.1 et 4.4.2 ont été utilisés pour la vérification respectivement de l'assemblage de deux interfaces et de l'adaptation. D'autre part, nous avons repris les résultats sur la détection, le diagnostic et la correction des spécifications B d'une manière générale (voir Section 2.6 p. 24) et nous les avons appliqués pour la correction de la spécification de l'adaptateur, en tenant compte du contexte.

Notre approche reste limitée pour l'assemblage de plusieurs composants. Nous abordons dans le prochain chapitre (Chapitre 6), une approche pour l'assemblage de plusieurs composants.

6

Assemblage de plusieurs composants

Sommaire

6.1 Introduction 127
6.2 Présentation de l'approche 128
6.3 Les trois premières étapes 130
6.3.1 Spécification globale du système 130
6.3.2 Description de l'architecture du système en termes de composants . . . 130
6.3.3 Spécification des fonctionnalités en termes des composants 131
6.4 Formalisation 132
6.4.1 Les STEs 133
6.4.2 Diagrammes de séquences 133
6.4.3 Comportement du système assemblé 134
6.5 Vérification 135
6.5.1 Traduction en B 135
6.5.2 Vérifications en B 137
6.6 Détection d'erreur, diagnostic et correction 138
6.6.1 Typologie des erreurs 138
6.6.2 Correction du médiateur 138
6.6.3 Stratégies de correction 141
6.6.4 Étude de cas 141
6.7 Conclusion 149

6.1 Introduction

Ce chapitre présente une approche de vérification de protocoles d'assemblage de composants [MSA08a]. Nous considérons d'une part, une spécification du protocole du système abstrait en termes de fonctionnalités et d'autre part, une spécification du système assemblé prenant en compte les protocoles des composants présentés pour réaliser le système ainsi que les communications entre ces composants. Nous considérons un médiateur qui a comme rôle de réaliser les fonctionnalités du système ; de plus toutes les interactions entre les composants passent par ce médiateur.

La vérification de l'assemblage consiste à montrer que le système assemblé est un raffinement correct du système abstrait. Par conséquent, les fonctionnalités du système abstrait sont réalisées par le système assemblé et il n'y a pas d'états de blocage dans les interactions entre les composants

du système. Pour effectuer cette vérification, nous utilisons des « constructions B » définies dans le chapitre 4. Comme dans le chapitre précédent nous effectuons la détection et le diagnostic des erreurs ainsi que la correction des spécifications. Les résultats concernant le diagnostic et la correction des erreurs ont été présentés dans [MSA08b].

Le chapitre est organisé comme suit. La section 6.2 présente l'approche suivie. La section 6.3 décrit les trois premières étapes correspondant au développement des spécifications avec application à l'étude de cas simplifiée du contrôle d'accès à un bâtiment. Une formalisation des différents concepts des systèmes de transitions étiquetés et des diagrammes de séquences nécessaires à la vérification de l'assemblage des composants sur le plan des protocoles est donnée à la section 6.4. La section 6.5 décrit l'étape de vérification conduisant à la détection d'erreurs et nécessite la transformation en B des différentes spécifications. L'analyse du retour de la preuve en cas d'échec et un diagnostic d'erreurs, basés sur une typologie des erreurs, nous conduisent à une proposition de correction des spécifications présentée dans la section 6.6. Ces résultats sont illustrés par l'étude de cas du contrôle d'accès à un bâtiment.

6.2 Présentation de l'approche

L'approche proposée est constituée de sept étapes. Les trois premières correspondent au développement des spécifications dans notre cadre de travail ; les quatre étapes suivantes constituent l'apport de notre approche avec la vérification des spécifications, la détection d'erreurs, leur analyse et la proposition de correction des spécifications :

Étape 1. Spécification du fonctionnement global du système sous la forme d'un automate décrit par un système de transitions étiqueté.

Étape 2. Description de l'architecture du système en termes de composants existants, chaque composant étant lui-même défini par un système de transitions étiqueté.

Étape 3. Spécification des fonctionnalités du système en termes des composants de l'architecture. Chaque fonctionnalité est décrite à l'aide d'un ou plusieurs diagrammes de séquences explicitant les interactions entre les composants.

Étape 4. Vérification. L'ensemble des spécifications produites lors des trois étapes précédentes est transformé en B permettant d'effectuer la vérification et d'utiliser les outils de preuves associés. Le résultat de cette étape est déterminant pour la suite du processus. Soit la preuve se passe bien et le processus est terminé, soit la preuve échoue, des erreurs sont détectées et le processus continue avec les trois étapes suivantes.

Étape 5. Détection d'erreurs. Cette étape est le résultat de la vérification de l'assemblage des composants sur le plan des protocoles.

Étape 6. Analyse et diagnostic d'erreurs. Ils s'effectuent à partir de la détection d'erreurs de l'étape 5 et d'une typologie des erreurs liée à notre cadre de travail.

Étape 7. Proposition de correction des spécifications du système.

Étude de cas : un système de contrôle d'accès à un bâtiment

Cette étude de cas est une variante simplifiée de l'étude de cas du contrôle d'accès à un bâtiment [Abr98, AFA00]. L'objectif est de construire un système chargé de contrôler l'accès de certaines personnes à un bâtiment correspondant à un lieu de travail. Le contrôle s'effectue sur la base de l'autorisation que chaque personne concernée est censée posséder. Cette autorisation

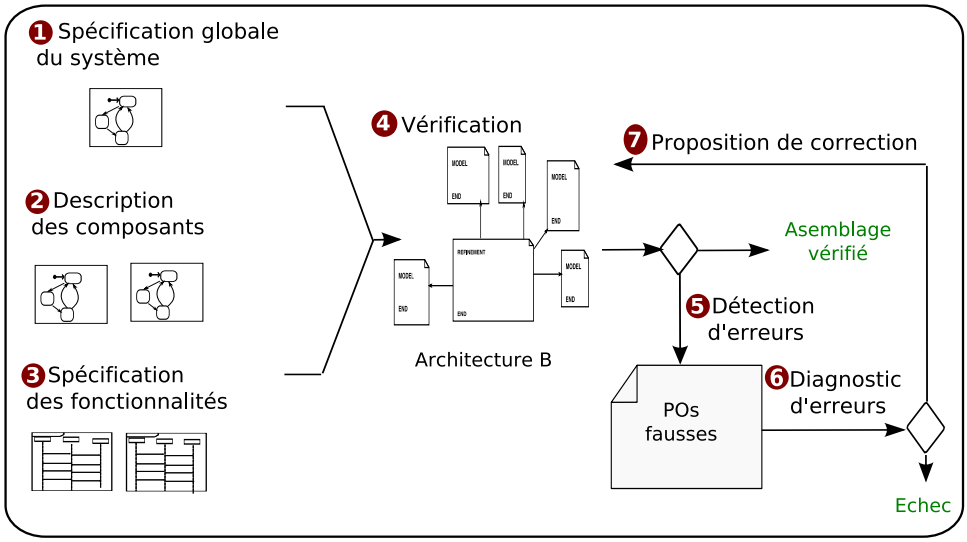


FIGURE 6.1 – Vue d'ensemble de notre approche

permet à une personne sous le contrôle du système, d'entrer dans le bâtiment. Les autorisations sont données de façon permanente, c'est-à-dire qu'elles ne changent pas durant le fonctionnement normal du système.

Chaque personne autorisée dispose d'une carte d'accès avec un code. Un lecteur de cartes est installé à l'entrée du bâtiment. À proximité du lecteur se trouvent deux voyants, un rouge et un vert, chacun d'eux pouvant être allumé ou éteint.

Le système utilise deux tourniquets, un pour l'entrée dans le bâtiment, l'autre pour la sortie. Les tourniquets peuvent être bloqués ou débloqués. Lorsqu'un tourniquet est débloqué par le système, le passage d'une personne est détecté par un capteur. L'entrée obéit au protocole suivant :

- Si la personne demandant à entrer dans le bâtiment est autorisée à entrer, le voyant vert s'allume et le tourniquet se débloque pour un laps de temps donné. Dès que la personne franchit le tourniquet, le voyant vert s'éteint et le tourniquet se bloque immédiatement. Si la carte n'a pas été reprise par la personne au bout du laps de temps autorisé, elle est « avalée » par le lecteur.
- Si la personne n'est pas autorisée à entrer, le voyant rouge s'allume et le tourniquet reste bloqué. Ici encore, le retrait de la carte est soumis à une durée limite au-delà de laquelle la carte est « avalée » par le lecteur.

Lorsqu'une personne se trouve à l'intérieur du bâtiment, sa sortie doit également être contrôlée par le système de façon à ce qu'il soit possible de savoir à tout instant qui se trouve dans le bâtiment.

Le comportement global du système est le suivant. Chaque matin, il est initialisé, permettant aux personnes autorisées d'entrer et de sortir du bâtiment. À minuit, une fois le bâtiment vide, le système est désactivé et il n'est plus possible d'accéder au bâtiment.

6.3 Les trois premières étapes

6.3.1 Spécification globale du système

Une première description globale est donnée dans laquelle le système est considéré comme un composant boîte noire offrant un ensemble de fonctionnalités. Elle précise l'enchaînement entre les différentes fonctionnalités offertes. Le comportement global du système ou protocole, est défini par un système de transitions étiqueté gardé (voir Définition. 4.2.5). Il est également possible de spécifier en langage naturel des propriétés générales du système, comme par exemple l'absence de blocage.

Étude de cas : spécification globale du système

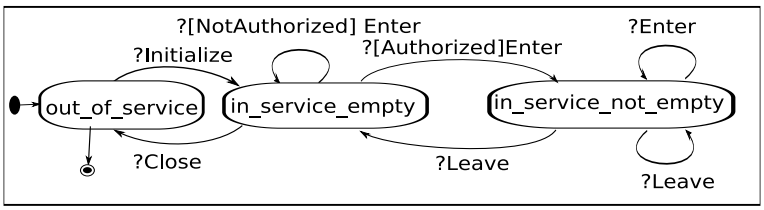


FIGURE 6.2 – Spécification globale du système de contrôle d'accès

La Figure 6.2 présente le protocole d'utilisation du système de contrôle d'accès dans lequel trois états ont été introduits :

- *out_of_service* désigne l'état dans lequel le système de contrôle est désactivé. Le bâtiment est vide et les personnes autorisées à entrer dans le bâtiment sont connues du système. Cet état correspond à l'état initial et final du système,
- *in_service_empty* désigne l'état dans lequel le système est prêt pour le contrôle et les personnes autorisées peuvent entrer
- *in_service_not_empty* désigne l'état où les personnes autorisées peuvent entrer ; les personnes présentes dans le bâtiment peuvent sortir.

Le système offre les quatre fonctionnalités suivantes :

- l'activation du système en début de journée, modélisée par la transition étiquetée par ?*Initialize*,
- la demande d'entrer d'une personne dans le bâtiment, modélisée par la transition étiquetée par ?*Enter*,
- la demande de sortie du bâtiment, modélisée par la transition étiquetée par ?*Leave*,
- la désactivation du système en fin de journée si le bâtiment est vide, modélisée par la transition étiquetée par ?*Close*.

Remarque 6.3.1. Dans l'étiquette de chaque transition, le symbole ? (respectivement !) devant l'événement correspond à un événement reçu (respectivement envoyé).

6.3.2 Description de l'architecture du système en termes de composants

Une spécification de l'architecture globale du système est ensuite explicitée sous la forme d'un diagramme de structure composite d'UML 2.0 en termes des composants utilisés et des interfaces à connecter.

Dans notre approche, nous utilisons systématiquement un médiateur chargé de synchroniser les communications entre les différentes instances des composants du système, toutes les communications passant exclusivement par ce médiateur.

Le protocole de chaque composant est lui-même décrit par un système de transitions étiqueté gardé (voir Définition 4.2.5 et Définition 6.4.1).

Étude de cas : architecture globale du système et composants utilisés

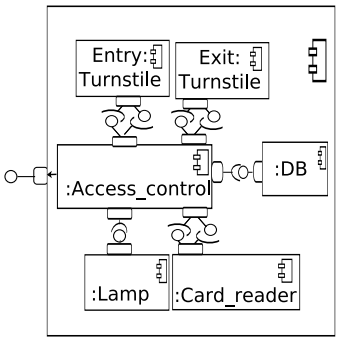


FIGURE 6.3 – Architecture globale du système en termes des composants utilisés

L'architecture du système de contrôle d'accès présentée dans la Figure 6.3 fait appel aux quatre composants *Lamp*, *Card_reader*, *DB* et *Turnstile*. *Entry* et *Exit* sont deux instances du composant *Turnstile*. *Access_control* est ici le médiateur.

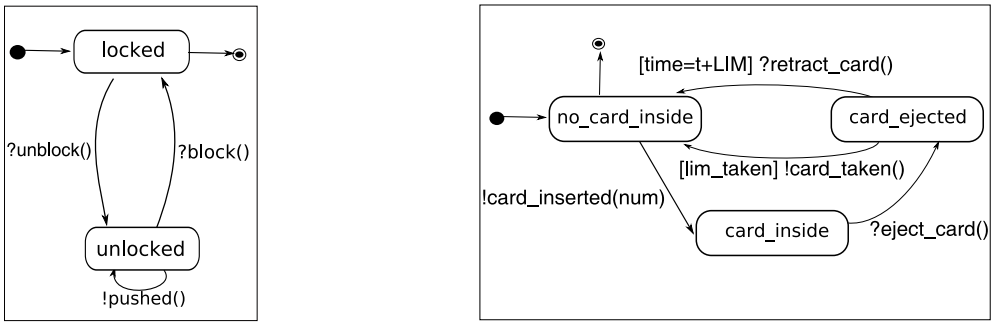
Le comportement du composant *Turnstile* est présenté par le système de transitions étiqueté de la Figure 6.4(a) ; celui-ci comporte deux états *locked* et *unlocked*, *locked* correspondant à l'état initial. Les deux transitions étiquetées par ?*block* et ?*unblock* permettent le changement d'état du composant. La transition étiquetée par !*pushed* correspond au franchissement du tourniquet par une personne qui provoque son entrée dans le bâtiment (ou sa sortie).

Le comportement du composant *Card_reader* est décrit par le système de transitions étiqueté de la Figure 6.4(b) ; il comporte trois états, l'état initial et l'état final étant le même, *no_card_inside*, et quatre transitions. La garde de la transition étiquetée $[lim\_taken]!card\_taken()$ entre les états *card_ejected* et *no_card_inside* correspond à une abstraction de prédicat défini par $time \in [t...t + LIM]$ où LIM représente une constante.

La description complète des protocoles des composants du système est donnée dans l'annexe C Section C.1, p 181.

6.3.3 Spécification des fonctionnalités en termes des composants

Chaque fonctionnalité du système est décrite à l'aide d'un ou de plusieurs diagrammes de séquences d'UML 2.0 explicitant les interactions entre les composants. À chaque instance de composant intervenant dans une fonctionnalité, on fait correspondre une ligne de vie dans les diagrammes de séquences, le médiateur ayant aussi sa ligne de vie. Le médiateur étant chargé de synchroniser les communications entre les différentes instances des composants du système, les flèches relient uniquement les lignes de vie des instances des composants à celle du médiateur mais en aucun cas les lignes de vie des instances des composants entre elles.



(a) STE du composant *Turnstile*

(b) STE du composant *Card_reader*

FIGURE 6.4 – Composants de l'architecture du système de contrôle d'accès

Remarque 6.3.2. Un diagramme de séquences décrit un scénario et ne décrit pas forcément tous les cas possibles. Notre objectif n'est pas de vérifier l'exhaustivité dans la spécification mais d'apporter des aides à la correction des spécifications en cours de construction.

Étude de cas : description des fonctionnalités *Leave* et *Enter*

Le diagramme de séquences de la Figure 6.5 décrit le comportement du système lorsqu'une personne quitte le bâtiment, il correspond à la fonctionnalité *Leave* : le tourniquet est activé par la personne qui sort (*pushed*()) et la base de données est informée du départ de cette personne et met à jour ses informations (*delete*()).

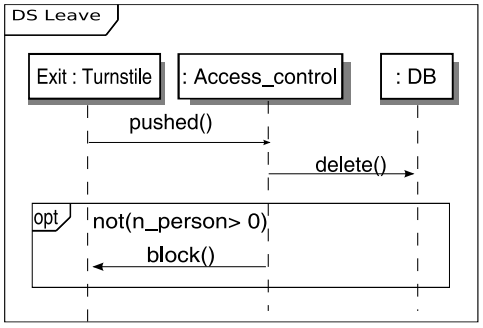


FIGURE 6.5 – Un scénario pour la fonctionnalité *Leave*

Remarque 6.3.3. Lorsqu'une fonctionnalité est décrite à l'aide de plusieurs diagrammes de séquences, ces diagrammes sont composés séquentiellement ou de façon conditionnelle. Nous n'avons pas encore étudié le cas où plusieurs diagrammes de séquences sont exécutés en parallèle.

6.4 Formalisation

Nous décrivons les notations utilisées pour formaliser l'assemblage des composants, principalement : les STEs et les diagrammes de séquences (DS). Comme certaines de ces notions ont été définies précédemment mais sous des formes sensiblement différentes, nous mettons en évidence et justifions ces différences.

6.4.1 Les STEs

La description globale du protocole est donnée en terme de STE (Définition 4.2.1, p. 60), les étiquettes (ou événements) utilisées correspondent aux fonctionnalités du système auxquelles nous ajoutons un signe ? ou ! selon que l'événement est reçu ou envoyé. Le protocole de chaque composant censé réaliser le système est décrit par un STE gardé tel qu'il est défini dans la définition 4.2.5 p. 61 et chaque étiquette de ces STEs gardés est définie comme suit :

Définition 6.4.1 (Étiquette)**.**
Les étiquettes des systèmes de transitions associés aux composants sont de la forme $[g]\ sig\ e$ où :

- e est un événement du composant.
- g est une garde, c'est-à-dire un prédicat dépendant des paramètres de e, des variables du composant et des variables du contexte.
- sig est le signe qui a pour valeur ? ou ! selon que l'événement est reçu ou envoyé par le composant.

Remarque 6.4.1. La représentation et la définition utilisées pour les systèmes de transitions étiquetés est proche de celle des « Protocol State Machine » introduites dans UML 2.0 [OMG05] qui sont une spécialisation des State Machines [Har87], avec une distinction entre les événements reçus et les événements envoyés.

L'absence de blocage dans le protocole des composants est une propriété assez importante. Dans ce qui suit, nous définissons un état de blocage par rapport aux STEs gardés. Nous donnons, ensuite, la définition de l'absence de blocage dans un STE gardé. Ces définitions restent valable pour les STEs non gardés (en considérant toutes les gardes vraies).

Définition 6.4.2 (État de blocage)**.**
Soient $T = (A,\ G(x), D,\ S,\ s_0,\ F,\ \rightarrow)$ un STE gardé. On définit le prédicat $deadlock(s)$ signifiant qu'un état s de S est un état de blocage de T par :

$$deadlock(s) \stackrel{\text{def}}{=} s \notin F \wedge\ \neg((\forall x \in D)\ (\exists e \in A)\ (\exists g \in G)\ (\exists s' \in S)\ (s \xrightarrow{[g(x)]e}\ s'))$$

s est un état de blocage s'il n'est pas un état terminal et s'il n'existe pas de transition d'origine s.

Définition 6.4.3 (Absence de blocage dans un STE gardé)**.**
Soit T un STE gardé et S son ensemble des états, T possède la propriété d'absence de blocage si et seulement si :

$$(\forall s \in S)\ \neg deadlock(s)$$

6.4.2 Diagrammes de séquences

UML ne définit pas formellement de sémantique pour les diagrammes de séquences, généralement la sémantique considérée est basée sur les traces.

En UML 2.0 les gardes ne sont pas directement associées aux événements, mais les fragments combinés permettent d'associer une garde à un ensemble d'événements grâce aux constructions telles que *alt*, *opt* et *loop*. De cette façon on peut associer une garde à chaque événement d'un diagramme de séquences. Les états des composants (de leur protocole) n'apparaissent pas directement dans les diagrammes de séquences, on peut toutefois associer des états aux composants

tout au long de leur ligne de vie. Plus précisément, à tout point de la ligne de vie d'un composant, on peut associer un état à condition que ce composant soit correctement utilisé (*i.e.* conformément à son protocole défini par son STE) dans le diagramme de séquences tout au long de sa ligne de vie avant le point donné.

