INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de

DOCTEUR DE l'IP Grenoble

Spécialité : Systèmes et Logiciels

Préparée au Laboratoire d'Informatique de Grenoble – UMR 5217
avec un bourse BDI cofinancée par le CNRS et STMicroelectronics

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

présentée et soutenue publiquement par

Nicolas STOULS

Le 14 décembre 2007

TITRE :

Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés

Directeurs de thèse :
Marie-Laure Potet, Professeur à l'IP Grenoble
Sylvain Boulmé, Maître de conférence à l'IP Grenoble

JURY

M. Jean-Claude Fernandez , *Président*
Mme. Marie-Laure Potet , *Directeur de thèse*
M. Sylvain Boulmé , *Co-encadrant*
M. Jacques Julliand , *Rapporteur*
M. Michael Leuschel , *Rapporteur*
M. Claude Marché , *Examinateur*
M. Christophe Planat , *Examinateur*

Remerciements

Au terme de cette thèse, une prise de conscience s'impose à moi : bien que mon nom soit inscrit en place d'honneur sur la couverture de ce manuscrit, ce résultat est loin d'être le mien en nom propre. Il est le fruit de nombreuses collaborations et discussions. Cette aventure a commencé en licence, lorsque j'ai été accepté en magistère d'informatique. C'est cette découverte des laboratoires de recherche qui m'a donné goût à la recherche et à l'enseignement. C'est grâce à vous, Laurent Trilling, Didier Bert, Marie-Laure Potet et Daniel Dollé, que cette thèse a pu avoir lieu. Je vous dis donc : merci.

Ensuite, Marie-Laure, encore et toujours, se lance avec Sylvain Boulmé pour m'encadrer durant ces années de thèse. Ils ont tous les deux accepté de leur plein gré et avec le sourire et m'ont supporté quand nous avons discuté, ri, ou échangé calmement des avis divergents à coups de noms d'oiseaux et de tout accessoire de bureau homologué disponible dans les locaux. Personnellement, j'ai tiré un grand enseignement de cette collaboration, mais honnêtement, si l'on vous avait décrit précisément ce en quoi vous vous engagiez avant que vous ne vous lanciez, auriez-vous signé ? Pour tout ce que vous avez fait : merci.

Mais je pense que les personnes qui sont le plus à remercier sont Jacques Julliand, Michael Leuschel, Jean-Claude Fernandez, Claude Marché, Pascal Urard et Christophe Planat. En effet, bien que ne me connaissant pas ou peu, ils m'ont fait l'honneur d'accepter de lire mon manuscrit ***en entier*** (ce qui est un exploit en soit) et de me faire part de leurs remarques. En particulier, l'intérêt que tu as porté à mes travaux, Jacques, lors du séminaire que le LIFC m'a invité à venir faire à Besançon, ainsi que lors de nos échanges sur mon manuscrit m'ont permis de creuser certains aspects qui étaient encore restés superficiels ou mal expliqués. De même, Michael, les remarques très fines que tu m'as faites sur la définition proposée des systèmes hiérarchiques m'ont permis de corriger certaines erreurs insidieuses que j'avais glissé dans mon manuscrit. Pour tout cela : merci.

Enfin, en y regardant de plus près, la thèse ne représente pas seulement de longues heures à préparer, seul, l'argumentaire d'une idée rien moins que révolutionnaire qui va être balayée d'une remarque d'autant plus agaçante qu'elle est vraie, mais c'est aussi une femme, un mariage, des collègues, des amis, une vie. Si je devais énumérer l'ensemble de toutes les personnes ayant influées sur ma vie tout au long de ces quelques années, et donc sur ma thèse, il me faudrait prévoir autant de place que pour le manuscrit lui-même. Cependant, si je ne dois citer qu'eux, alors je voudrais remercier ma femme, a qui j'ai déjà dû faire regretter souvent de m'avoir épousé ; mes amis et collègues de « *la bande joyeuse* », qui n'ont toujours pas fini de me faire rire ; mes amis de « *le groupe* » (autrement appelés « *les gens* »), que j'ai perdus de vue en milieu de thèse mais auxquels je pense encore souvent ; Paul Amblard, Michel Burlet et plus généralement tous mes collègues enseignants pour leur aide, leurs conseils et la patience dont ils ont pu faire preuve ; mes collègues du LIG et plus particulièrement de l'équipe VASCO, avec qui j'ai toujours beaucoup de plaisir à discuter ; et enfin, les derniers en date, mais pas des moindres, je tiens particulièrement à remercier tous les membres passés et présents de l'équipe Démons, qui ont pris le temps de m'accueillir chaleureusement dans une ambiance studieuse mais néamoins détendue, et qui m'ont fait découvrir leur interprétation du mot « *équipe* ».