Nous formalisons un diagramme de séquences comme une composition d'étiquettes (Définition 4.2.12, p. 63) en utilisant les constructeurs *seq*, *alt*, *opt*, et *loop*, les prédicats utilisés peuvent être sur les états des protocoles des composants mais aussi sur les variables du contexte.

Définition 6.4.4 (Formalisation des diagrammes de séquences)**.**

Soit A un ensemble d'événements et P un ensemble de prédicats. L'ensemble $DS(A)$ des diagrammes de séquences associé à A est défini de manière inductive [LS06] de la façon suivante :

- $\{\epsilon\} \cup A$ est l'ensemble des éléments de base de $DS(A)$
- Les règles de formation des diagrammes de séquences sont les suivantes :
  - $ds_1, ds_2 \in DS(A) \wedge p \in P \Rightarrow ds_1\ seq\ ds_2 \in DS(A) \wedge ds_1\ alt\ p\ ds_2 \in DS(A)$
  - $ds \in DS(A) \wedge p \in P \Rightarrow loop\ p\ n\ sub \in DS(A) \wedge opt\ p\ ds \in DS(A)$

Cette définition inductive des diagrammes de séquences ne prend pas en compte tous les diagrammes de séquences puisque les compositions parallèles ou les négations ne sont pas considérées. Les étiquettes considérées dans la formalisation d'un diagramme de séquences sont les étiquettes des STEs des protocoles des composants intervenant dans le diagramme de séquences. Cette formalisation nous permet d'effectuer des traductions rigoureuses des diagrammes de séquences et d'utiliser les définitions de la section 4.3.3.

Exemple 6.4.1. La formalisation du diagramme de séquences du service *Leave* (Figure 6.5, p. 132) est : $pushed\ \mathbf{seq}\ delete\ \mathbf{seq}\ \mathbf{opt}(not(n\_person > 0))\ block$

6.4.3 Comportement du système assemblé

Le système abstrait est caractérisé par un STE décrivant son protocole. Le système assemblé est constitué des STEs gardés décrivant les protocoles des composants et par des diagrammes de séquences associés aux fonctionnalités.

Nous définissons le comportement du système assemblé par un raffinement du STE du système abstrait par les STEs des composants, par rapport à une relation entre états et une relation entre étiquettes. Notre objectif est de vérifier que le système assemblé est un raffinement correct du système abstrait.

La définition suivante établit un raffinement d'un STE par un ensemble de STEs par rapport à une relation entre étiquettes et une relation entre états. Le premier STE est celui spécifiant le système global (Étape 1, Section 6.3.1), les seconds sont ceux des composants (Étape 2, Section 6.3.2), la relation d'étiquette est définie par les diagrammes de séquences (Étape3, Section 6.3.3) et la relation d'états est complétée par le concepteur.

Définition 6.4.5. Soient $T_0, ..., T_n$, n+1 STEs associés respectivement aux protocoles du système global et des composants et $T = (A,\ G,\ D,\ S,\ s_0,\ F,\ \rightarrow)$ le produit libre de $T_1, ..., T_n$. Le comportement du système assemblé est défini par le raffinement du T_0 par $T_1, ..., T_n$ par rapport à une relation R et une application χ, si on a les quatre propriétés suivantes :

1. $(s_0^0, (s_0^1, ..., s_0^n)) \in R$
2. $(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ (\forall x \in D)\ (\forall e \in A_0)\ (\forall sub \in \mathcal{E}(A))$
   $[sub = \chi(e) \wedge (p, (q_1, ..., q_n)) \in R \wedge p \xrightarrow{e}_0 \Rightarrow ((q_1, ..., q_n), x) \in pre(sub, T)]$

3. $(\forall p \in S_0)\ (\forall (q_1, ..., q_n, q'_1, ..., q'_n) \in S)\ (\forall e \in A_0)\ (\forall sub \in \mathcal{E}(A))\ (\forall c \in Chemin(sub, T))$ $[sub = \chi(e)\ \wedge\ (p, (q_1, ..., q_n)) \in R\ \wedge origine(c) = (q_1, ..., q_n) \wedge extremite(c) = (q'_1, ..., q'_n)$ $\Rightarrow (\exists p' \in S_0)\ [p \overset{e}{\rightarrow}_0 p' \wedge (p', (q'_1, ..., q'_n)) \in R]]$
4. $(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ [(p, (q_1, ..., q_n)) \in R\ \Rightarrow\ ((q_1, ..., q_n) \in F \Rightarrow p \in F_0)]$

Proposition 6.4.1 (Raffinement de chemin)**.** *Tout chemin dans le système assemblé (Def. 6.4.5) est un raffinement d'un autre chemin dans le système abstrait.*

Démonstration. La sémantique opérationnelle du système assemblé est définie par un ensemble de chemins (Définition 4.4.2 p. 82). Cette proposition se déduit de manière directe à partir des items un et trois de la définition 6.4.5 et du corollaire 4.4.5. □

La proposition suivante met en évidence la préservation de la propriété d'absence de blocage.

Proposition 6.4.2 (Absence de blocage)**.** *Si la propriété d'absence de blocage est préservée dans le système abstrait alors elle est aussi préservée dans le système assemblé.*

Démonstration. Nous considérons les états atteignables à partir de l'état initial du raffinement. Nous effectuons une preuve par contradiction. On suppose que dans le raffinement il existe un état s de blocage qui est atteignable. deux cas se présentent :

- s est atteignable à partir de s_0 et à la fin de l'exécution d'une fonctionnalité. D'après le corollaire 4.4.5 p. 83, il existe un état s' dans le STE abstrait tel que $(s, s') \in R$, deux cas se présentent :
  - s' est un état final dans T_0, alors la quatrième propriété de la définition 6.4.5 est violée.
  - s' n'est pas un état final dans T_0, alors $(\forall x \in D)\ (\exists e \in A)\ (\exists g \in G)\ (\exists s' \in S)\ (s \xrightarrow{[g(x)]e}\ s')$ puisque la propriété d'absence de blocage est préservée dans T_0. Par conséquent la deuxième propriété de la définition 6.4.5 n'est pas respectée.
- s est atteignable au milieu d'exécution d'une fonctionnalité e à partir de s_0 avec $sub = \chi(e)$. Alors s est un état au milieu d'un chemin c appartenant à l'ensemble $Chemin(sub, T)$ et ne peut pas être un état de blocage.

CQFD. □

6.5 Vérification

Afin d'effectuer les vérifications des propriétés de la définition 6.4.5, nous traduisons en B l'assemblage des composants.

6.5.1 Traduction en B

Nous reprenons la construction de la section 4.4.3, p. 83 en considérant les STEs $T_0 = (A_0,\ S_0,\ s_0^0,\ F_0,\ \rightarrow_0), T_i = (A_i,\ G,\ D,\ S_i,\ s_0^i,\ F_i, \rightarrow_i), 1 \leq i \leq n$ et $T = (A,\ G,\ D,\ S,\ s^0,\ F,\ \rightarrow)$ qui décrivent respectivement le protocole du système abstrait, le protocole des différents composants dans le système assemblé et le produit libre de $T_1, ..., T_n$, une relation $R \subseteq S_0 \times (S_1 \times ... \times S_k)$ et une application χ de A_0 vers $\mathcal{E}(\bigcup_{i=1}^{n} A_i)$ qui est définie par des diagrammes de séquences.

Soit M_a le modèle B, traduction du STE correspondant à la spécification globale du système (T_0 décrit dans l'étape 1). Le modèle B, M_{med}, qui traduit l'architecture du système en termes de composants décrite dans les étapes 2 et 3 de l'approche, est défini comme suit :

- Dans la traduction des STEs (du système abstrait et des composants) en B nous nous basons sur la section 4.3.1 p. 65), mais nous ne considérons pas les préconditions pour les opérations (cela revient à avoir des conditions vraies). Dans cette traduction, nous ajoutons la propriété d'absence de blocage (Définition. 6.4.3) dans la clause **ASSERTIONS** afin de la vérifier.
- M_{med} est un raffinement de M_a, au sens de B : M_{med} **REFINES** M_a
- Le raffinement B impose que les deux modèles possèdent les mêmes opérations. Le corps de chaque opération dans M_{med} est obtenu en traduisant en B le diagramme de séquences de la fonctionnalité associée à l'opération considérée (Section 4.3.3, p. 67).
- L'invariant de collage entre M_{med} et M_a établit une relation R entre les états du STE du système global et celui du STE modélisant l'architecture du système en termes de composants. L'écriture de cet invariant de collage est à la charge de l'utilisateur et donc sujet à erreurs.
- Pour prendre en compte les différents composants de l'architecture, on utilise la clause **INCLUDES** qui permet d'utiliser les variables et les opérations des différents modèles B, correspondant aux STEs de ces composants.
- Pour terminer la construction de M_{med}, un modèle B, *Clock*, représentant une horloge globale est incluse via la clause **INCLUDES**, permettant de simuler le temps grâce à une variable *time*.
- Dans le but de renforcer les propriétés de la relation R nous ajoutons les formules suivantes dans la clause **ASSERTIONS** de M_{med}.

  1. À chaque fois que les composants sont tous dans un état final le système abstrait le sera aussi. Cette propriété s'écrit comme suit.

     $$(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ [\ (q_1, ..., q_n) \in F \Rightarrow p \in F_0]$$

  2. Pour s'assurer qu'à chaque fois qu'une fonctionnalité est déclenchable dans le système abstrait alors elle est déclenchable dans le système concret, nous introduisons la propriété suivante J dans l'assertion.

     $J\ :\ (\forall p \in S_1)\ (\forall (q_1, ..., q_n) \in S)\ (\forall e \in A_1)\ (\forall sub \in \mathcal{E}(A))$

     $$[sub = \chi(e)\ \wedge\ p \xrightarrow{e}_1 \Rightarrow ((q_1, ..., q_n), x) \in pre(sub, T)]$$

Pour la deuxième assertion qu'on ajoute dans M_{med}, le calcul de l'ensemble $pre(sub)$ n'est pas si évident puisqu'il dépend de tous les évènements qui apparaissent dans sub. Dans le but de générer les obligations de preuve associées à cette assertion sans avoir à l'écrire, nous proposons de faire une nouvelle construction B. Pour réaliser celle-ci, nous proposons de reprendre les spécifications B, associées aux différents STEs de cette construction B en ajoutant les préconditions aux différentes opérations des spécification des STEs (qui décrivent les protocoles des composants et du système abstrait). Ensuite, nous considérons seulement les obligations de preuve associées aux préconditions qui sont les mêmes que les obligations de preuve générées pour l'assertion J_{med}.

Étude de cas : architecture du système en termes de composants en B

L'architecture B du système présentée en Figure 6.6 est composée des spécifications suivantes :

- Le modèle Abstract_AccessControl correspondant à la spécification globale du système.
- Les machines Card_reader, DB, Lamp, Exit . Turnstile et Entry . Turnstile associées aux composants utilisés dans l'architecture du système. Clock modélise l'horloge globale.

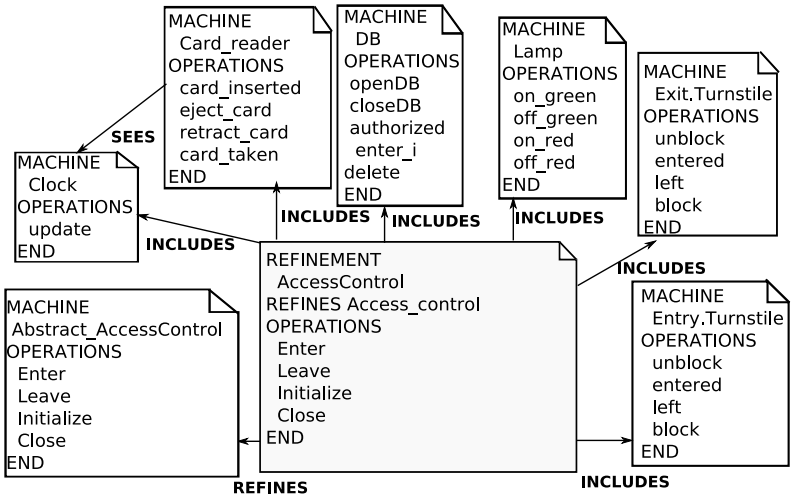


FIGURE 6.6 – Architecture B du système en termes de composants

– La spécification AccessControl qui est définie par un raffinement : AccessControl **REFINES** Abstract_AccessControl. Elle inclut les différentes machines : **INCLUDES** Card_reader, DB, Lamp, Exit. Turnstile, Entry. Turnstile, Clock. Elle possède les mêmes opérations que la machine Abstract_AccessControl, à savoir Enter, Leave, Initialize et Close.

L'invariant de collage entre les variables du système assemblé et du système abstrait est la suivante :

(state ↦ ((Left1.stateT) ↦ (stateDB ↦ (stateL ↦ (stateCard ↦ (Entry.stateT)))))) ∈ REL

Il est basé sur une relation *REL* (Figure 6.7), celle-ci exprime la relation liant les états des composants avec les états du système global.

```
PROPERTIES
    REL = {(out_of_service↦ ( locked ↦ ( DBclosed ↦ ( off ↦ ( no_card_inside ↦ locked))))) ,
           (in_service_empty ↦ ( locked ↦ ( DBclosed ↦ ( off ↦ ( no_card_inside ↦ locked))))) ,
           (in_service_not_empty ↦ ( unlocked ↦ ( DBclosed ↦ ( off ↦ ( no_card_inside ↦ locked)))))}
```

FIGURE 6.7 – Définition de la relation REL

6.5.2 Vérifications en B

Prouver que M_{med} **REFINES** M_a consiste à prouver que chaque opération de M_{med} « raffine » l'opération correspondante de M_a (c'est-à-dire possédant le même nom). La proposition suivante, montre les propriétés assurées par les obligations de preuve à démontrer pour le raffinement de chaque fonctionnalité du système.

Proposition 6.5.1. *Les obligations de preuve relatives au raffinement M_a par M_{med} avec la même construction présentée en Section 6.5.1 assure les propriétés données dans la définition 6.4.5.*

Démonstration. Cette propriété se déduit en partie à partir des propositions 4.4.4 p. 79, 4.4.7 p. 84 et 4.4.8 p. 84. Les items deux et quatre de la définition 6.4.5 se déduisent directement des obligations de preuve associées aux assertions. □

6.6 Détection d'erreur, diagnostic et correction

Notre objectif est d'apporter des aides à la correction de la spécification du système assemblé et de ses fonctionnalités, sans modification ni remplacement des composants utilisés et sans modification de sa spécification abstraite. Précisément, nous nous intéressons à la correction de la spécification du médiateur.

Nous nous basons sur les résultats concernant la détection, l'analyse et la correction des erreurs dans une spécification B (introduites dans la section 2.6 p. 24). De plus, nous disposons des informations supplémentaires sur la forme générale des spécifications et leur sémantique. Nous proposons, ainsi, des corrections spécifiques à notre contexte.

L'organisation de cette section est la même que la section 5.5 p. 114 du chapitre 5. Nous commençons par présenter une typologie des erreurs, dans la section 6.6.1. Ensuite, nous proposons, dans la section 6.6.2, un ensemble de corrections selon le type d'erreur. À noter, que l'approche proposée pour guider la correction du médiateur, présente des points communs avec celle pour guider la correction de l'adaptateur puisque dans les deux cas, on considère la vérification d'une relation de raffinement entre des STEs.

6.6.1 Typologie des erreurs

La détection des erreurs dépend du processus de vérification utilisé et des propriétés à vérifier. Dans le cadre de notre approche de développement des spécifications par assemblage de composants avec vérification de la compatibilité sur le plan des protocoles, ces types d'erreurs ont été identifiés :

1. Incompatibilité de l'initialisation. Les états initiaux des deux spécifications abstraite et concrète ne sont pas liés. Ce type d'erreur est détecté par les POs associées à l'initialisation, en particulier, aux POs de type $po7$. On note ce type d'erreur par Err_pi.
2. Évènement non déclenchable. Une fonctionnalité est déclenchable dans le système abstrait mais elle n'est pas déclenchable dans le système assemblé, *i.e.* une ou plusieurs opérations qui apparaissent dans cette fonctionnalité ne sont pas déclenchables. Cette erreur est détectée par des POs fausses de type $po8$ et $po11$. On distingue deux cas.
   (a) Évènement non déclenchable dû à une erreur de garde. On note ce type d'erreur par Err_po.
   (b) Évènement non déclenchable dû à une erreur de protocole. On note ce type d'erreur par Err_go.
3. Les incompatibilités générées par la non-conformité entre la spécification abstraite et la spécification concrète. Ce type d'erreur est détecté par les POs fausses, associées aux opérations, de type $po10$. On note ce type d'erreur par Err_po'.
4. État final non lié à un état final. Ce type d'erreur est détecté par les POs fausses associées aux assertions (POs de type $po2$), en particulier, l'assertion suivante pose problème.

$$(\forall p \in S_0)\ (\forall (q_1, ..., q_n) \in S)\ [\ (q_1, ..., q_n) \in F \Rightarrow p \in F_0]$$

6.6.2 Correction du médiateur

La spécification du médiateur est définie par un raffinement de la spécification abstraite du système réalisée en se basant sur un ensemble de contraintes de synchronisation entre les composants. La correction de cette spécification peut être effectuée de deux manières : (i) soit

en modifiant les contraintes de synchronisation décrites par le médiateur (ii) soit en modifiant la relation entre les états des protocoles du système abstrait et du système assemblé.

Les corrections prennent en compte les informations collectées dans l'étape d'analyse et de diagnostic d'erreurs telles que :

- le type d'erreur,
- les variables sources de cette erreur,
- l'évènement et l'interface responsables de l'erreur (dans le cas d'une erreur de type Err_*po* ou Err_*go*),
- la valeur des différentes variables et des hypothèses prises par le système sur les variables dans la PO fausses,

Elles considèrent également la spécification du système abstrait ainsi que la spécification des interfaces de chaque composant.

Dans ce qui suit, on considère deux spécifications :

- une spécification abstraite du système décrite par le STE $T_0 = (A_0,\ S_0,\ s_0^0,\ F_0,\ \rightarrow)$,
- la spécification des n composants $T_1, ..., T_n$,

On considère M_a la spécification B associée à l'interface T_0 avec stateMa la variable de contrôle utilisée et $M_1, ..., M_n$, les spécifications B associées aux STEs $T_1, ..., T_n$, avec stateMi $(1 \leq i \leq n)$ les variables de contrôle de chaque T_i. La spécification M_{med} qui décrit le protocole du médiateur avec I_{med} comme invariant. R est la relation qui lie les variables de contrôle du système abstrait et les composants. Pour chaque type d'erreur qui peut apparaître dans la spécification de M_{med}, nous proposons des corrections appropriées.

Incompatibilité de l'initialisation

Une seule correction est possible pour l'initialisation qui est applicable dans ce type d'erreur : Modifier B_{med}. La modification de B_{med} consiste à modifier R. Nous proposons d'ajouter dans la définition de la relation R le couple $(s_0^0, (s_1^0, ..., s_n^0))$, s'il n'existe pas et de supprimer tous les autres couples de la forme $(s_0, (s_1, ..., s_n))$ avec $(s_1, ..., s_n) \neq (s_1^0, ..., s_n^0)$.

Évènement non déclenchable dû à une erreur de protocole

On considère, par exemple, une opération $op = K_{med}$ du médiateur non vérifiée à cause d'une opération fournie $op_i = P_i|K_i$ de l'interface d'un composant C_i, qui est appelée dans la définition de op et qui n'est pas déclenchable à cause d'une erreur de protocole.

Les différentes corrections possibles, dans ce cas, sont : Modifier B_{med}, Modifier K_{med}. Nous détaillons, chacune de ces corrections.

1. **Modifier modifier K_{med}.**
   1.1 *Insérer des opérations dans K_{med}.* On propose de préfixer l'appel de op_i par une composition d'opérations. On considère deux substitutions sub_1 et sub_2 qui correspondent respectivement à la substitution qui précède et qui suit op_i dans K_{med}. On propose de modifier $K_{med} = sub_1; sub; sub_2$ par $sub_1; sub; op_i; sub_2$ telle que sub est une composition d'opérations sur le STE T_i définie par $P|K$.
   Soit s l'ensemble d'états possibles du STE T_i avant l'appel de op_i dans la spécification du médiateur. Soient $s' = pre(op_i, T_i)$ l'ensemble des états à partir desquels op_i est déclenchable et $W = chemin(T_i, s,\ s')$ l'ensemble des chemins possibles entre s et s'.
   Si l'ensemble W n'est pas un ensemble vide, alors on peut déduire sub à partir des chemins de W sinon cette correction ne sera pas applicable.

1.2 *Supprimer l'opération non déclenchable.* On propose de supprimer op_i. Cette correction permet d'éliminer la PO fausse considérée.

1.3 *Ajouter une garde.* Si $s'' \subseteq s$ est un ensemble non vide, tel que $s'' \subseteq s'$ alors on peut corriger K_{adapt} par l'ajout d'une garde avant l'appel de op_k. Cette garde a pour effet de renforcer les hypothèses de la PO élémentaire considérée.

2. **Modifier B_{med}.** Cette correction consiste à ajouter dans la définition de la relation R des liens qui lient tous les éléments de s avec les éléments de s' non liés.

Évènement non déclenchable dû à une erreur de garde

De la même manière, on considère l'opération $op_i = P_i|K_i$ qui n'est pas déclenchable mais cette fois à cause d'une erreur de garde. Soient s et g respectivement l'état du STE T_i et la garde de e avant l'appel de l'événement e dans la spécification de la fonctionnalité, $s \in pre(e, \rightarrow)$. Soit g_C la garde de e dans C relativement à s, telle que $g \not\Rightarrow g_C$. Nous proposons les corrections suivantes :

- **Supprimer l'évènement**
  L'évènement e de la spécification de service n'étant pas déclenchable, il peut être supprimé.
- **Renforcer la garde**
  Ce cas peut se produire si $(\exists x, g \wedge g_C) = True$. Dans le meilleur des cas, on la remplace par $g \wedge g_C$
  Cette correction conduit à une modification de l'ensemble des traces d'exécution de la fonctionnalité. Si à partir de l'état s, un seul événement e est déclenchable, la restriction de sa garde a pour effet :
  - d'introduire implicitement deux cas d'exécution possibles (voir Figure 6.8 (a)).
  - d'introduire une branche non faisable (voir Figure 6.8 (b)).

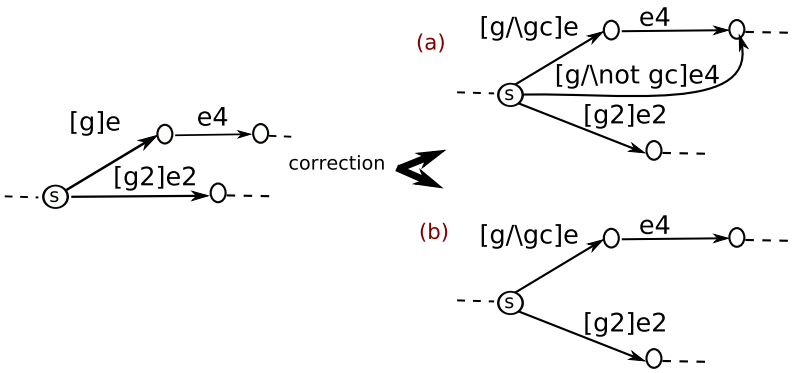


FIGURE 6.8 – Renforcer une garde

De la même manière, si à partir de l'état s plusieurs événements sont déclenchables, des cas d'exécution où rien ne se passe peuvent alors être introduits (voir Figure 6.8 (b)).

- **Remplacer la garde**
  Ce cas peut se produire si $(\forall x, g \wedge g_C) = FALSE$. Le cas le plus simple est de la remplacer par g_C.

Remarque 6.6.1. Dans certains cas, faire des corrections en modifiant des gardes entraîne l'introduction de nouveaux cas d'exécution. Puisque ces derniers n'étaient pas prévus par le concepteur dès le départ, alors il y a de fortes chances qu'on introduise des erreurs de protocole.

Si on modifie la garde d'un évènement, alors il peut être déclenché si sa garde est vérifiée ou rien ne peut se passer (*skip*).

État final non conforme

Dans ce cas, il y a une seule correction possible qui correspond à modifier B_{med}. En particulier, elle correspond à modifier la relation R : on propose de modifier les états de l'ensemble F_0 qui sont liés à des états qui n'appartiennent pas à F et inversement.

Les incompatibilités générées par la non-conformité entre la spécification abstraite et la spécification concrète

Les différentes corrections possibles dans ce cas, sont : Modifier I_{med}, Modifier B_{med}, Modifier K_{med}.

Nous affinons les corrections proposées dans ce type d'erreur, comme suit :

1. **Modifier K_{med}.**
   1.1 *Postfixer K_{med}.* On propose d'ajouter à la fin de la définition de K_{med} une composition d'opérations fournies *sub* définie par $P|K$. Pour guider la recherche de *sub*, on procède comme suit. Soient s un état du protocole de l'interface fournie atteint à la fin de K_{med} et s' les états attendus du protocole requis. On propose de chercher tous les chemins $W = chemin(T_p, s,\ R^{-1}(s'))$. Si W est différent de l'ensemble vide, alors on déduit *sub* à partir des éléments de W sinon cette correction n'est pas applicable.
   1.2 *Supprimer des opérations.* On peut proposer de supprimer certains appels d'opération dans K_{med}.
   1.3 *Ajouter des gardes.* Renforcer certaines gardes ou les ajouter dans K_{med} afin d'exclure les états du système concret atteignables à la fin d'une fonctionnalité et qui posent problème.
2. **Modifier I_{med}.**
3. **Modifier B_{med}.** Ajouter dans la définition de la relation R des liens manquants.

6.6.3 Stratégies de correction

Nous considérons la stratégie de correction par ordre, déjà présentée dans le chapitre 2 Section 2.6.3 p. 26. Par conséquent, nous traitons les erreurs selon leur type dans l'ordre suivant :

1. Incompatibilité de l'initialisation.
2. Évènement non déclenchable dû à une erreur de garde.
3. Évènement non déclenchable dû à une erreur de protocole.
4. Les incompatibilités générées par la non-conformité entre la spécification abstraite et le système assemblé.
5. L'état final du protocole fourni non respecté.

Nous considérons aussi la stratégie de correction en cascade.

6.6.4 Étude de cas

Nous avons introduit une erreur dans la définition de la relation REL qui lie les variables de contrôle :

```
CONSTANTS
    REL
PROPERTIES
    REL = {(out_of_service  ↦ (locked    ↦ (DBclosed ↦ (off ↦ (no_card_inside ↦ locked)))))),
           (in_service_empty  ↦ (locked    ↦ (DBclosed ↦ (off ↦ (no_card_inside ↦ locked)))))),
           (in_service_not_empty ↦ (unlocked ↦ (DBclosed ↦ (off ↦ (no_card_inside ↦ locked)))))))}
```

FIGURE 6.9 – Fausse définition de la relation REL

On définit la relation REL de telle façon que le composant DB reste dans l'état (DBclosed) au début et à la fin de tous les services Figure 6.9. Un scénario d'entrée d'une personne dans le bâtiment erroné est fourni Figure 6.10. Le médiateur Access_Control, communique avec les composants Card_reader, DB, Entry.Turnstile et Lamp. La spécification B de tout le système est présentée dans la Figure C.5, p 185.

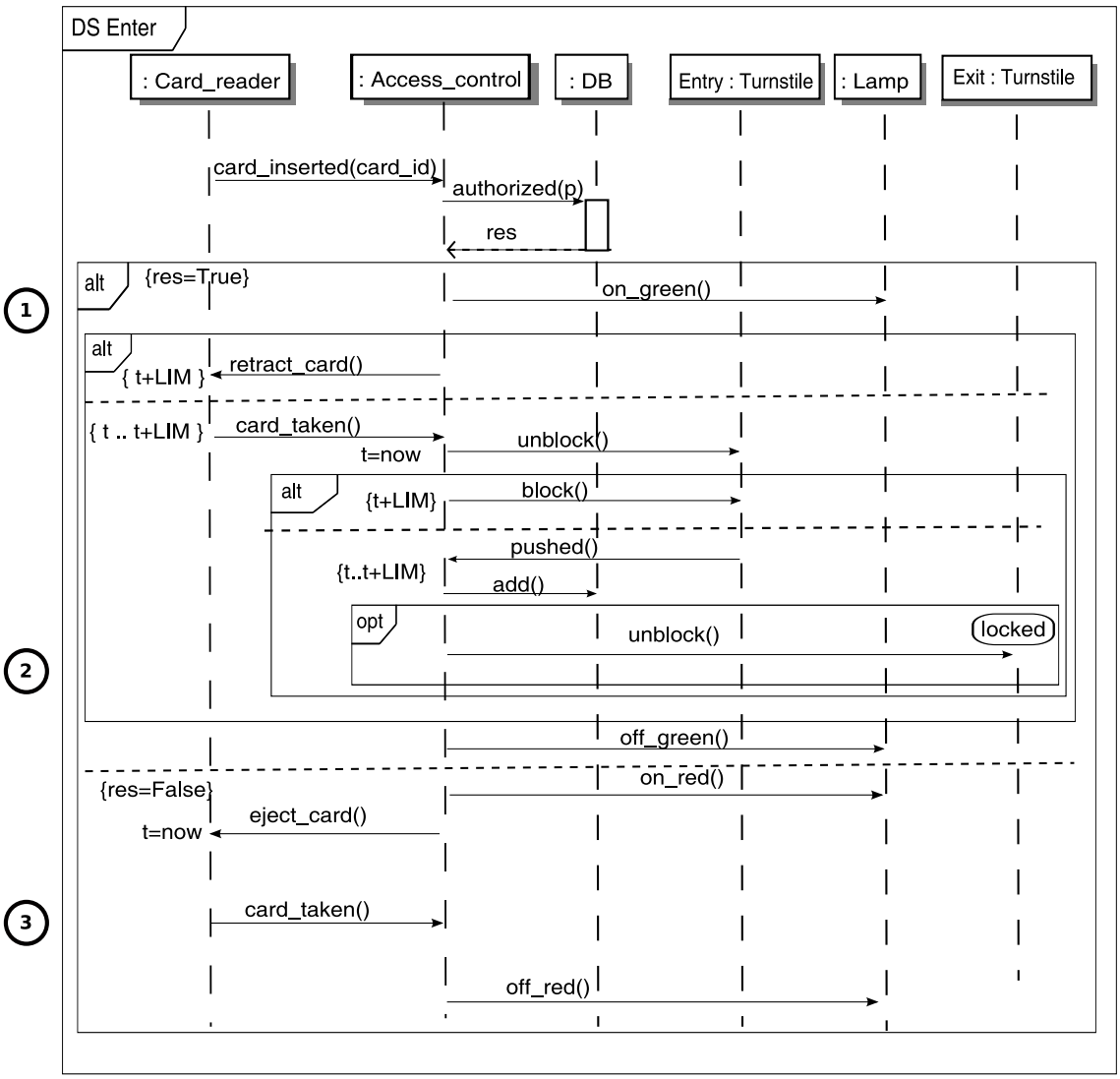


FIGURE 6.10 – Diagramme erroné de la fonctionnalité Enter

La vérification de la spécification présentée dans la Figure C.5 échoue, nous détectons un ensemble d'obligations de preuve fausses générées pour les services *Enter* et *Leave*. Nous traitons les erreurs détectées une à une dans ce qui suit.

Première correction

1. **Détection d'erreur.** L'obligation de preuve suivante, associée à la fonctionnalité *Enter* est fausse.

```
"‘UserEnter preconditions in this component’" ∧
state∈ {in_service_empty,in_service_not_empty} ∧
"‘Local hypotheses’" ∧
stateCard$1 = no_card_inside ∧
"‘Check preconditions of called operation , ∨ While loop construction , ∨ Assert predicates ’"
⇒
stateDB$1 = DBopened
```

2. **Analyse et diagnostic d'erreurs.** À partir des erreurs détectées, nous constatons que l'échec est dû à la variable de contrôle stateDB de l'interface DB. Il s'agit d'une erreur de type Err_*po*.
3. **Proposition de correction.** À partir de cette analyse, les corrections suivantes sont proposées :
   - préfixer la fonctionnalité Enter par l'événement ?openDB,
   - modifier la relation REL en liant les états in_service_not_empty et in_service_empty avec l'état DBopened de l'interface *DB*.

Nous retenons la deuxième correction proposée (voir Figure C.6, p. 186). Comme l'invariant est modifié par cette correction, nous proposons de suivre la stratégie de correction par cascade qui consiste à vérifier si la modification effectuée sur l'invariant génère des erreurs dans d'autres fonctionnalités et à les corriger en premier. En effet, nous constatons que la correction précédente permet de corriger les erreurs détectées dans la fonctionnalité *Leave* et introduit de nouvelles erreurs en particulier dans les fonctionnalités *Init* et *Close*. Dans ce qui suit, nous considérons la correction de ces deux fonctionnalités, en premier lieu, même si la fonctionnalité *Enter* contient encore des erreurs.

Deuxième correction

1. **Détection d'erreur.** Une erreur dans la fonctionnalité *Init* est détectée par la PO suivante.

```
"‘ init preconditions in this component’" ∧
state = out_of_service ∧
"‘Check that the invariant
(state↦(Left1stateT↦(stateDB↦(stateL↦(stateCard↦EntrystateT))))∈ REL)
is preserved by the operation − ref 4.4, 5.5’"
⇒
in_service_empty↦(Left1stateT$1↦(stateDB$1↦(stateL$1↦
(stateCard$1↦EntrystateT$1))))∈ REL
```

2. **Analyse et diagnostic d'erreurs.** À partir de cette PO, nous identifions une erreur de type Err_*po′*. Comme hypothèse, l'état du système abstrait est *out_of_service* et comme but, son état est *in_service_empty*. En plus, la fonctionnalité *Init* ne modifie pas l'état

des composants. Dans la relation REL, les états $out\_of\_service$ et $in\_service\_empty$ sont liés aux mêmes états des composants sauf pour l'état du composant DB.

Nous disposons des informations suivantes :
- DBclosed correspond à l'état du composant DB à la fin de la fonctionnalité Enter,
- DBopened est l'état du composant DB attendu à la fin de Enter,
- il existe un chemin reliant les deux états

$$chemin(T_{DB},\ \{DBclosed\},\ \{DBopened\}) = \{?DBopen()\} \neq \emptyset\ .$$

3. **Proposition de correction.** Suite à cette analyse, la proposition de correction consiste à :
   - postfixer la fonctionnalité Init par l'événement ?openDB,
   - modifier la relation REL en liant in_service_empty avec l'état DBclosed de l'interface DB ou out_of_service avec l'état DBopened. À noter que le premier cas est contradictoire avec une correction effectuée précédemment.

   Nous appliquons la première correction (voir Figure C.6).

Nous considérons maintenant la correction de la fonctionnalité *Close*

Troisième correction

1. **Détection d'erreur.**

```
"'close preconditions in this component'" ∧
state = in_service_empty ∧
"'Check that the invariant
(state↦(Left1stateT↦(stateDB↦(stateL↦(stateCard↦EntrystateT))))∈ REL)
is preserved by the operation – ref 4.4, 5.5'"
⇒
out_of_service↦(Left1stateT$1↦(stateDB$1↦(stateL$1↦(stateCard$1↦EntrystateT$1))))∈ REL
```

2. **Analyse et diagnostic d'erreurs.** Cette erreur est de type Err_po'. Nous effectuons une analyse analogue à l'erreur précédente.

   Nous disposons des informations suivantes :
   - DBopened correspond à l'état du composant DB à la fin de la fonctionnalité Enter,
   - DBclosed est l'état du composant DB attendu à la fin de Enter,
   - il existe un chemin reliant les deux états

$$chemin(T_{DB},\ \{DBopened\},\ \{DBclosed\}) = \{?DBclose()\} \neq \emptyset\ .$$

3. **Proposition de correction.** Nous proposons alors les corrections suivantes.
   - postfixer la fonctionnalité Close par l'événement ?closeDB,
   - modifier la relation REL en liant in_service_empty avec l'état DBclosed de l'interface DB ou in_service_empty avec l'état DBopened.

   Nous appliquons la première correction (voir Figure C.6).

Il nous reste à corriger la fonctionnalité *Enter* qui contient encore des erreurs. Nous terminons sa correction.

Quatrième correction

1. **Détection d'erreurs.** Les obligations de preuves non prouvées identifiées sont :

```
"'UserEnter preconditions in this component'" ∧
state∈ {in_service_empty,in_service_not_empty} ∧
```

"'Local hypotheses'" ∧
res$0∈ 𝔹 ∧
stateCard$1 = no_card_inside ∧
stateDB$1 = DBopened ∧
res$0 = TRUE ∧
stateL$1 = off ∧
xx∈ ℤ ∧
0≤xx ∧
xx≤2147483647 ∧
time$1+1≤2147483647−xx ∧
t_eject$1+LIM≤time$1+xx ∧
"'Check preconditions of called operation,
or While loop construction, ∨ Assert predicates'"
⇒
card_inside = card_ejected

2. **Analyse et diagnostic d'erreurs.**

   À partir des erreurs détectées, nous constatons que les obligations de preuve non prouvées concernent la variable de contrôle stateCard du composant Card_reader. L'examen des hypothèses permet de situer précisément l'erreur : elle correspond au cas où l'utilisateur est autorisé à entrer ($res = True$ dans l'alternative) et se situe au niveau du symbole **1** encerclé dans la partie gauche du diagramme de séquences Figure 6.10. Par ailleurs, ces erreurs ne sont pas liées au premier événement concernant ce composant appelé dans la fonctionnalité Enter. Par conséquent, il s'agit d'une erreur liée à un événement non déclenchable dû à une erreur de type Err_*po*. Plus précisément, en l'état du système card_inside, l'événement ?retract_card() n'est pas déclenchable. Intuitivement, une erreur a été introduite dans le protocole d'utilisation du lecteur de cartes, la carte n'a pas été éjectée après sa lecture. Nous avons une autre PO fausse de type Err_*po* pour l'évènement !card_taken() et qui a la même conclusion que la PO considérée dans cette section.

   Nous disposons des informations suivantes :
   - card_inside correspond à l'état du composant Card_reader avant l'erreur détectée dans Enter,
   - les deux événements non déclenchables sont ? $retract\_card()$ et ! $card\_taken()$. Pour ces deux événements, on a :
     $Prec(T_{Card\_reader},\ !\ card\_taken()) =$
     $Prec(T_{Card\_reader},\ ?\ retract\_card()) = \{card\_ejected\}$
   - il existe un chemin reliant les deux états $card\_inside$ et $card\_ejected$:
     $chemin(T_{Card\_reader},\ \{card\_inside\},\ \{card\_ejected\}) = \{?\ eject\_card()\} \neq \emptyset$

3. **Proposition de correction.**

   Nous proposons de corriger la fonctionnalité Enter en ajoutant l'événement ? $eject\_card$ avant ces deux événements identifiés comme non déclenchables. Cette correction est prise en compte dans la spécification de la Figure 6.12 et dans le diagramme associé à la fonctionnalité Enter de la Figure 6.11.

Cinquième correction

Continuons l'analyse de la fonctionnalité Enter.