Enfin, je remercie tous les anonymes : les moteurs cachés de la recherche (Pascal Poulet, Carol Pasanisi, Gilles Thieblemont, Christiane Plumere, Dominique Moreira, etc.), les relecteurs ne rentrant pas dans les catégories précitées (Emmanuelle Rey), Les victimes consentantes de mes répétitions de pré-soutenances à répétition (Yannick Moy, Johannes Kanig, Aurélien Oudot, Florence Plateau, Christine Paulin, Louis Mandel, Amal Haddad, Pierre Berlioux, etc.), ma famille (parents, frère, grands-parents, oncles, tantes, cousins, etc.), les plongeurs (Corinne Fontaine-Lanoë, Jean-Marc Koch, Soline Schmauch, Remy Moesch, etc.), les motards (Martin Heusse, Pascal Sicard, Julien Tomassone, etc.), les amis éloignés (Alexandre Demeure, Marc Salvati, Claude, Olivier Leclere, Nicolas Chalons, Jean-François Millithaler, Sébastien Walger, etc.), les auteurs de scripts latex sur internet (masterbech, E.Depardieu, Arnau :D, etc.), les chats trop mignons (Chipie, etc.), ...

Je dédicace cette thèse à Catherine, professeur des écoles de son état, et qui a choisi comme étrange but dans la vie de faire mon bonheur éternel. Surtout, ne change rien.

Table des matières

Résumé 1

I Introduction 3

1 Motivations et contexte 5
- 1.1 Introduction . . . 5
- 1.2 Méthodes formelles . . . 7
  - *1.2.1 Introduction* . . . 7
  - *1.2.2 Apports des méthodes formelles* . . . 8
  - *1.2.3 Quelques limites des méthodes formelles* . . . 12
- 1.3 Première introduction à la méthode B . . . 14
  - *1.3.1 Utilisations industrielles* . . . 14
  - *1.3.2 Différentes approches liées à la méthode B* . . . 15
  - *1.3.3 Raffinement : pierre angulaire du développement selon la méthode B* . . . 15
- 1.4 Problématique et contributions . . . 16
  - *1.4.1 Problématique* . . . 16
  - *1.4.2 Complémentarité des vues : orientées données / comportements* . . . 18
  - *1.4.3 Visualiser le lien entre deux niveaux de description* . . . 19
  - *1.4.4 Appliquer la méthode à la vérification de propriétés* . . . 20
- 1.5 Organisation de ce document . . . 21

2 De l'orienté données à l'orienté comportements : un état de l'art 23
- 2.1 Langages informels de description de comportements . . . 24
- 2.2 Langages de description orientés comportements . . . 25
  - *2.2.1 Systèmes de transitions concrets (**STC**)* . . . 26
  - *2.2.2 Systèmes de transitions symboliques (**STS**)* . . . 27
  - *2.2.3 Systèmes de transitions hiérarchiques* . . . 30

2.3 D'une description orientée données vers une description comportementale 33
2.3.1 Description explicite des comportements 34
2.3.2 Extraction des comportements d'un modèle 35
2.3.3 Représentation de lien de raffinement 39
2.4 Vérification de propriétés comportementales 39
2.5 Synthèse 41

3 La méthode B **43**
3.1 Notions ensemblistes 44
3.2 Description par l'exemple d'un composant B 44
3.3 Les substitutions généralisées 48
3.3.1 Substitutions primitives 49
3.3.2 Substitutions dérivées 50
3.3.3 Forme normalisée d'une substitution 51
3.4 Calcul des obligations de preuve 52
3.5 Raffinement 54
3.5.1 Exemple de raffinement B 54
3.5.2 Le raffinement en B 56
3.6 L'approche B événementiel 58
3.6.1 Introduction 59
3.6.2 Exemple de système B événementiel 59
3.6.3 Calcul des obligations de preuve 60
3.6.4 Le raffinement en B événementiel 61
3.6.5 Exemple de raffinement B événementiel 61
3.7 Outils 63
3.8 Synthèse 63

II Contributions **65**

4 Calcul et représentation des comportements d'un système B événementiel **67**
4.1 Choix d'un formalisme de description des comportements 69
4.1.1 Motivations 69
4.1.2 Systèmes de transitions étiquetées symboliques 70
4.1.3 Forme normalisée d'un événement B 72
4.2 Extraction des comportements d'un système B 73
4.2.1 Définition de l'espace d'états 73

4.2.2 Construction de la relation de transition 75

4.3 Lien entre un **STES** et le système B dont il est issu 80

4.3.1 Sémantique d'un système B événementiel 81

*4.3.2 Sémantique d'un **STES** (Système de transitions étiquetées symbolique)* 81