1. **Détection d'erreurs.** Les obligations de preuves non prouvées sont :

```
"'UserEnter preconditions in this component'" ∧
state∈ {in_service_empty,in_service_not_empty} ∧
"'Local hypotheses'" ∧
res$0∈ 𝔹 ∧
xx∈ ℤ ∧
0≤xx ∧
xx≤2147483647 ∧
time$1+1≤2147483647−xx ∧
¬(time$1+LIM≤time$1+xx) ∧
xx$0∈ ℤ ∧
0≤xx$0 ∧
xx$0≤2147483647 ∧
time$1+xx+1≤2147483647−xx$0 ∧
time$1+xx+xx$0+1≤time$1+xx+LIM ∧
n_person$1+1≤2147483647 ∧
1≤n_person$1 ∧
¬(Left1stateT$1 = locked) ∧
stateCard$1 = no_card_inside ∧
stateDB$1 = DBopened ∧
res$0 = TRUE ∧
stateL$1 = off ∧
EntrystateT$1 = locked ∧
no_card_inside∈ Card_States ∧
DBopened∈ DB_States ∧
n_person$1+1∈ ℤ ∧
0≤n_person$1+1 ∧
off∈ Lights_States ∧
unlocked∈ Turnstile_States ∧
time$1+xx+xx$0∈ ℤ ∧
0≤time$1+xx+xx$0 ∧
time$1+xx+xx$0≤2147483647 ∧
state = in_service_not_empty ⇒
¬(in_service_not_empty↦(Left1stateT$1↦(DBopened↦
(off↦(no_card_inside↦unlocked))))∈ REL∧
res$0∈ 𝔹 ∧ time$1+xx∈ ℤ ∧ 0≤time$1+xx ∧ time$1+xx≤2147483647) ∧
"'Check that the invariant
(state↦(Left1stateT↦(stateDB↦(stateL↦(stateCard↦EntrystateT))))∈ REL)
is preserved by the operation − ref 4.4, 5.5'"
⇒
∃(state$0).(state$0∈ {in_service_empty,in_service_not_empty} ∧
state$0↦(Left1stateT$1↦(DBopened↦(off↦(no_card_inside↦unlocked))))∈ REL)
```

2. **Analyse et diagnostic d'erreurs.** Cette erreur est de type Err_po'. Nous constatons que les obligations de preuves non prouvées concernent l'état final du composant Entry. Turnstile dans la fonctionnalité Enter, l'erreur dans le diagramme de séquences se situe au niveau de **2**. Intuitivement, le tourniquet en entrée n'a pas été re-bloqué après le passage de la personne. Nous disposons des informations suivantes :
   - locked correspond à l'état du composant Entry. Turnstile à la fin de la fonctionnalité Enter,
   - unlocked est l'état du composant Entry. Turnstile attendu à la fin de Enter,
   - il existe un chemin reliant les deux états
   $$chemin(T_{Turnstile},\ \{locked\},\ \{unlocked\}) = \{?block()\} \neq \emptyset\ .$$
3. **Proposition de correction.** Suite à cette analyse, la proposition de correction consiste

à :
- postfixer la fonctionnalité Enter par l'événement ?block(),
- modifier l'invariant de collage en liant les états in_service_empty et in_service_not_empty avec l'état unlocked du composant Entry.Turnstile.

Nous considérons la première correction (voir Figure 6.11 et Figure 6.12).

Sixième correction

Après correction de l'erreur précédente, la vérification continue.

1. **Détection d'erreurs.** La preuve de Enter relativement au composant Card_reader échoue. L'obligation de preuve non prouvée identifiée est :

$$
\begin{array}{l}
\text{"`UserEnter preconditions in this component'"} \wedge \\
\mathsf{state} \in \{\mathsf{in\_service\_empty}, \mathsf{in\_service\_not\_empty}\} \wedge \\
\text{"`Local hypotheses'"} \wedge \\
\mathsf{res\$0} \in \mathbb{B} \wedge \\
\mathsf{stateCard\$1} = \mathsf{no\_card\_inside} \wedge \\
\mathsf{stateDB\$1} = \mathsf{DBopened} \wedge \\
\neg(\mathsf{res\$0} = \mathsf{TRUE}) \wedge \\
\mathsf{stateL\$1} = \mathsf{off} \wedge \\
\mathsf{xx} \in \mathbb{Z} \wedge \\
0 \leq \mathsf{xx} \wedge \\
\mathsf{xx} \leq 2147483647 \wedge \\
\mathsf{time\$1}+1 \leq 2147483647-\mathsf{xx} \wedge \\
\text{"`Check preconditions of called operation, } \vee \text{ While loop construction,} \\
\text{or Assert predicates'"} \\
\Rightarrow \\
\mathsf{time\$1}+\mathsf{xx}+1 \leq \mathsf{time\$1}+\mathsf{LIM}
\end{array}
$$

2. **Analyse et diagnostic d'erreurs.** À partir des erreurs détectées, nous constatons que l'obligation de preuve non prouvée concerne une contrainte temporelle associée à l'événement ?card_taken() du composant Card_reader. Cet événement n'est pas le premier de ce composant appelé dans Enter. Nous en déduisons que l'erreur est liée à la garde de cet événement, erreur de type Err_*go*. L'erreur localisée est notée **3** dans la Figure 6.10.

   Nous disposons des informations suivantes :
   - Il n'y a pas de garde qui porte sur la variable *time* et qui est associée à l'événement ?card_taken() dans Enter lors de la détection de l'erreur,
   - $time + LIM$ est la garde requise par le composant Card_reader pour le déclenchement de l'événement ?card_taken().

3. **Proposition de correction.** Nous proposons de corriger la fonctionnalité Enter en ajoutant une garde à l'événement ?card_taken(). Nous introduisons implicitement un deuxième cas d'exécution possible lorsque la garde ajoutée n'est pas vérifiée, dans lequel nous ajoutons l'appel de retract_card. Concrètement nous remplaçons l'appel de card_taken non déclenchable par :

```
IF time ≥ t_eject + LIM THEN
   retract_card
ELSE
     card_taken
END
```

Cette erreur est corrigée dans le diagramme de séquences de la Figure 6.11 et dans la spécification de la Figure 6.12.

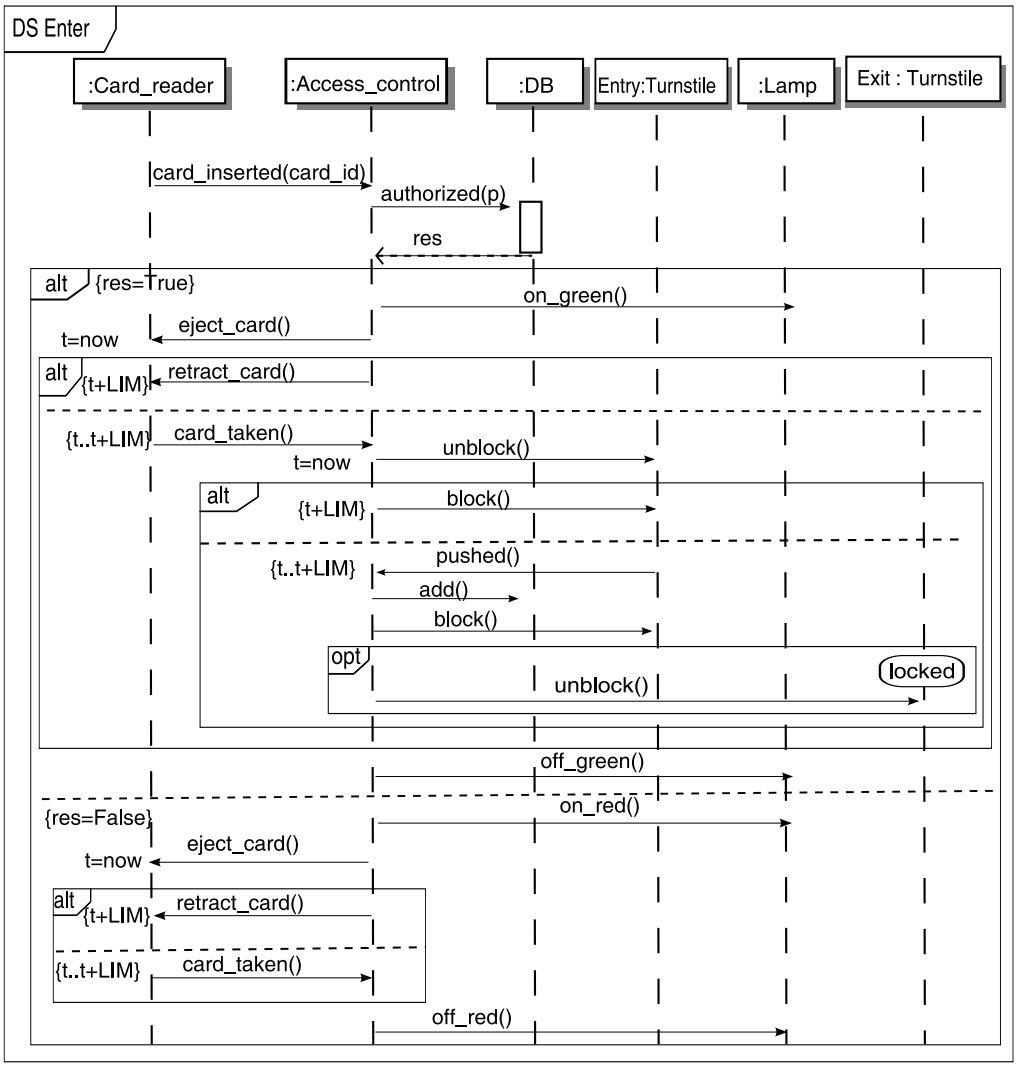


FIGURE 6.11 – Diagramme corrigé de la fonctionnalité Enter

La Figure 6.12 présente l'opération B corrigée associée à la fonctionnalité Enter.

Grâce au prouveur de la méthode B, les expérimentations menées sur l'étude de cas nous ont permis de pointer les erreurs dans les spécifications B. La définition de la typologie des erreurs et d'heuristiques de correction nous ont aidés à corriger les erreurs, permettant d'apporter des corrections pas toujours évidentes. Cela montre l'intérêt et la puissance de la méthode. Cette démarche nécessite toutefois une bonne connaissance de la modélisation de l'assemblage ainsi qu'une expertise dans l'utilisation du prouveur de B, qui bien que supportant les preuves interactives a plutôt été conçu pour effectuer des preuves automatiques. Il apparaît donc nécessaire d'outiller notre méthode non seulement pour effectuer automatiquement les traductions vers B, mais aussi pour avoir un retour assisté des erreurs dans les spécifications UML.

```
UserEnter =
BEGIN
card_inserted ;
res ⟵ authorised ;
IF res = TRUE THEN
   on_green ;
   eject_card;
   ANY xx WHERE xx ∈ NAT ∧ time < MAXINT − xx THEN
      MAJ ( xx )
   END ;
   IF time ≥ t_eject + LIM THEN
      retract_card
   ELSE
        card_taken ;
        Entry.unblock ;
        t_unblock := time ;
        ANY xx WHERE xx ∈ NAT ∧ time < MAXINT − xx THEN
           MAJ ( xx )
        END ;
        IF time < t_unblock + LIM ∧ n_person < MAXINT THEN
           Entry.pushed ;
           enter_i ;
           pp := FALSE ;
           Entry.block;
           IF (Left1.stateT) = T1 THEN
              Left1.unblock
           END
        ELSE
             Entry.block
        END
   END;
   off_green
ELSE
     on_red ;
     eject_card ;
     ANY xx WHERE xx ∈ NAT ∧ time < MAXINT − xx THEN
        MAJ ( xx )
     END;
     IF time >= t_eject + LIM THEN
        retract_card
     ELSE
          card_taken
     END;
     off_red
END
END ;
```

FIGURE 6.12 – Spécification corrigée de la fonctionnalité Enter

6.7 Conclusion

Dans ce chapitre, nous avons proposé une aide au diagnostic et à la correction de spécifications en cours de développement à partir du retour des outils de vérification dans le cadre du développement de spécifications par assemblage de composants existants et la vérification de leur interopérabilité sur le plan des protocoles. L'approche est fondée sur :

– l'utilisation d'UML.2.0 et des systèmes de transitions étiquetés pour l'expression de différents niveaux de spécifications. Les systèmes de transitions sont utilisés pour décrire le comportement des composants, les diagrammes de structure composite pour décrire l'architecture du système et les diagrammes de séquences pour décrire les fonctionnalités en explicitant les interactions entre composants. Un médiateur chargé de synchroniser les com-

munications entre les différentes instances des composants du système, est introduit, toutes les communications passant exclusivement par ce médiateur,
- l'utilisation de B pour la preuve de l'interopérabilité entre composants. L'ensemble des spécifications UML est transformé en B en utilisant des règles de transformation systématiques. Les vérifications à l'aide des outils de preuve de B consistent à démontrer que le comportement, décrit par les interactions entre les composants du système à travers le médiateur, correspond à un raffinement de la spécification abstraite fournie et qu'il n'y a pas de blocage dans ces interactions. Cette vérification repose sur les schémas introduits dans la section 4.4 p. 70.

À partir de l'échec de la preuve et d'une typologie des erreurs liées à notre cadre de travail, celui du développement des spécifications par assemblage de composants, une analyse du retour de la preuve et un diagnostic d'erreurs sont établis. Ils permettent de guider la correction de la spécification du système et de ses fonctionnalités. Les composants utilisés ne sont ni modifiés, ni substitués. Ces propositions sont déduites pour chaque fonctionnalité du système et tiennent compte du type d'erreur.

Certaines corrections nécessitent de revisiter l'ensemble des spécifications. C'est souvent le cas lorsque l'on est amené à modifier un invariant de collage, les opérations du système devant préserver l'invariant, doivent être modifiées en conséquence. La correction d'une fonctionnalité ou d'une partie de l'invariant de collage établissant la relation entre les variables des différentes spécifications peut avoir des répercussions en cascade affectant les différentes parties de la spécification. Une aide peut être apportée sur les modifications à effectuer à partir d'une analyse systématique de l'impact d'une modification sur l'ensemble de la spécification. C'est pour cela que le processus de correction est guidé par quelques stratégies globales proposées. Nous étudions actuellement de nouvelles stratégies permettant d'enchaîner un ensemble de corrections.

7

Conclusion

Sommaire

7.1 Bilan 151

7.2 Perspectives 152

7.2.1 Schémas B 152

7.2.2 Détection des erreurs, analyse et correction des spécifications B 152

7.2.3 Application à l'assemblage des composants 153

Ce mémoire s'achève par un bilan des principaux résultats obtenus et par une description des perspectives.

7.1 Bilan

Dans cette thèse nous avons étudié la vérification et la correction des spécifications B en nous plaçant dans le cadre de l'assemblage des composants. Pour vérifier des propriétés portant sur les protocoles et décrites par des STEs, nous avons mis en évidence des schémas B obtenus par traduction des STEs et basés sur les clauses **REFINES** et **INCLUDES** et nous avons étudié les propriétés de ces schémas en terme de STEs. Nous avons étudié la détection, l'analyse et la correction des erreurs dans les spécifications B, de façon générale en considérant les différentes clauses B et plus particulièrement dans le contexte de notre application en développant des études de cas. Nous avons proposé une traduction des diagrammes UML, utilisés dans notre contexte, en B. La traduction se fait en deux étapes : une étape de formalisation des diagrammes, suivie de la traduction systématique de cette formalisation.

Nos autres contributions concernent l'assemblage des composants, c'est-à-dire comment utiliser la méthode B dans ce contexte et comment partiellement automatiser le processus de vérification. Notre approche est composée de trois étapes successives : (i) la traduction des spécifications en B, (ii) la vérification des spécifications et (iii) la détection et l'analyse des erreurs et la correction des spécifications. Nous avons étudié cette approche dans un premier temps pour l'assemblage et l'adaptation de deux interfaces requise et fournie, en considérant les trois niveaux d'interopérabilité, syntaxique, sémantique et protocole, ensuite en traitant le problème de l'assemblage et la coordination de plusieurs composants par l'intermédiaire d'un médiateur en considérant les niveaux syntaxique et protocole.

7.2 Perspectives

Nous distinguons les perspectives relatives à chacune des contributions de cette thèse.

7.2.1 Schémas B

Notre travail peut être étendu en introduisant de nouveaux schémas B de façon à modéliser de nouvelles relations entre STEs. Nous avons commencé à réfléchir à la relation modélisée par les schémas de la Figure 7.1. Dans ces deux schémas, nous considérons une relation d'étiquettes plus élaborée entre deux STEs. Cette relation permet de lier deux compositions d'étiquettes. Notre objectif est d'analyser les POs générées pour vérifier ces schémas et déduire ensuite les propriétés préservées. Cette extension constitue un objectif à court terme.

Ces nouveaux schémas peuvent par exemple être appliqués pour étendre les cas d'adaptations entre deux interfaces requise et fournie, en considérant de nouvelles correspondances entre elles (*e.g.* correspondre plusieurs opérations requises à plusieurs opérations fournies).

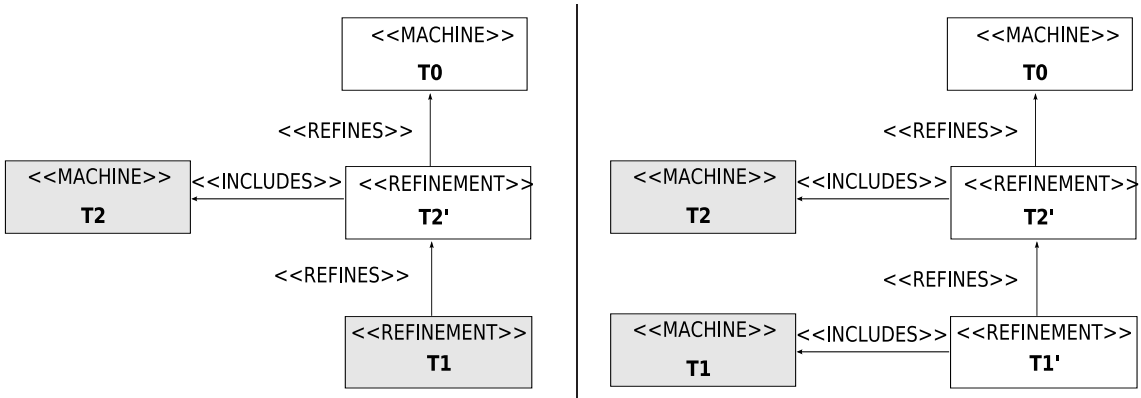


FIGURE 7.1 – Construction B

Une autre extension possible de notre travail est de considérer des STEs qui incluent des données telles que les STEs symboliques. Dans le chapitre 5, nous avons considéré des données avec des STEs pour décrire les interfaces des composants. Néanmoins, nous avons pris une hypothèse assez forte qui consiste à considérer des données et des protocoles indépendants. Il nous semble intéressant d'étendre ce travail en éliminant cette hypothèse. Ce travail demande une réflexion conséquente sur la traduction à effectuer et les relations à modéliser.

7.2.2 Détection des erreurs, analyse et correction des spécifications B

L'approche proposée pour détecter les erreurs, analyser et corriger les spécifications B peut être complétée de différentes manières :

- Le processus de correction des spécifications est indéterministe et par conséquent l'utilisation des stratégies de correction est nécessaire pour garantir le bon déroulement de cette étape. Actuellement, peu de stratégies sont proposées dans notre travail pour guider l'étape de correction. Pour améliorer l'efficacité de notre approche, il serait nécessaire de l'enrichir avec de nouvelles stratégies.
- Nous nous sommes focalisés dans ce travail sur les retours du prouveur de l'atelier B afin de détecter les erreurs, analyser et corriger les spécifications B. Cependant, il existe

d'autres outils permettant de vérifier les spécifications B. Plus particulièrement, il nous semble intéressant d'exploiter l'échec de la vérification retourné par ProB (*e.g.* les contre exemples retournés) pour corriger les spécifications B. Il nous semble également intéressant de combiner le retour de l'échec de preuve de l'atelier B et l'échec de vérification retourné par ProB. L'utilisation conjointe de ces deux outils est déjà considérée en pratique pour guider la construction et la vérification des spécifications B.
- L'étude de correction des spécifications B peut être appliquée à d'autres domaines que la vérification des assemblages de composants. Nous pouvons ainsi penser à de nouveaux domaines d'application.

7.2.3 Application à l'assemblage des composants

Nous énumérons une liste d'extensions relatives aux approches d'assemblage de composants précédemment étudiées.

Description des interfaces

Les aspects suivants concernent les interfaces des composants :
- La description des interfaces dans l'approche d'assemblage de plusieurs composants ne fait pas intervenir le niveau sémantique. Une solution à court terme pour considérer la sémantique des interfaces consiste à décrire la sémantique des interfaces et à vérifier leur compatibilité indépendamment du protocole de la même manière que pour l'assemblage de deux interfaces. Cette restriction est relativement forte et il serait intéressant de réfléchir à une autre modélisation.
- Niveaux de hiérarchie. Une extension intéressante des interfaces est de les décrire de manière hiérarchique. Ceci nous permet d'exploiter la hiérarchie des interfaces pour tester des propriétés par niveau et ainsi simplifier leur vérification.
  Nous avons commencé, à travers l'étude de cas présenté dans la section C.2 de l'annexe C, à traiter des interfaces hiérarchiques. Dans cette étude de cas nous avons procédé comme suit. Nous considérons initialement la vérification de l'assemblage de plusieurs composants en raffinant une spécification abstraite du système considéré. À ce niveau, les composants du système sont considérés comme des boîtes noires et leurs fonctionnalités sont décrites à travers des interfaces. Nous reprenons ensuite la spécification des interfaces des composants qui sont eux mêmes composés par des composants internes. Nous vérifions l'assemblage des composants internes de chaque composant en raffinant la description de son interface. La vérification de chaque niveau est ainsi effectuée indépendamment des autres jusqu'à arriver à des composants basiques (*i.e.* ils ne contiennent pas de structure interne). L'approche suivie dans cette étude de cas peut être facilement généralisée.
- Nous pouvons également envisager de tenir compte dans la description des interfaces d'autres aspects telle que la qualité de service (QoS).

Description des coordinations

Nous avons utilisé les diagrammes de séquences pour la description de la coordination entre les composants du système. Nous avons restreint notre étude à un sous-ensemble de diagrammes de séquences d'UML 2.0. Nous travaillons à une généralisation permettant de prendre en compte l'ensemble des constructions des diagrammes de séquences avec notamment les fragments combinés correspondant à des interactions parallèles et les symboles de continuation ou label correspondant à une manière de définir des branchements.

Vérification des propriétés

L'extension de la description des interfaces entraîne la nécessité de définir de nouvelles propriétés entre les interfaces des composants pour vérifier l'assemblage. Par conséquent, il ne suffit pas de réfléchir comment ces extensions vont être décrites mais il faut aussi identifier les propriétés qui doivent être préservées, afin de garantir la vérification de l'assemblage.

Dans le processus de vérification de l'assemblage, nous nous sommes basés principalement sur la vérification du raffinement B pour vérifier certaines propriétés dans un assemblage de composants. L'avantage de cette approche est que la vérification de l'assemblage se fait d'une manière automatique, cependant nous sommes limités aux propriétés du raffinement B. Nous pouvons proposer comme perspective d'étudier d'autres modélisations B permettant de considérer de nouvelles propriétés.

Correction des incompatibilités dans un assemblage

Si nous envisageons d'étendre la description des interfaces et des propriétés à tester dans l'assemblage, nous allons alors rencontrer de nouveaux types d'incompatibilités à résoudre. Par conséquent, nous devons réfléchir à de nouveaux types d'adaptations permettant de traiter ces cas.

Dans les deux approches d'assemblage étudiées, nous avons considéré la correction des spécifications respectivement de l'adaptateur et du médiateur. Une évolution intéressante de ce travail consiste à prendre en compte la remise en cause des composants utilisés dans l'assemblage.

Études de cas

Jusqu'à présent, nous avons seulement appliqué les approches proposées à des spécifications académiques plus ou moins classiques. En effet, nous avons utilisé pour la première approche d'assemblage de deux interfaces les spécifications des lampes et d'un lecteur de CD. Quant à la seconde approche d'assemblage de plusieurs composants, nous avons utilisé les spécifications d'un système de contrôle d'accès et des feux d'un carrefour. La prochaine étape est d'appliquer notre approche à d'autres études de cas ainsi qu'à des systèmes industriels réalistes basés sur des modèles de composants logiciels usuels tels que Corba ou .NET. Le deuxième cas d'application nécessite une étude approfondie à la fois des modèles composants existants et des correspondances entre ces modèles et notre approche. Nous pensons également que de futures études de cas vont fournir des informations quant à l'applicabilité de nos approches d'assemblage et qu'elles nous aideront à les améliorer et particulièrement à étendre les stratégies proposées pour la correction des spécifications.

Implantation

Plusieurs cas d'assemblage sont étudiés dans notre travail. Les aides à la correction de spécifications sont actuellement proposées pour des spécifications B obtenues par transformation systématique des spécifications de départ exprimées en UML. L'approche proposée nécessite d'être outillée.

En effet, les différentes étapes de l'approche que nous proposons peuvent être complètement ou partiellement automatisées afin de fournir un environnement pour la vérification et la correction semi-automatique de l'assemblage des composants. L'implantation peut toucher les points suivants :

- Nous utilisons des règles systématiques pour la traduction des STEs et des diagrammes UML en B, mais toutes ces traductions sont faites à la main. Il existe différents prototypes de traduction de UML vers B et de B vers UML [IL06] ainsi que des travaux permettant de faire évoluer conjointement des spécifications UML et B [OOSJ05]. En revanche, ces différents prototypes ont essentiellement servi à valider les idées développées et sont d'une utilisation restreinte. En outre, nous avons vu en Section 4.5 p. 85, que diverses approches ont été proposées pour associer une description en terme de STEs à des modèles B, en proposant des outils [BPS05, LB08].
  Nous envisageons donc de développer un outil pour supporter la traduction en B. Il nous semble également intéressant de coupler les outils existants, qui ont considéré la traduction de B vers UML ou de B vers STEs, avec notre approche afin de parvenir à un outil de vérification et de correction multi-vues.
- Les schémas étudiés peuvent être considérés comme des patterns. Par conséquent, nous pouvons développer une bibliothèque de ces schémas ainsi que des mécanismes pour guider le concepteur dans le choix de l'un d'entre eux, selon le problème d'assemblage qu'il considère. Si le contexte est la construction d'un adaptateur entre deux interfaces, nous pouvons guider le concepteur dans la construction de la spécification de l'adaptateur. Cette construction se fait en sélectionnant le schéma approprié et en effectuant ensuite les traductions en B et finalement en ajoutant la traduction des propriétés nécessaires pour réussir la vérification de l'assemblage.
- Nous pouvons également considérer l'étape de la correction des spécifications en implémentant partiellement les stratégies de correction. Les corrections proposées s'appuient sur des mécanismes de recherche de chemins ou de recherche d'opérations en respectant des conditions d'appariement. Nous pouvons envisager d'implémenter ces mécanismes de recherche ou faire un couplage avec des outils existants qui traitent ce genre de problème.

Bibliographie

[AAA06] P. André, G. Ardourel, and C. Attiogbé. Spécification d'architectures en kmelia : hiérarchie de connexion et composition. In *1ère Conférence Francophone sur les Architectures Logicielles*, pages 101–118. Hermes - Lavoisier, 2006.

[AAA07] P. André, G. Ardourel, and C. Attiogbé. Adaptation for hierarchical components and services. *Electr. Notes Theor. Comput. Sci.*, 189 :5–20, 2007.

[Abr96a] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[Abr96b] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *First B conference, Putting into practice Methods and Tools for Information System Design*, pages 169–190, 1996.

[Abr98] J.R. Abrial. Etude Système : méthode et exemple. Technical report, ClearSy System Engineering, Octobre 1998.

[Abr10] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[AC03] J.-R. Abrial and D. Cansell. Click'n'Prove : Interactive proofs within set theory. In D. Basin and B. Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *LNCS*, pages 1–24. Springer-Verlag, 2003.

[Ada07] Adaptor. Adaptor, december 2007 distribution (lgpl licence), 2007. `http://adaptorweb.lcc.uma.es/`.

[AFA00] AFADL. Étude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. actes LSR/IMAG.

[Ame91] P. America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90. Springer-Verlag, 1991.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, Paris, 1992.

[Arn94] A. Arnold. *Finite transition systems*. Prentice-Hall, 1994.

[B-C96] B-Core(UK) Ltd. *B-Toolkit User's Manual, Release 3.2*, 1996.

[BA05] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project : Roissy VAL. In *Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 334–354. Springer-Verlag, 2005.

[BBC05] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1) :45–54, 2005.

[BBG+06] S. Becker, A. Brogi, I. Gorton, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. *Architecting Systems with Trustworthy Components*, 3938 :193–215, 2006.

[BBM99] P. Behm, P. Benoit, and J. M. Meynadier. METEOR : A successful application of B in a large project. In *Integrated Formal Methods*, volume 1708 of *LNCS*, pages 369–387. Springer Verlag, 1999.

[BC00] D. Bert and F. Cave. Construction of finite labelled transistion systems from b abstract systems. In *IFM '00 : Proceedings of the Second International Conference on Integrated Formal Methods*, pages 235–254. Springer-Verlag, 2000.

[BCG+05] B. Benatallah, F. Casati, D. Grigori, H. R. Motahari Nezhad, and F. Toumani. Developing adapters for web services integration. In *17th International Conference on Advanced Information System Engineering (CAiSE 2005)*, pages 415–429. Springer, 2005.

[BCP06] A. Brogi, C. Canal, and E. Pimentel. Component adaptation through flexible subservicing. *Science of Computer Programming*, 63 :39–56, 2006.

[BCT04] B. Benatallah, F. Casati, and F. Toumani. Analysis and management of web services protocols. In *ER 2004*, November 2004.

[BIM95] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1) :232–268, January 1995.

[BJK00] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In *Fundamental Aspects of Software Engineering (FASE'00)*, volume 1783 of *LNCS*, pages 266–283. Springer Verlag, 2000.

[BLLS08] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour Rodin. *Technique et Science Informatiques*, 27(8) :1065–1084, 2008.

[BPS05] D. Bert, M.-L. Potet, and N. Stouls. Genesyst : a tool to reason about behavioral aspects of b event specifications. application to security properties. In *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2) :323 342, 1983.

[CFP+03] C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to corba objects. *Software Engineering, IEEE Transactions on*, 29(3) :242–260, March 2003.

[CHS06] S. Chouali, M. Heisel, and J. Souquières. Proving component interoperability with B refinement. *Electronic Notes in Theoretical Computer Science*, 160 :157–172, 2006.

[Cle08a] Clearsy. *Atelier B–Interactive Prover Reference Manual, version 3.7*, 2008.

[Cle08b] Clearsy. *Atelier B–Interactive Prover Reference User, version 3.7*, 2008.

[Cle08c] Clearsy. *Atelier B 4.0 User manual*, 2008.

[Cle08d] Clearsy. B4free. *website,* `http://www.clearsy.com/`, 2008.

[CLS07] S. Colin, A. Lanoix, and J. Souquières. Trustworthy interface compliancy : data model adaptation. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite workshop of ETAPS*, March 2007.

[CM06] D. Cansell and D. Méry. *Tutorial on the event-based B method*, 2006. `http://cel.archives-ouvertes.fr/inria-00092846/en/`.

[CMP06] C. Canal, J. M. Murillo, and P. Poizat. Software adaptation. *L'Objet*, 12(1) :9–31, 2006.

[CPR00] C. Choppy, P. Poizat, and J.-C. Royer. A global semantics for views. In *AMAST '00 : Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 165–180. Springer-Verlag, 2000.

[CPS06] C. Canal, P. Poizat, and G. Salaün. Synchronizing behavioural mismatch in software composition. In *8th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems FMOODS'2006*, volume 4037 of *LNCS*, pages 63–77. Springer Verlag, June 2006.

[CPS08] C. Canal, P. Poizat, and G. Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4) :546–563, 2008.

[dAH01] L. de Alfaro and T.A. Henzinger. Interface automata. In *9th Annual Aymposium on Foundations of Software Engineering, FSE*, pages 109–120. ACM Press, 2001.

[DC05] S. Dunne and S. Conroy. Process refinement in B. In *ZB 2005 : Formal Specification and Development in Z and B*, pages 45–64. Springer Berlin, 2005.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DSW06] M. Dumas, M. Spork, and K. Wang. Adapt or perish : Algebra and visual notation for service interface adaptation. In *International Conference on Business Process Management*, pages 65–80. Springer-Verlag, September 2006.

[Dun03] S. Dunne. Introducing backward refinement into b. In *ZB 2003 : Formal Specification and Development in Z and B*, pages 178–196. Springer Berlin, 2003.

[GR01] R. Gorrieri and A. Rensink. *Action refinement*, chapter 16, pages 1047–1147. Elsevier, j. bergstra, a. ponse, and s. smotka eds. edition, 2001.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8 :231–274, 1987.

[HC01] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[Hem05] D. Hemer. A formal approach to component adaptation and composition. In *ACSC '05 : Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 259–266. Australian Computer Society, Inc., 2005.

[HHS06] D. Hatebur, M. Heisel, and J. Souquières. A method for component-based software and system development. In *Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications*, pages 72–80. IEEE Computer Society, 2006.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580 & 583, 1969.

[IL06] A. Idani and Y. Ledru. Dynamic graphical UML views from formal B specifications. *International Journal of Information and Software Technology*, 48(3) :154–169, 2006. Elsevier.

[IT03] P. Inverardi and M. Tivoli. Software architecture for correct components assembly. In *SFM*, pages 92–121. Springer-Verlag, 2003.

[Kon95] D. Konstantas. Interoperation of object oriented application. In *Object-Oriented Software Composition*, pages 69–95. Prentice Hall, 1995.

[LB03] M. Leuschel and M. Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[LB05] M. Leuschel and M. Butler. Automatic refinement checking for b. In *ICFEM*, pages 345–359, 2005.

[LB08] M. Leuschel and M. Butler. Prob : an automated analysis toolset for the b method. *STTT*, 10(2) :185–203, 2008.

[LCS07] A. Lanoix, S. Colin, and J. Souquières. Schémas de développement d'adaptateurs à l'aide de B. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'07)*, pages 91–108, June 2007.

[Led02] H. Ledang. *Traduction systématique de spécifications UML vers B.* Thèse d'université, LORIA - Nancy 2, 2002.

[LHHS07] A. Lanoix, D. Hatebur, M. Heisel, and J. Souquières. Enhancing dependability of component-based systems. In Springer Verlag, editor, *Reliable Software Technologies Ada-Europe 2007*, number 4498 in LNCS, pages 41–54. Springer-Verlag, 2007.

[LS06] M. S. Lund and K. Stolen. A fully operational semantics for uml 2.0 sequence diagrams with potential and mandatory choice. In *14th International Symposium on Formal Methods*, number 4085 in LNCS, pages 380–395. Springer-Verlag, 2006.

[LW94] B. H. Liskov, , and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 :1811–1841, 1994.

[MA03] B. Morel and P. Alexander. Automating component adaptation for reuse. In *18th IEEE International Conference on Automated Software Engineering(ASE'03)*, pages 142–151, 2003.

[MA09] I. Mouakher and F. Alexandre. Raffinement B de systèmes de transitions étiquetés. Research Report inria-00435896, LORIA, 2009. `http://hal.inria.fr/inria-00435896/en/`.

[Mey97] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[Mic95] Microsoft Corporation. *The Component Object Model Specification, Version 0.9*, 1995. http ://www.microsoft.com/com/resources/comdocs.asp.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[MLS06] I. Mouakher, A. Lanoix, and J. Souquières. Component adaptation : Specification and verification. In *Proc. of the 11th Int. Workshop on Component Oriented Programming, satellite workshop of ECOOP*, pages 23–30, 2006.

[MNBM+07] H. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW '07 : Proceedings of the 16th international conference on World Wide Web*, pages 993–1002. ACM, 2007.

[MSA08a] I. Mouakher, J. Souquières, and F. Alexandre. Protocol Verification in a Software Component-Based Approach. In *15th IEEE International Conference on Engineering of Computer-Based Systems*, pages 136–145. IEEE Computer Society, 2008.

[MSA08b] I. Mouakher, J. Souquières, and F. Alexandre. Diagnostic et correction d'erreurs de spécifications : application à l'assemblage de composants. *RTSI - L'Objet*, 14 :11–42, 2008. Numéro spécial Composents, services et aspects.

[Obj98] The Object Mangagement Group (OMG). *The Common Object Request Broker : Architecture and Specification, Revision 2.2*, February 1998. http ://cgi.omg.org/library/corbaiiop.html.

[OMG05] Object Management Group (OMG). *UML Superstructure Specification*, 2005. version 2.0.

[OOSJ05] D.-D. Okalas Ossami, J. Souquières, and J.-P. Jacquot. Consistency in UML and B multi-view specifications. In *Proc. of the Int. Conf. on Integrated Formal Methods*, number 3771 in LNCS, pages 386–405. Springer-Verlag, 2005.

[PdAHSV02] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis : Two faces of the same coin. In *International Conference on Computer Aided Design 2002*, November 2002.

[Pot03] M.-L. Potet. Spécifications et développements structurés dans la méthode b. *Technique et Science Informatiques, RSTI série TSI*, 22(1) :61–88, 2003.

[PS04] M.-L. Potet and N. Stouls. Explicitation du contrôle de développement B événementiel. In J. Julliand, editor, *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'04)*, pages 13–27, JUN 2004.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Software Eng*, 28(11), November 2002.

[Reu03] R. H. Reussner. Automatic component protocol adaptation with the coconut/j tool suite. *Future Gener. Comput. Syst.*, 19(5) :627–639, 2003.

[Sch02] S. Schneider. *The B-Method : An Introduction*. Palgrave, 2002.

[SG01] B. Spitznagel and D. Garlan. A compositional approach for constructing connectors. *Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, pages 148–157, 2001.

[SG03] B. Spitznagel and D. Garlan. A compositional formalization of connector wrappers. In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 374–384. IEEE Computer Society, may 2003.

[SP05] G. Salaün and P. Poizat. Interacting extended state diagrams. *Electronic Notes in Theoretical Computer Science*, 115 :49–57, 2005. Proceedings of the Second Workshop on Semantic Foundations of Engineering Design Languages (SFEDL 2004).

[Spi04] B. Spitznagel. *Compositional transformation of software connectors*. PhD thesis, Carnegie Mellon University, 2004.

[SR02] H. W. Schmidt and R. H. Reussner. Generating adapters fo concurrent component protocol synchronisation. In I. Crnkovic, S. Larsson, and J. Stafford, editors, *Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 213–229, 2002.

[Sun97] Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http ://-java.sun.com/products/javabeans/docs/ spec.html.

[Sun01] Sun Microsystems. *Enterprise JavaBeans Specification*, 2001. Version 2.0.

[SYN] SYNTHESIS. `http://www.di.univaq.it/tivoli/SYNTHESIS/synthesis.html`.

[Szy99] C. Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1999.

[TA06] M. Tivoli and M. Autili. Synthesis, a tool for synthesizing correct and protocol-enhanced adaptors. *L'OBJET*, 12 :77–103, 2006.

[Tiv05] M. Tivoli. *An architectural approach to the automatic composition and adaptation of software components*. PhD thesis, L'Aquila University, 2005.

[Tru06] Ninh Thuan Truong. *Utilisation de B pour la vérification de spécifications UML et le développement formel orienté objet*. PhD thesis, Université Nancy 2, LORIA, May 2006.

[TS06] N. T. Truong and J. Souquières. Validation of UML scenario using the B prover. In *Proceeding of the TFIT, Third Taiwanese-French Conference on Information Technology*, 2006.

[VHT99] A. Vallecillo, J. Hernandez, and J. M. Troya. Object interoperability. In *Object Oriented Technology : ECOOP'99 Workshop Reader*, pages 1–21, 1999.

[VHT00] A. Vallecillo, J. Hernández, and J. M. Troya. Component interoperability. Technical Report ITI-2000-37, University of Málaga, july 2000. `http://www.lcc.uma.es/~av/Publicaciones/00/Interoperability.pdf`.

[Weg96] P. Wegner. Interoperability. *ACM Computing Survey*, 28(1) :285–287, 1996.

[YS97] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

[ZW95] A. M. Zaremski and J. M. Wing. Signature matching : a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2) :146–170, 1995.

[ZW97] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transaction on Software Engeniering Methodolology*, 6(4) :333–369, 1997.

Acronymes

$\mathcal{WP}$ Weakest precondition. 11

ADL Architecture description language. 50

AMN Abstract Machine Notation. 10

BIDL Behavioural Interface Description Language. 36

bMSC basic Message Sequence Chart. 42

BPEL Business Process Execution Language. 45

CBSE Component-Based Software Engineering. 1

CORBA Common Object Request Broker Architecture. 3

COTS Components off the shelf. 3

eLTS extended labelled transition system. 43

FSM Finite state machine. 39

FSP Finite state processes. 50

HBIDL Hierarchical Behavioural Interface Description Language. 43

hMSC high level message sequence chart. 42

IDL Interface description language. 35

IDM Interface data model. 122

LTL Linerar-time Temporal Logic. 42

PI Provided interface. 87

PO Proof obligation. 2, 10, 19

POO Programmation orientée objet. 34

QoS Quality of service. 3, 35

RI Required interface. 87

SOA Services oriented architecture. 34, 44

STE Système de transitions étiqueté. 90

UML Unified Modelling Language. 3

WID WebSphere Integration Developer. 47

WSDL Web services description language. 44

WSOA Web services oriented architecture. 44

XML Extensible markup language. 44

A

Prouveur de l'Atelier B

A.1 Quelques formes normales

Expression	Forme normale
$n>m-1$	$m\leq n$
$m<n-1$	$m\leq n$
$a\Leftrightarrow b$	$a\Rightarrow b \wedge b\Rightarrow a$
$a\subseteq b$	$a\in\mathbb{P}(b)$
$a\subset b$	$a\in\mathbb{P}(b)\wedge \neg(a=b)$
$a\notin b$	$\neg(a\in b)$
$a\neq b$	$\neg(a=b)$
$a\not\subseteq b$	$\neg(a\in\mathbb{P}(b))$
$a\not\subset b$	$a\in\mathbb{P}(b)\Rightarrow a = b$
$a\in\mathbb{N}$	$a\in\mathbb{Z}\wedge 0\leq a$
$\mathbb{N}_1$	$\mathbb{N}-\{0\}$
NAT1	NAT$-\{0\}$
FIN1(A)	FIN(A)$-\{\varnothing\}$
$\mathbb{P}_1(A)$	$\mathbb{P}(A)-\{\varnothing\}$
seq1(A)	seq(A)$-\{\varnothing\}$
iseq1(A)	iseq(A)$-\{\varnothing\}$
perm(E)	iseq(E) $\cap$ ($\mathbb{N}-0 \twoheadrightarrow$E)
{x,y}	{x}$\cup$ {y}
{x|P}	SET(x).P

TABLE A.1 – Certaines formes normales

A.2 POs générées pour l'exemple 4.4.1

```
aa =
PRE stateT1 ∈{p0,p1,p6,p10} THEN
    SELECT stateT1 = p0 THEN stateT1 :∈{p2,p3}
    WHEN stateT1 = p1 THEN stateT1 :∈{p4,p5}
    WHEN stateT1 = p6 THEN stateT1 :∈{p7,p8,p9}
    WHEN stateT1 = p10 THEN stateT1 :∈{p11,p12}
    WHEN  stateT1 = p13 THEN stateT1 :∈{p14,p15}
    END
END
```

(a) Abstrait

```
aa =
PRE stateT2 ∈{q0,q2,q5,q7,q11,q13} THEN
    SELECT stateT2 = q0 THEN stateT2 :=  q1
    WHEN stateT2 = q2 THEN stateT2 :∈{q3,q4}
    WHEN stateT2 = q5 THEN stateT2 :=q6
    WHEN stateT2 = q7 THEN stateT2 :∈{q8,q9,q10}
    WHEN stateT2 = q11 THEN stateT2 := q12
    WHEN stateT2 = q13 THEN stateT2 :∈{q14,q15}
    END
END
```

(b) Raffinement

```
PROPERTIES
    RR ∈ P1 ↔ P2 ∧
    RR = {(p0 ↦q0), (p1 ↦ q0), (p2 ↦q1), (p5 ↦ q1),
        (p6 ↦q2), (p6 ↦ q5), (p7 ↦q3), (p8 ↦ q4),
        (p8 ↦ q6), (p10 ↦ q7), (p11 ↦ q8), (p11 ↦ q9),
        (p11 ↦ q10), (p16 ↦ q13),...}
```

(c) Relation entre états

FIGURE A.1 – Rffinement d'une étiquette *aa*

PO1

"'Check precondition (stateT2∈ {q0,q2,q5,q7,q11,q13}) deduction'"
⇒ stateT2$1∈ {q0,q2,q5,q7,q11,q13}

PO2

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'" stateT2$1 = q0 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
"'Check operation refinement - ref 4.4, 5.5'"
⇒ stateT1 = p0

PO3

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$1 = q0 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q1∈ P2 ∧ stateT1$0↦q1∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
"'Check operation refinement - ref 4.4, 5.5'"

$\Rightarrow \exists(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p2},\mathsf{p3}\} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{q1} \in \mathsf{RR})$

PO4

"'aa preconditions in this component'"
$\mathsf{stateT2\$1} \in \{\mathsf{q0},\mathsf{q2},\mathsf{q5},\mathsf{q7},\mathsf{q11},\mathsf{q13}\} \wedge \mathsf{stateT1} \in \{\mathsf{p0},\mathsf{p1},\mathsf{p6},\mathsf{p10}\} \wedge$
"'Local hypotheses'"
$\mathsf{stateT2\$0} \in \{\mathsf{q3},\mathsf{q4}\} \wedge$
$\mathsf{stateT2\$1} = \mathsf{q2} \wedge$
$\mathsf{stateT1} = \mathsf{p13} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p14},\mathsf{p15}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p10} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p11},\mathsf{p12}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p6} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p7},\mathsf{p8},\mathsf{p9}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p1} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p4},\mathsf{p5}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
"'Check that the invariant (stateT1|->stateT2 : RR) is preserved by the operation - ref 4.4, 5.5'"
"'Check operation refinement - ref 4.4, 5.5'"
$\Rightarrow \mathsf{stateT1} = \mathsf{p0}$

PO5

"'aa preconditions in this component'"
$\mathsf{stateT2\$1} \in \{\mathsf{q0},\mathsf{q2},\mathsf{q5},\mathsf{q7},\mathsf{q11},\mathsf{q13}\} \wedge \mathsf{stateT1} \in \{\mathsf{p0},\mathsf{p1},\mathsf{p6},\mathsf{p10}\} \wedge$
"'Local hypotheses'"
$\mathsf{stateT2\$0} \in \{\mathsf{q3},\mathsf{q4}\} \wedge$
$\mathsf{stateT2\$1} = \mathsf{q2} \wedge$
$\mathsf{stateT1} = \mathsf{p13} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p14},\mathsf{p15}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p10} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p11},\mathsf{p12}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p6} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p7},\mathsf{p8},\mathsf{p9}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p1} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p4},\mathsf{p5}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
"'Check that the invariant ($\mathsf{stateT2} \in \mathsf{P2}$) is preserved by the operation - ref 4.4, 5.5'"
$\Rightarrow \mathsf{stateT2\$0} \in \mathsf{P2}$

PO6

"'aa preconditions in this component'"
$\mathsf{stateT2\$1} \in \{\mathsf{q0},\mathsf{q2},\mathsf{q5},\mathsf{q7},\mathsf{q11},\mathsf{q13}\} \wedge \mathsf{stateT1} \in \{\mathsf{p0},\mathsf{p1},\mathsf{p6},\mathsf{p10}\} \wedge$
"'Local hypotheses'"
$\mathsf{stateT2\$0} \in \{\mathsf{q3},\mathsf{q4}\} \wedge$
$\mathsf{stateT2\$1} = \mathsf{q2} \wedge$
$\mathsf{stateT1} = \mathsf{p13} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p14},\mathsf{p15}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p10} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p11},\mathsf{p12}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p6} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p7},\mathsf{p8},\mathsf{p9}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
$\mathsf{stateT1} = \mathsf{p1} \Rightarrow \forall(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p4},\mathsf{p5}\} \Rightarrow \neg(\mathsf{stateT2\$0} \in \mathsf{P2} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})) \wedge$
"'Check that the invariant (stateT1|->stateT2 : RR) is preserved by the operation - ref 4.4, 5.5'"
$\Rightarrow \exists(\mathsf{stateT1\$0}).(\mathsf{stateT1\$0} \in \{\mathsf{p2},\mathsf{p3}\} \wedge \mathsf{stateT1\$0} \mapsto \mathsf{stateT2\$0} \in \mathsf{RR})$

PO7

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10}
"'Local hypotheses'"
stateT2$1 = q5 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT1 = p0

PO8

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧ "'Local hypotheses'"
stateT2$1 = q5 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q6∈ P2 ∧ stateT1$0↦q6∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ ∃(stateT1$0).(stateT1$0∈ p2,p3 ∧ stateT1$0↦q6∈ RR)

PO9

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$0∈ {q8,q9,q10} ∧
stateT2$1 = q7 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT1 = p0

PO10

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$0∈ {q8,q9,q10} ∧
stateT2$1 = q7 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧

stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR))
"'Check that the invariant (stateT2 : P2) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT2$0∈ P2

PO11

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$0∈ {q8,q9,q10} ∧
stateT2$1 = q7 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ ∃(stateT1$0).(stateT1$0∈ {p2,p3} ∧ stateT1$0↦stateT2$0∈ RR)

PO12

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$1 = q11 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR))∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR))∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT1 = p0

PO13

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$1 = q11 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(q12∈ P2 ∧ stateT1$0↦q12∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ ∃(stateT1$0).(stateT1$0∈ {p2,p3} ∧ stateT1$0↦q12∈ RR)

PO14

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"

stateT2$0∈ {q14,q15} ∧
stateT2$1 = q13 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
"'Check that the invariant (stateT1|->stateT2 : RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT1 = p0

PO15

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$0∈ {q14,q15} ∧
stateT2$1 = q13 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
"'Check that the invariant (stateT2∈ P2) is preserved by the operation - ref 4.4, 5.5'"
⇒ stateT2$0∈ P2

PO16

"'aa preconditions in this component'"
stateT2$1∈ {q0,q2,q5,q7,q11,q13} ∧ stateT1∈ {p0,p1,p6,p10} ∧
"'Local hypotheses'"
stateT2$0∈ {q14,q15} ∧
stateT2$1 = q13 ∧
stateT1 = p13 ⇒ ∀(stateT1$0).(stateT1$0∈ {p14,p15} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR))∧
stateT1 = p10 ⇒ ∀(stateT1$0).(stateT1$0∈ {p11,p12} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR))∧
stateT1 = p6 ⇒ ∀(stateT1$0).(stateT1$0∈ {p7,p8,p9} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR))∧
stateT1 = p1 ⇒ ∀(stateT1$0).(stateT1$0∈ {p4,p5} ⇒ ¬(stateT2$0∈ P2 ∧ stateT1$0↦stateT2$0∈ RR)) ∧
"'Check that the invariant (stateT1↦stateT2∈ RR) is preserved by the operation - ref 4.4, 5.5'"
⇒ ∃(stateT1$0).(stateT1$0∈ {p2,p3} ∧ stateT1$0↦stateT2$0∈ RR)

B

Spécifications B des exemples du chapitre 5

B.1 Les spécifications B des lampes

```
REFINEMENT
   newPI_GRlight
REFINES
   RI_GRlight
SEES
   ctx
VARIABLES
   lightG  ,
   lightR  ,
   piGRState
INVARIANT
   piGRState ∈ Light_States ∧
   piGRState ∈ {off, greenOn, redOn, greenRedOn}∧
   lightG  ∈𝔹 ∧
   lightR  ∈𝔹 ∧
   piGRState = riGRState
INITIALISATION
   lightG  , lightR  , piGRState := FALSE, FALSE, off
OPERATIONS
   onGreen =
   PRE piGRState ∈{off, redOn} THEN
       SELECT piGRState ∈{off, redOn} THEN
           lightG  := TRUE ||
           IF piGRState = off THEN
               piGRState := greenOn
           ELSE  piGRState := greenRedOn
           END
       END
   END;

   offGreen =
     PRE piGRState ∈{greenOn, greenRedOn} THEN
         SELECT piGRState ∈{greenOn, greenRedOn} THEN
             lightG  := FALSE ||
             IF piGRState = greenOn THEN
                 piGRState := off
             ELSE  piGRState := redOn
             END
         END
     END;

   onRed =
   PRE piGRState ∈ {off,greenOn} THEN
       SELECT piGRState ∈ {off,greenOn} THEN
           lightR  := TRUE ||
           IF piGRState = off THEN
               piGRState := redOn
           ELSE  piGRState := greenRedOn
           END
       END
   END;

   offRed =
   PRE piGRState ∈{redOn, greenRedOn} THEN
       SELECT piGRState ∈{redOn, greenRedOn} THEN
           lightR  := FALSE ||
           IF piGRState = redOn THEN
               piGRState := off
           ELSE  piGRState := greenOn
           END
       END
   END
END
```

FIGURE B.1 – Raffinement B newPI_GRlight

```
MACHINE
   PI_Mlight
SEES
   ctx
ABSTRACT_CONSTANTS
   Light_StatesPIM
PROPERTIES
   Light_StatesPIM ⊆ Light_States ∧
    Light_StatesPIM = {off,greenOn,redOn,orangeOn}
VARIABLES
   lightG , lightR , lightO , color , piMState
INVARIANT
   lightG ∈ 𝔹 ∧
   lightR ∈ 𝔹 ∧
   lightO ∈ 𝔹 ∧
   color ∈ Colors ∧
   piMState ∈ Light_StatesPIM
INITIALISATION
   lightG , lightR , lightO := FALSE, FALSE,FALSE||
   piMState, color := off ,red
OPERATIONS
   onGreen1 =
   PRE piMState = off ∧ color = green THEN
      SELECT piMState = off ∧ color = green THEN
         lightG := TRUE || piMState := greenOn
      END
   END;

   offGreen =
   PRE piMState = greenOn THEN
      SELECT piMState = greenOn THEN
         lightG := FALSE || piMState := off
      END
   END;

   onRed1 =
   PRE piMState = off ∧ color = red THEN
      SELECT piMState = off ∧ color = red THEN
         lightR := TRUE || piMState := redOn
      END
   END;

   offRed=
   PRE piMState = redOn THEN
      SELECT piMState = redOn THEN
         lightR := FALSE || piMState := off
      END
   END;

   onOrange =
   PRE piMState = off ∧ color = orange THEN
      SELECT piMState = off ∧ color = orange THEN
         lightO := TRUE || piMState := orangeOn
      END
   END;

   offOrange =
   PRE piMState = orangeOn THEN
      SELECT piMState = orangeOn THEN
         lightO := FALSE || piMState := off
      END
   END;

   change(col)=
   PRE col ∈ Colors ∧ piMState = off THEN
      SELECT col ∈ Colors ∧ piMState = off THEN
         color := col|| piMState := off
      END
   END
END
```

FIGURE B.2 – Machine B associée à l'interface PI_Mlight

```
MACHINE PI_SMlight
SEES ctx
ABSTRACT_CONSTANTS Light_StatesPISM
PROPERTIES
    Light_StatesPISM ⊆ Light_States ∧
    Light_StatesPISM = {off, on}
VARIABLES
   light , color , piSMState
INVARIANT
   light ∈ 𝔹 ∧
   color ∈ Colors ∧
   piSMState ∈ Light_StatesPISM
INITIALISATION
   light , piSMState,color:= FALSE,off,red
OPERATIONS
   onColor(col) =
   PRE piSMState = off ∧ col ∈Colors THEN
      SELECT piSMState = off ∧ col ∈Colors THEN
         light := TRUE || piSMState := on||color:=col
      END
   END;
   offColor =
   PRE piSMState = on THEN
      SELECT piSMState = on THEN
         light := FALSE || piSMState := off
      END
   END
END
```

FIGURE B.3 – Machine B associée à l'interface *PI_SMlight*

```
MACHINE RI_Blight
SEES ctx
ABSTRACT_CONSTANTS Light_StatesRIB
PROPERTIES
    Light_StatesRIB ⊆ Light_States ∧
    Light_StatesRIB = {off}
VARIABLES
   piBState
INVARIANT
   piBState ∈ Light_StatesRIB
INITIALISATION
   piBState :=off
OPERATIONS
   blinkRed(nn) =
   PRE piBState = off ∧ nn∈1..10 THEN
      SELECT piBState = off THEN
         piBState := off
      END
   END;
   blinkGreen(nn) =
   PRE piBState = off ∧ nn∈1..10 THEN
      SELECT piBState = off THEN
         piBState := off
      END
   END
END
```

FIGURE B.4 – Machine B associée à l'interface *RI_Blight*

```
MACHINE
   PI_Slight
SEES ctx
VARIABLES
    light , color , piSState
ABSTRACT_CONSTANTS
    Light_StatesPIS
PROPERTIES
    Light_StatesPIS ⊆ Light_States ∧
    Light_StatesPIS = {off, on}
INVARIANT
   light ∈ 𝔹 ∧
   color ∈ Colors ∧
   piSState ∈ Light_States
INITIALISATION
   light , piSState, color := FALSE, off,red
OPERATIONS
   onLight =
   PRE piSState = off THEN
       SELECT piSState = off THEN
           light := TRUE || piSState := on
       END
   END;
   offLight =
   PRE piSState = on THEN
       SELECT piSState = on THEN
           light := FALSE || piSState := off
       END
   END
END
```

FIGURE B.5 – Machine B associée à l'interface *PI_Slight*

```
MACHINE
   PI_Tlight
SEES
   ctx
VARIABLES
   lightG, lightR, lightO, color, piTState
INVARIANT
   lightG ∈ 𝔹 ∧
   lightR ∈ 𝔹 ∧
   lightO ∈ 𝔹 ∧
   color ∈ Colors ∧
   piTState ∈ Light_States∧
   piTState ∈ {off,greenOn,redOn,orangeOn}
INITIALISATION
    lightG, lightR,lightO:= FALSE, FALSE,FALSE||
    piTState, color:= off,orange
OPERATIONS
   onGreen =
   PRE piTState = off ∧ color = green THEN
       SELECT piTState = off ∧ color = green THEN
           lightG := TRUE || piTState := greenOn
       END
   END;

   offGreen =
   PRE piTState = greenOn THEN
       SELECT piTState = greenOn THEN
           lightG := FALSE || piTState := off||color := red
       END
   END;

   onRed =
   PRE piTState = off ∧ color = red THEN
       SELECT piTState = off ∧ color = red THEN
           lightR := TRUE || piTState := redOn
       END
   END;

   offRed =
   PRE piTState = redOn THEN
       SELECT piTState = redOn THEN
           lightR := FALSE || piTState := off||color := orange
       END
   END;

   onOrange =
   PRE piTState = off ∧ color = orange THEN
       SELECT piTState = off ∧ color = orange THEN
           lightO := TRUE || piTState := orangeOn
       END
   END;

   offOrange =
   PRE piTState = orangeOn THEN
       SELECT piTState = orangeOn THEN
           lightO := FALSE || piTState := off||color := green
       END
   END
END
```

FIGURE B.6 – Machine B associée à l'interface *PI_Tlight*

```
REFINEMENT Adapt1
REFINES RI_GRlight
SEES ctx
INCLUDES PI_Mlight
PROMOTES offGreen, offRed
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL ∈ Light_StatesRIGR ↔ Light_StatesPIM ∧
    REL = {(off↦off),(greenOn↦greenOn),(redOn↦redOn)}
INVARIANT
    (riGRState↦piMState) ∈ REL
OPERATIONS
   onGreen =
   BEGIN
     ANY col WHERE col = green THEN
        change(col)
     END;
     onGreen1
   END;
   onRed =
   BEGIN
     ANY col WHERE col = red THEN
       change(col)
     END;
     onRed1
   END
END
```

FIGURE B.7 – Adaptateur entre les interfaces *RI_GRlight* et *PI_Mlight*

```
REFINEMENT Adapt2
REFINES RI_GRlight
INCLUDES PI_SMlight
SEES ctx
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL ∈ Light_StatesRIGR ↔ Light_StatesPISM ∧
    REL = {(off↦off),(greenOn↦on),(redOn↦on)}
INVARIANT
    (riGRState↦ piSMState) ∈ REL
OPERATIONS
    onGreen =
    ANY col WHERE col = green THEN
        onColor(col)
    END;
    offGreen =
    offColor ;
    onRed =
    ANY col WHERE col = green THEN
        onColor(col)
    END;
   offRed =
   offColor
END
```

FIGURE B.8 – Adaptateur entre les interfaces *RI_GRlight* et *PI_SMlight*

```
REFINEMENT Adapt3
REFINES
   RI_SMlight
SEES
   ctx
INCLUDES
    PI_GRlight
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL ∈ Light_StatesRISM ↔ Light_StatesPIGR   ∧
    REL = {(off↦off),(greenOn↦on),(redOn↦on)}
INVARIANT
    (piSMState ↦ piGRState) ∈ REL
OPERATIONS
   onColor (col) =
   BEGIN
     SELECT col = green THEN
         onGreen
     WHEN col = red THEN
         onRed
     END
   END;

   offColor =
   BEGIN
     SELECT piGRState = greenOn THEN
         offGreen
     WHEN piGRState = redOn THEN
         offRed
     END
   END
END
```

FIGURE B.9 – Adaptateur entre les interfaces *PI_SMlight* et *PI_GRlight*

```
REFINEMENT Adapt4
REFINES PI_GRlight
INCLUDES Red.PI_Slight, Green.PI_Slight
SEES ctx
ABSTRACT_CONSTANTS
    REL
PROPERTIES
    REL = {(off↦(off↦off)),(greenOn↦(on ↦ off)),
           (redOn↦(off ↦ on)), (greenRedOn↦(on ↦ on))}
INVARIANT
    (piGRState↦(Green.piSState↦Red.piSState)) ∈ REL∧
     Green.light=lightG∧
     Red.light=lightR
OPERATIONS
     onGreen  = Green.onLight;
     offGreen = Green.offLight;
     onRed    = Red.onLight;
     offRed   = Red.offLight
END
```

FIGURE B.10 – Adaptateur entre *RI_GRlight*, *Red.PI_Slight* et *Green.PI_Slight*

```
IMPLEMENTATION
   Adaptboucle
REFINES
   RI_Blight
SEES
   ctx
IMPORTS
   PI_GRlight
INVARIANT
   (piGRState = off ) ⇔ ( piBState = off )
OPERATIONS
   blinkRed ( nn ) =
    BEGIN
     VAR tt IN
       tt := nn ;
       WHILE tt > 0 DO
         onRed ;            /* corps de la boucle*/
         offRed;
         tt := tt − 1
       INVARIANT
         tt ∈ NAT ∧ tt ≥ 0∧ piGRState = off /*conditions invariantes*/
       VARIANT
         tt                 /*valeur entière qui décroît */
       END
     END
   END ;

   blinkGreen ( nn ) =
   BEGIN
     VAR tt IN
       tt := nn ;
       WHILE tt > 0 DO
         onGreen ;          /* corps de la boucle*/
         offGreen ;
         tt := tt − 1
       INVARIANT
         tt ∈ NAT ∧ tt ≥ 0∧ piGRState = off /*conditions invariantes*/
       VARIANT
         tt                 /*valeur entière qui décroît */
       END
     END
   END
END
```

FIGURE B.11 – Adaptateur entre les interfaces RI_Blight et $PI\_GRlight$

B.2 Les spécifications B de lecteur de CD

```
MACHINE
   RI_player
SETS
    R_States = { r1 , r2 , r3 , r4 , r5 , r6 }
ABSTRACT_CONSTANTS
    RF_States
PROPERTIES
    RF_States ⊆ R_States ∧
    RF_States = { r5 }
ABSTRACT_VARIABLES stateR , moneyr
INVARIANT stateR ∈ R_States ∧ moneyr ∈ NAT
INITIALISATION  stateR := r1 || moneyr := 0
OPERATIONS
    pay ( mm ) =
    PRE  stateR = r1 ∧ mm ∈ { 20 , 50 } THEN
        SELECT  stateR = r1 THEN  stateR := r2 || moneyr := mm
        END
    END ;

    play =
    PRE  stateR ∈ { r2 , r4 , r6 } ∧ moneyr ∈ { 20 , 50 } THEN
        SELECT  stateR = r2 THEN  stateR := r3 || moneyr := 0
        WHEN stateR = r4 THEN  stateR := r3
       WHEN stateR = r6 THEN  stateR := r3
        END
    END ;

    volume =
    PRE  stateR = r3 THEN
        SELECT  stateR = r3 THEN  stateR := r3
        END
    END ;

    speed =
    PRE  stateR = r3 THEN
        SELECT  stateR = r3 THEN  stateR := r6
        END
    END ;

    stop =
    PRE  stateR = r3 THEN
        SELECT  stateR = r3 THEN  stateR := r5
        END
    END ;

    pause =
    PRE  stateR = r3 THEN
        SELECT  stateR = r3 THEN  stateR := r4
        END
    END
```

FIGURE B.12 – Machine B associée à l'interface *RI_player*

```
MACHINE
   PI_player
SETS
    P_States = {p0,p1,p2,p3,p4,p5,p6,p7,p8}
ABSTRACT_CONSTANTS
    PF_States
PROPERTIES
    PF_States ⊆ P_States ∧
    PF_States = { p8 }
ABSTRACT_VARIABLES stateP , money
INVARIANT stateP ∈ P_States ∧ money ∈ NAT
INITIALISATION  stateP := p0 || money := 0
OPERATIONS
    on =
    PRE  stateP = p0 THEN
       SELECT  stateP = p0 THEN  stateP := p1
       END
    END ;

    volume =
    PRE  stateP ∈ {p0 ,p1,p2,p3,p4,p5,p6,p7}
       THEN
       SELECT  stateP = p0 THEN  stateP := p0
       WHEN stateP = p1 THEN  stateP := p1
       WHEN stateP = p2 THEN  stateP := p2
       WHEN stateP = p3 THEN  stateP := p3
       WHEN stateP = p4 THEN  stateP := p4
       WHEN stateP = p5 THEN  stateP := p5
       WHEN stateP = p6 THEN  stateP := p6
       WHEN stateP = p7 THEN  stateP := p7
       END
    END ;

    insertcoin  ( cc ) =
    PRE  stateP = p1 ∧ cc ∈ { 10 , 20 , 50 } THEN
       SELECT stateP = p1 THEN
              stateP := p2 || money := cc
       END
    END ;

    SelectCD1 =
    PRE  stateP ∈ { p2 , p7 } ∧ money ≥ 10 THEN
       SELECT stateP = p2 THEN
              stateP := p3 || money := money − 10
       WHEN stateP = p7 THEN
              stateP := p3 || money := money − 10
       END
    END ;

    SelectCD2 =
    PRE  stateP ∈ { p2 , p7 } ∧ money ≥ 20 THEN
       SELECT stateP = p2 THEN
              stateP := p3 || money := money − 20
       WHEN stateP = p7 THEN
              stateP := p3  || money := money − 20
       END
    END ;

    SelectCD3 =
    PRE  stateP ∈ { p2 , p7 } ∧ money ≥ 50 THEN
       SELECT stateP = p2 THEN
              stateP := p3 || money := money − 50
       WHEN stateP = p7 THEN
              stateP := p3  || money := money − 50

       END
    END ;

    play1 =
    PRE  stateP ∈ { p3 , p5 } THEN
       SELECT  stateP = p3 THEN  stateP := p4
       WHEN stateP = p5 THEN  stateP := p4
       END
    END ;

    stop1 =
    PRE  stateP = p4 THEN
       SELECT  stateP = p4 THEN  stateP := p7
       END
    END ;

    pause1 =
    PRE  stateP ∈ { p4 , p6 } THEN
       SELECT  stateP = p4 THEN  stateP := p5
       WHEN stateP = p6 THEN  stateP := p5

       END
    END ;

    speed1 =
    PRE  stateP = p5 THEN
       SELECT  stateP = p5 THEN  stateP := p6
       END
    END ;

    off  =
    PRE  stateP = p7 THEN
       SELECT  stateP = p7 THEN  stateP := p8
       END
    END
END
```

FIGURE B.13 – Machine B associée à l'interface *PI_player*

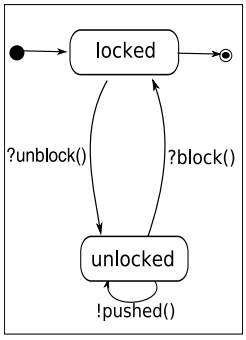
C

Études de cas d'assemblage de plusieurs composants

C.1 Un système de contrôle d'accès à un bâtiment

L'étude de cas du système de contrôle d'accès à un bâtiment est déjà présentée dans le chapitre 6. Nous présentons par la suite les spécifications des différents composants dans cette étude de cas.

Le composant *Turnstile*



(a) STE

```
MACHINE Turnstile
SEES ctx
ABSTRACT_VARIABLES stateT
INVARIANT
   stateT ∈ Turnstile_States
INITIALISATION
   stateT := locked
OPERATIONS
   unblock =
   PRE stateT = locked THEN
      SELECT stateT = locked THEN
            stateT := unlocked
      END
   END ;
   block =
   PRE stateT = unlocked THEN
      SELECT stateT = unlocked THEN
            stateT := locked
      END
   END ;
   pushed =
   PRE stateT = unlocked THEN
      SELECT stateT = unlocked THEN
            stateT := unlocked
      END
   END
END
```

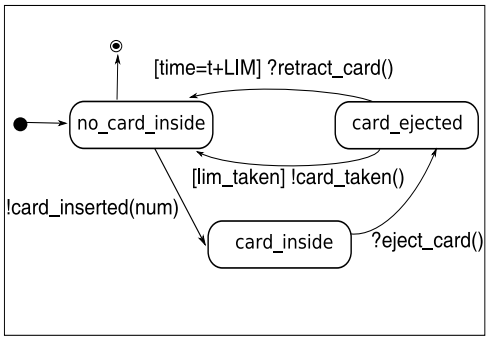
(b) Spécification B

FIGURE C.1 – Le composant *Turnstile*

Le comportement du composant *Turnstile* est présenté par le système de transitions étiqueté de la Figure C.1(a) ; celui-ci comporte deux états *locked* et *unlocked*, *locked* correspondant à l'état initial et final. Les deux transitions étiquetées par ?*block* et ?*unblock* permettent le changement d'état du composant. La transition étiquetée par !*pushed* correspond au franchissement du tourniquet par une personne qui provoque son entrée dans le bâtiment (ou sa sortie).

La Figure C.1(b) présente le modèle B associé au composant *Turnstile*.

Le composant *Card_reader*



(a) STE

```
MACHINE
   Card_reader
SEES
   ctx, Time
ABSTRACT_VARIABLES
   stateCard , t_eject
INVARIANT
    stateCard ∈   Card_States ∧
    t_eject ∈ NAT
INITIALISATION
    stateCard := no_card_inside || t_eject := 0
OPERATIONS
    card_inserted =
    PRE stateCard =  no_card_inside THEN
        SELECT stateCard =  no_card_inside THEN
               stateCard := card_inside
        END
    END;

    eject_card =
    PRE stateCard = card_inside THEN
        SELECT stateCard = card_inside THEN
               stateCard := card_ejected ||t_eject := time
        END
    END;

    retract_card =
    PRE stateCard = card_ejected ∧ time ≥ t_eject + LIM  THEN
        SELECT stateCard = card_ejected THEN
               stateCard :=  no_card_inside
        END
    END;

     card_taken =
     PRE  stateCard = card_ejected ∧ time < t_eject + LIM  THEN
          SELECT stateCard = card_ejected THEN
                 stateCard :=  no_card_inside
          END
     END
END
```

(b) Spécifications B

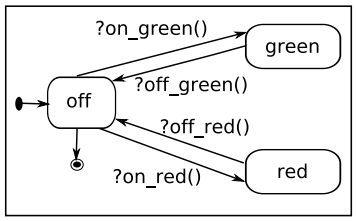
FIGURE C.2 – Le composant Card_reader

Le comportement du composant *Card_reader* est décrit par le système de transitions étiqueté de la Figure C.2(a) ; il comporte trois états, l'état initial et l'état final étant le même, *no_card_inside*, et quatre transitions. La garde de la transition étiquetée $[lim\_taken]!card\_taken()$ entre les états *card_ejected* et *no_card_inside* correspond à une abstraction de prédicat défini par $time \in [t...t + LIM]$ où LIM représente une constante.

La Figure C.2(b) présente le modèle B obtenu par transformation du STE associé au composant *Card_reader* présenté Figure C.2(a), dans laquelle :

- stateCard désigne la variable de contrôle et ses états possibles, no_card_inside, card_inside, card_ejected. La variable locale t_eject et la variable de contexte time sont nécessaires pour exprimer les délais,
- les opérations B correspondent aux événements,
- la visibilité de la variable time, définie dans le modèle Clock, est assurée par la clause **SEES** Clock.

Le composant Lamp



(a) STE

```
MACHINE
   Lights
SEES
   ctx
ABSTRACT_VARIABLES
    stateL
INVARIANT
    stateL ∈ Lights_States
INITIALISATION
    stateL := off
OPERATIONS
    on_green =
    PRE stateL = off THEN
        SELECT stateL = off THEN
                stateL := green
        END
    END;

    off_green =
    PRE stateL = green THEN
        SELECT stateL = green THEN
                stateL := off
        END
    END;

    on_red =
    PRE stateL = off THEN
        SELECT stateL = off THEN
                stateL := red
        END
    END;

    off_red =
    PRE stateL = red THEN
        SELECT stateL = red THEN
                stateL := off
        END
    END
END
```

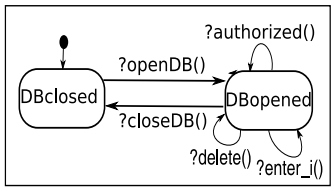
(b) Spécification B

FIGURE C.3 – Le composant Lamp

Le comportement du composant *Lamp* est présenté Figure C.3(a) : il comporte trois états possibles, un état initial et un état final qui sont le même, off. Les changements d'état sont effectués à l'aide de l'une des quatre transitions étiquetées par $?on\_green$, $?off\_green$, $?on\_red$ et $?off\_red$.

La Figure C.3(b) présente le modèle B associé au composant *Lamp*.

Le composant *DB*



(a) STE

```
MACHINE
   DB
SEES
   ctx
ABSTRACT_VARIABLES
    stateDB ,
    n_person
INVARIANT
    stateDB ∈ DB_States ∧ n_person ∈ NAT
INITIALISATION
     stateDB := DBclosed || n_person := 0
OPERATIONS
     openDB=
     PRE stateDB = DBclosed THEN
         SELECT stateDB = DBclosed THEN
                stateDB := DBopened
         END
     END;
     closeDB=
     PRE stateDB = DBopened THEN
         SELECT stateDB = DBopened THEN
                stateDB := DBclosed
         END
     END;
     res ⟵ authorised =
     PRE stateDB = DBopened THEN
         SELECT stateDB = DBopened THEN
                res :∈ 𝔹||
                stateDB:= DBopened
         END
     END;
     enter_i =
     PRE stateDB = DBopened THEN
         SELECT stateDB = DBopened THEN
                IF 0 < n_person ∧ n_person < MAXINT THEN
                   n_person := n_person + 1
                END||
                stateDB:= DBopened
     END
     END;
     delete =
     PRE stateDB = DBopened THEN
         SELECT stateDB = DBopened THEN
                IF n_person > 0 THEN
                   n_person := n_person − 1
                END||
                stateDB:= DBopened
         END
     END
END
```

(b) Spécification B

FIGURE C.4 – Le composant *DB*

Le comportement du composant *DB* est présenté par le système de transitions étiqueté de la Figure C.4(a) ; celui-ci comporte deux états *opened* et *closed*, *closed* correspondant à l'état initial. Les deux transitions étiquetées par ?*open* et ?*close* permettent le changement d'état du

composant. Les transitions étiquetées par ?*delete*, ?*add* et ?*authorized* désignent des évènements de consultation ou de modification de la base de données.

La Figure C.4(b) présente le modèle B associé au composant DB.

Spécifications associées au système assemblé

```
REFINEMENT
   AccesControl0
REFINES
   Abstract_AccesControl
SEES
   ctx
INCLUDES
   Card_reader, DB, Lights, Entry. Turnstile ,
   Left1. Turnstile , temp
VARIABLES
   res , t_unblock, pp
CONSTANTS
    REL
PROPERTIES
    REL = {( out_of_service↦ (locked ↦ (DBclosed  ↦
    ( off  ↦ (no_card_inside ↦ locked))))) ,
    (in_service_empty ↦  (locked ↦ (DBclosed ↦
    ( off  ↦ (no_card_inside ↦ locked))))) ,
    (in_service_not_empty ↦ (unlocked ↦ (DBclosed ↦
    ( off  ↦ (no_card_inside ↦ locked)))))}
INVARIANT
    pp ∈ ℙ ∧
    ( state  ↦ ((Left1.stateT) ↦ (stateDB ↦
    (stateL↦(stateCard ↦ (Entry.stateT))))))∈REL ∧
    res  ∈ ℙ ∧
    t_unblock ∈ NAT
INITIALISATION
    res  := FALSE ; t_unblock := 0 ; pp := TRUE
OPERATIONS
    UserEnter =
    BEGIN
    card_inserted  ;
    res ⟵ authorised  ;
    IF res = TRUE THEN
       on_green ;
       ANY xx WHERE xx ∈ NAT ∧
                    time < MAXINT − xx THEN
           MAJ ( xx )
       END ;
       IF time ≥ t_eject + LIM THEN
          retract_card
       ELSE
            card_taken ;
            Entry.unblock ;
            t_unblock := time ;
            ANY xx WHERE xx ∈ NAT ∧
                         time < MAXINT − xx THEN
                MAJ ( xx )
            END ;
            IF time < t_unblock + LIM ∧
               n_person < MAXINT  THEN
               Entry.pushed ;
               enter_i ;
               pp := FALSE ;
               IF (Left1.stateT) = locked THEN
                  Left1.unblock
               END
            ELSE
                Entry.block
            END

       END;
       off_green
    ELSE
        on_red ;
        eject_card  ;
        ANY xx WHERE xx ∈ NAT ∧
                     time < MAXINT − xx THEN
           MAJ ( xx )
        END;
        card_taken ;
        off_red
    END
    END ;

    init  =
    BEGIN
     skip
    END ;

    UserLeft  =
    BEGIN
    Left1.pushed ;
    delete  ;
    pp := bool (n_person > 0);
    IF pp = TRUE    THEN
       Left1.block
    END
    END ;

    close  =
    BEGIN
     skip
    END
END
```

FIGURE C.5 – Raffinement AccessControl erroné

```
REFINEMENT
   AccesControl
REFINES
   Abstract_AccesControl
SEES
   ctx
INCLUDES
   Card_reader, DB, Lights, Entry. Turnstile ,
   Left1. Turnstile , Time
VARIABLES
   res, t_unblock, pp
CONSTANTS
    REL
PROPERTIES
    REL = {(out_of_service↦ (locked ↦ (DBclosed  ↦
    (off ↦ (no_card_inside ↦ locked))))) ,
    (in_service_empty ↦  (locked ↦ (DBopened ↦
    (off ↦ (no_card_inside ↦ locked)))) ,
    (in_service_not_empty ↦ (unlocked ↦ (DBopened ↦
    (off ↦ (no_card_inside ↦ locked))))))}
INVARIANT
    pp ∈ ℙ ∧
    (state ↦ ((Left1.stateT) ↦ (stateDB ↦
    (stateL↦(stateCard ↦ (Entry.stateT))))))∈REL ∧
    res ∈ ℙ ∧
    t_unblock ∈ NAT
INITIALISATION
    res := FALSE ; t_unblock := 0 ; pp := TRUE
OPERATIONS
    UserEnter =
    BEGIN
    card_inserted  ;
    res ⟵ authorised  ;
    IF res = TRUE THEN
       on_green ;
       eject_card  ;
       ANY xx WHERE xx ∈ NAT ∧
                    time < MAXINT − xx THEN
           MAJ ( xx )
       END ;
       IF time ≥ t_eject + LIM THEN
          retract_card
       ELSE
            card_taken ;
            Entry.unblock  ;
            t_unblock := time ;
            ANY xx WHERE xx ∈ NAT ∧
                         time < MAXINT − xx THEN
                MAJ ( xx )
            END ;
            IF time < t_unblock + LIM ∧
               n_person < MAXINT  THEN
                Entry.pushed ;
                enter_i  ;
                pp := FALSE ;
                IF (Left1.stateT) = locked THEN
                   Left1.unblock
                END
            ELSE
                Entry.block
            END

       END;
       off_green
    ELSE
        on_red ;
        eject_card  ;
        ANY xx WHERE xx ∈ NAT ∧
                     time < MAXINT − xx THEN
           MAJ ( xx )
        END;
        card_taken ;
        off_red
    END
    END ;

    init  =
    BEGIN
     skip
    END ;

    UserLeft  =
    BEGIN
    Left1.pushed ;
    delete  ;
    pp := bool (n_person > 0);
    IF pp = TRUE    THEN
       Left1.block
    END
    END ;

    close  =
    BEGIN
     skip
    END
END
```

FIGURE C.6 – Raffinement AccessControl corrigé

C.2 Les feux tricolores d'un carrefour

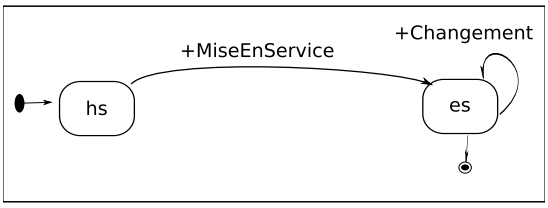
Présentation de l'étude de cas

La circulation d'un carrefour est réglée par deux feux tricolores dont les couleurs sont vert, orange ou rouge. Sur chacun des feux une seule des couleurs est active à la fois. Le système de feux du carrefour peut être en service ou hors service. Lorsque le système est hors service, les

deux feux sont oranges. Lorsque le système est en service, la couleur de chacun des feux change suivant le cycle : orange puis rouge puis vert puis orange, *etc.*

Un véhicule ne peut s'engager sur une voie que si le feu n'est pas rouge. Lorsque le système est en service, les feux doivent être réglés de façon à ce que deux véhicules venant de voies différentes ne se trouvent pas en même temps sur le carrefour. On désire obtenir un système assurant la mise en route du carrefour et la gestion du changement de couleur des feux

C.2.1 Spécification abstraite du système



(a) STE

```
MACHINE carrefour
SEES ctx
VARIABLES etat
INVARIANT
   etat∈Etats
INITIALISATION
   etat:=hs
OPERATIONS
 MiseEnService=
  PRE etat=hs THEN
   etat:=es
  END;
  Changement=
  PRE etat=es THEN
   skip
  END
END
```

(b) Spécifications B

FIGURE C.7 – Le système abstrait du Carrefour

Le système de feux du carrefour peut être en service ou hors service. La Figure C.7(a) présente son protocole d'utilisation dans lequel deux états ont été introduits :

- hs : le système est hors service,
- es : le système est en service,

Le système offre deux services modélisés par les transitions MiseEnService et Change.

La Figure C.7(b) présente le modèle B associé au système abstrait.

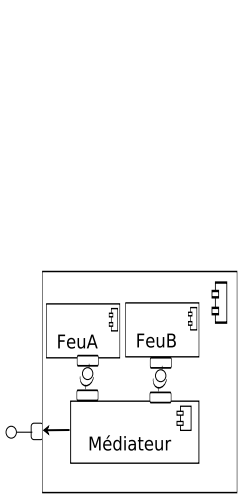
C.2.2 Description de l'architecture du système en termes de composants

Architecture du système. Une spécification concrète du système est explicitée sous la forme d'un diagramme de structure composite d'UML 2.0 Figure C.8(a). Le système comporte deux feux tricolores.

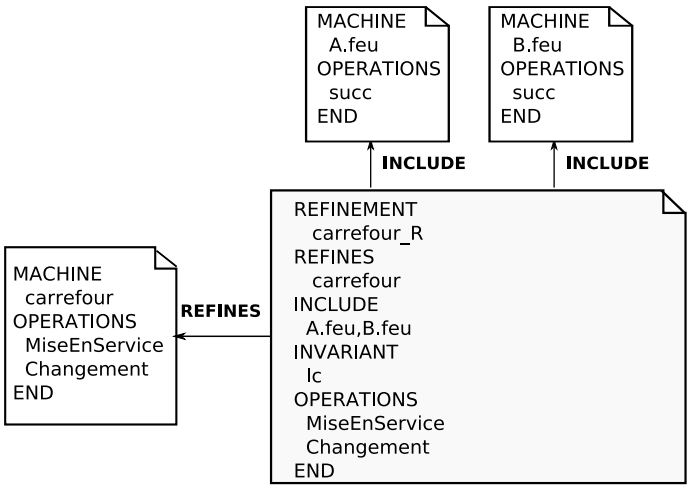
L'architecture du système B proposée en Figure C.8(b) utilise un composants existants, à savoir feu. Notons que ce composant est utilisé deux fois.

Le composant Feu_tricolore. Le comportement du composant Feu_tricolore est présenté Figure C.9(a) : il comporte trois états possibles orange, rouge et vert. Les changements d'état sont effectués à l'aide de l'évènement succ. La couleur du feu change suivant le cycle : orange puis rouge puis vert puis orange, *etc.*

La Figure C.9(b) présente le modèle B associé au composant Feu_tricolore.

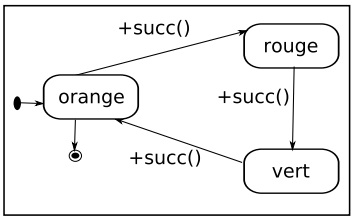


(a) Diagramme de structure composite



(b) Spécifications B

FIGURE C.8 – Architecture du système Carrefour



(a) STE

```
MACHINE feu
SEES ctx
VARIABLES ff
INVARIANT
  ff ∈ Couleurs
INITIALISATION
  ff := orange
OPERATIONS
  Succ=
   PRE ff∈Couleurs THEN
    SELECT ff= orange THEN
     ff := rouge
    WHEN  ff=rouge THEN
     ff := vert
    WHEN  ff=vert THEN
     ff := orange
    END
   END
END
```

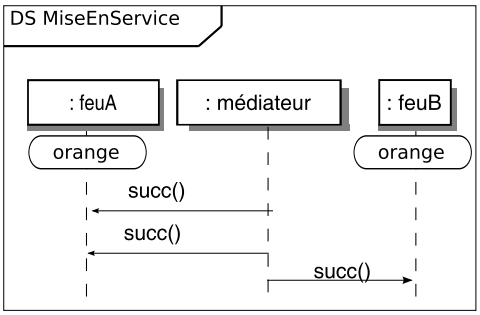
(b) Spécifications B

FIGURE C.9 – Le composant Feu_tricolore

C.2.3 Spécification des services en termes des composants de l'architecture

Description du service MiseEnService. La Figure C.10(a) décrit le comportement du système lorsqu'on le met en service. Nous avons choisi arbitrairement de mettre un feu au vert et l'autre au rouge pour mettre en service le système.

La Figure C.10(b) présente l'opération B associé au service MiseEnService dans le raffinement B carrefour_R.



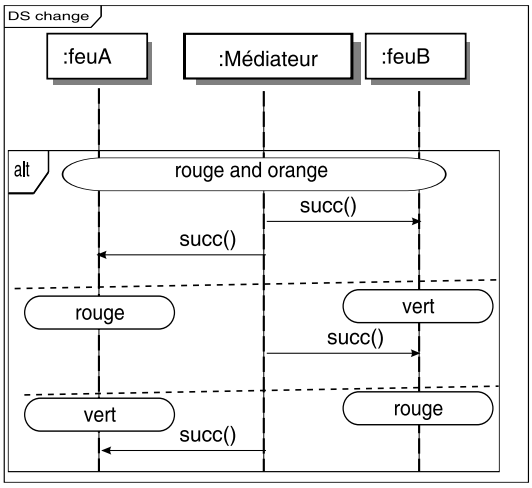
(a) Diagramme de séquences

```
MiseEnService=
 PRE A.ff=orange ∧ B.ff=orange THEN
  A.Succ;
  A.Succ;
  B.Succ
END;
```

(b) Spécification B

FIGURE C.10 – Description du service MiseEnService

Description du service Change. La Figure C.11(a) décrit le comportement du système lors-



(a) Diagramme de séquences

```
Changement=
  PRE ((A.ff=rouge)or(B.ff=rouge))
      ∧¬(A.ff=B.ff) THEN
   SELECT (A.ff=rouge ∧  B.ff=orange)or
          (B.ff=rouge ∧  A.ff=orange)THEN
    A.Succ;B.Succ
   WHEN  (A.ff=rouge ∧  B.ff=vert)THEN
    B.Succ
   WHEN  (B.ff=rouge ∧  A.ff=vert)THEN
    A.Succ
   END
  END
```

(b) Spécification B

FIGURE C.11 – Description du service Change

qu'on change l'état du système. Il y a trois cas de figures de changement qui dépendent de l'état initial des deux feux lors de l'exécution de l'évènement change.

La Figure C.11(b) présente l'opération B associé au service Change dans le raffinement B carrefour_R.

Définition de la relation entre le système abstrait et le système concret. La relation entre les état des deux systèmes est définit comme suit :

– hs : lorsque le système est hors service, les deux feux sont orange.

– es : lorsque le système est en service, un et seulement un des deux feux doit être rouge.

L'invariant de collage défini en B (dans le raffinementcarrefour_R) est présenté dans la Figure C.12.

INVARIANT
((etat = hs) ⇔ ((A.ff = orange) ∧ (B.ff=orange)))∧
((etat = es) ⇔ (((A.ff = rouge) ∨ (B.ff=rouge))∧ ¬(A.ff=B.ff)))

FIGURE C.12 – Invariant

C.2.4 Spécification abstraite du composant *Feu_tricolore*

Le composant Feu_tricolore utilisé dans le système de feux du carrefour comme un composant boîte noire offrant le service succ. La description abstraite du comportement du composant Feu_tricolore est présentée Figure C.9. Dans ce qui suit nous détaillons la structure interne de ce composant et son comportement interne. L'architecture du système carrefour finale est explicitée sous la forme d'un diagramme de structure composite d'UML 2.0 Figure C.13.

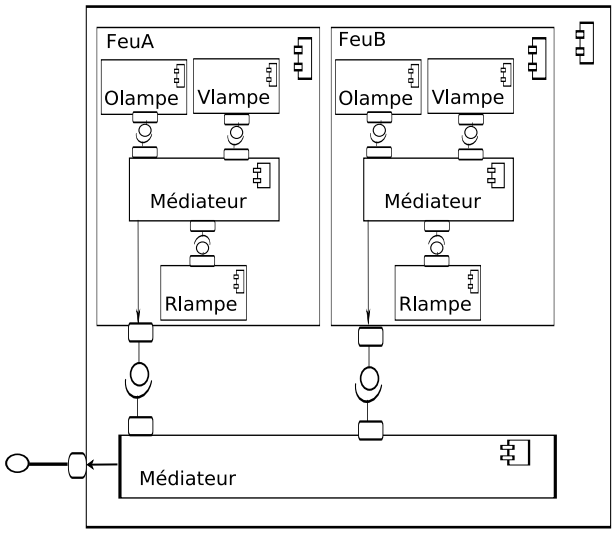
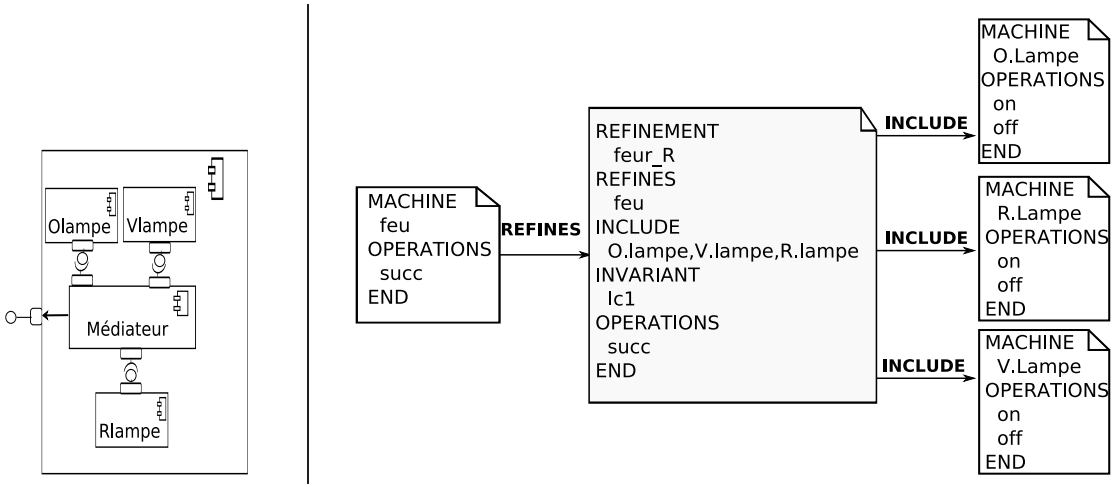


FIGURE C.13 – Architecture du système

C.2.5 Description de l'architecture du composant Feu_tricolore

l'architecture du composant Feu_tricolore est explicitée sous la forme d'un diagramme de structure composite d'UML 2.0 Figure C.14(a). Le système comporte trois lampe chacune d'elle est associée à une couleur.

L'architecture du système B proposée Figure C.14(b) utilise un composants existants, à savoir lampe. Notons que ce composant est utilisé trois fois.



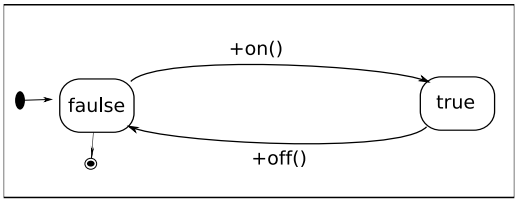
(a) Diagramme de structure composite

(b) Spécifications B

FIGURE C.14 – Architecture du composant Feu_tricolore

Le composant Lampe. Le comportement du composant Lampe est présenté Figure C.15(a) : il comporte deux états true et faulse. Deux transitions on et off permettent son changement d'état.

La Figure C.15(b) présente le modèle B associé au composant Lampe.



(a) STE

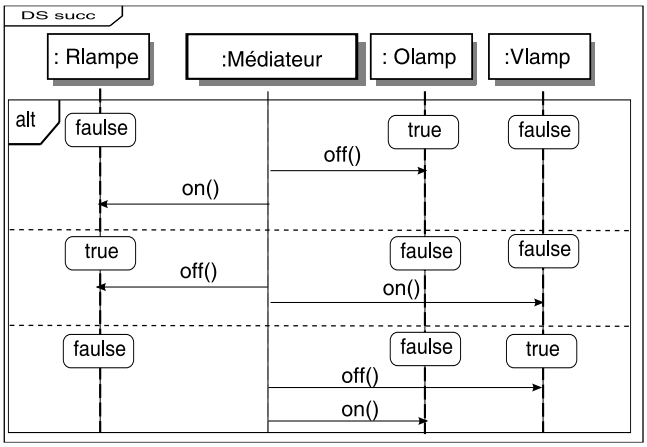
```
MACHINE lampe
VARIABLES stateL
INVARIANT
  stateL ∈ 𝔹
INITIALISATION
  stateL := FALSE
OPERATIONS
  on =
   PRE stateL = FALSE THEN
    stateL := TRUE
   END;
  off =
   PRE stateL = TRUE THEN
    stateL := FALSE
   END
END
```

(b) Spécifications B

FIGURE C.15 – Le composant Lampe

C.2.6 Spécification des services du composant Feu_tricolore

Description du service Succ. La Figure C.16(a) décrit le comportement du système lorsqu'on veut changé la couleur d'un feu. Le changement dépend de l'état initial des trois lampes. On a trois cas possibles d'exécution du service succ qui dépendent de l'état initial des trois lampes dans ce service. La Figure C.16(b) présente l'opération B associé au service succ dans le raffinement B feu_R.



(a) Diagramme de séquences

```
Succ=
PRE   ((R.stateL=FALSE) ∧ (O.stateL=TRUE) ∧ (V.stateL=FALSE))or
      ((R.stateL=TRUE) ∧ (O.stateL=FALSE) ∧ (V.stateL=FALSE))or
      ((R.stateL=FALSE) ∧ (O.stateL=FALSE) ∧ (V.stateL=TRUE)) THEN
SELECT   ((R.stateL=FALSE) ∧ (O.stateL=TRUE) ∧ (V.stateL=FALSE)) THEN  O.off;R.on
WHEN     ((R.stateL=TRUE) ∧ (O.stateL=FALSE) ∧ (V.stateL=FALSE)) THEN  R.off;V.on
WHEN     ((R.stateL=FALSE) ∧ (O.stateL=FALSE) ∧ (V.stateL=TRUE)) THEN  V.off;O.on
END
END
```

(b) Spécification B

FIGURE C.16 – Description du service Succ

Relation entre les états abstraits et concrets du composant Feu_tricolore. La relation entre les état des deux systèmes est définit comme suit :

- orange : lorsque le feu est orange, la lampe orange est allumée et les deux autres sont éteintes.
- rouge : lorsque le feu est rouge, la lampe rouge est allumée et les deux autres sont éteintes.
- vert : lorsque le feu est vert, la lampe verte est allumée et les deux autres sont éteintes.

L'invariant de collage défini en B (dans le raffinement feu_R) est présenté ci-dessous (Figure C.17).

```
(( ff=orange)⇔ ((R.stateL=FALSE) ∧ (O.stateL=TRUE) ∧ (V.stateL=FALSE)))∧
(( ff=rouge) ⇔ (R.stateL=TRUE) ∧ (O.stateL=FALSE) ∧ (V.stateL=FALSE)))∧
(( ff=vert)  ⇔  ((R.stateL=FALSE) ∧ (O.stateL=FALSE) ∧ (V.stateL=TRUE)))
```

FIGURE C.17 – Invariant

Résumé

Le sujet général de cette thèse est l'étude de la vérification et de la correction de spécifications B dans le contexte d'application d'une approche CBSE (« Component-Based Software Engineering »). La méthode B est reconnue comme une méthode formelle bien outillée pour développer formellement les logiciels, elle dispose du raffinement et de prouveurs permettant un développement rigoureux. L'approche CBSE consiste à développer des logiciels par assemblage de composants, elle trouve son intérêt pour des systèmes de grandes tailles.

Cette thèse comprend trois contributions principales. La première considère la mise en évidence de schémas de constructions B basés sur le raffinement et l'inclusion de machines B ainsi que l'étude de ces schémas pour modéliser des relations entre des Systèmes de Transitions Étiquetés (STEs). La deuxième contribution consiste à l'utilisation de deux formalismes : (i) le formalisme UML pour spécifier l'assemblage de deux interfaces (fournie et requise) et de plusieurs composants ainsi que les communications entre les composants, (ii) le formalisme B pour vérifier les assemblages. La troisième contribution étudie l'aide à la correction des spécifications B à partir de l'échec de la preuve en B. Cette étude est d'abord générale et indépendante du contexte, puis elle tient compte du contexte CBSE et s'intéresse à la détection et la correction des incompatibilités : pour l'assemblage de deux interfaces on corrige les adaptateurs en considérant les trois niveaux syntaxique, sémantique et protocole, pour l'assemblage et la coordination de plusieurs composants on corrige les médiateurs en considérant les niveaux syntaxique et protocole.

Mots-clés: méthode B, CBSE, raffinement, retour de l'échec de preuve, correction

Abstract

The subject of our thesis aims at studying the verification and the correction of B specifications in the context of applying a CBSE approach. The B method is recognized as a well equipped formal method, used for developing software formally. It is characterized by a refinement capability and a power of provers, and thus allows a rigorous development. The CBSE approach consists in developing software by components assembly ; it finds its interest for large complicated systems.

Our research provides three main contributions. The first contribution deals with some B constructions based on the refinement and the inclusion of B machines. Indeed, we study these B constructions for the specification of relations between labelled transition systems. The second contribution consists of the use of two formalisms : (i) the UML formalism to specify the assembly of two interfaces (required and provided) and of several components as well as the communications between the components ; (ii) the B formalism to verify the assemblies. The third contribution studies the correction of the B specifications from the feedback of unsuccessful proofs. This study is also generic and independent of the context, then it takes into consideration the CBSE context and it is interested in the detection and the correction of the incompatibilities : for the assembly of two interfaces we correct the adapters by considering the syntactic, semantics and protocol levels, while for the assembly and the coordination of several components we correct the mediators by considering the syntactic and protocol levels.

Keywords: B method, CBSE, refinement, feedback of the failed proofs, correction