4.3.3 Égalité des traces 82

4.4 Critères de choix des états 83

4.4.1 Exemple de mise en évidence d'une propriété 83

4.4.2 Techniques de choix d'états 84

4.4.3 Bilan 88

4.5 Application au B classique 88

4.5.1 Traduction d'une machine B en système B événementiel 89

4.5.2 Sémantique de la transformation proposée 90

4.5.3 Externalisation des paramètres 91

4.6 Synthèse 95

5 Représentation des comportements de raffinements B événementiel 97

5.1 Choix d'un formalisme de description 98

5.1.1 Définition des systèmes de transitions hiérarchiques 99

*5.1.2 Représentation des **STEH*** 102

5.2 Construction des comportements d'un raffinement B événementiel 105

5.2.1 Définition de l'espace d'états 107

5.2.2 Construction de la relation de transition sur les états-feuille 108

5.2.3 Réduction du nombre de transitions externes 110

5.3 Illustration de la construction d'un **STEH** 117

5.4 Application au B classique 119

5.5 Synthèse 119

III Outils et exemples d'application 121

6 L'outil *GénéSyst* 123

6.1 Comportements d'une spécification abstraite 123

6.1.1 Choix de l'espace d'états 124

6.1.2 Construction de la relation de transition 125

6.1.3 Exemple d'utilisation 129

6.2 Prise en compte du raffinement 130

6.2.1 Choix de l'espace d'états 131

6.2.2 Construction de la relation de transition 131

6.3 Expérimentations 132
6.3.1 *Batterie de tests de GénéSyst* 132
6.3.2 *Études de cas* 138
6.4 Généralités 140
6.5 Synthèse 141

7 Utilisation de *GénéSyst* au sein du projet GECCOO 143
7.1 Présentation du projet GECCOO 144
7.2 Introduction aux cartes à puce 146
7.2.1 *Technologies des cartes à puce* 146
7.2.2 *JavaCard : un OS pour cartes à puce* 146
7.2.3 *Sécurité des cartes à puce* 147
7.3 Étude de cas DEMONEY 148
7.3.1 *Description du modèle* 148
7.3.2 *Objectifs de sécurité* 154
7.4 Vérification de propriétés de sûreté 156
7.4.1 *Vérification de propriétés invariantes* 156
7.4.2 *Vérification de propriétés décrites par des automates* 158
7.4.3 *Vérification de propriétés décrites en SEPL* 160
7.5 Synthèse 166

IV Bilan 167

8 Conclusion et perspectives 169
8.1 Conclusion 169
8.2 Perspectives 172

V Annexes 175

A Complément sur les substitutions généralisées 177
A.1 Calcul de la terminaison 177
A.2 Calcul du prédicat avant-après 177
A.3 Calcul de $\mathcal{WP}$ conjugué 178

B Démonstrations 179

B.1 Forme normalisée d'un événement 179
B.2 Trace d'un système B événementiel 180
B.3 Prise en compte du raffinement : implication des conditions des transitions 180
B.4 *GénéSyst* : Simplification du calcul des conditions d'atteignabilité 182

C Glossaire **185**

Index **193**

Liste des illustrations **195**

Liste des théorèmes, définitions, lemmes, conditions et propriétés **199**

Bibliographie **201**

Résumé

Cette thèse propose une approche d'aide à la conception et au développement de modèles formels B. Cette approche se base sur la construction d'un système de transitions symboliques décrivant les comportements du modèle. Cette seconde vue est complémentaire avec la description orientée données du modèle B et peut être utilisée pour le décrire, le documenter ou le valider. Le système de transitions est élaboré à partir d'un espace d'états fourni par l'utilisateur et les transitions sont construites par résolution d'obligations de preuve. Nous proposons également de prendre en compte le processus de raffinement B en introduisant la notion de hiérarchie dans les systèmes de transitions. Cette représentation permet de mettre en évidence le lien entre les données des différents niveaux de raffinement. De plus, la méthode que nous proposons se base sur la décomposition des états d'une représentation du modèle abstrait, permettant ainsi de conserver la structure générale du système. Enfin, nous terminons ce manuscrit en décrivant l'outil *GénéSyst* qui implante cette méthode, ainsi que son utilisation dans le cadre du projet GECCOO, pour la vérification de propriétés de sécurité.

Mots clefs Méthode B, Systèmes de transitions, Raffinement, Vues, Aide au développement, Validation

Abstract

This thesis presents a new approach to help in the design and development of B models. This approach is based on the construction of a symbolic labeled transition system which describes the models behaviors. This description completes the data oriented description provided by the B model. It can also be used to document it or to validate it. The transition system is constructed from a user-defined data space on which transition relation is computed by solving proof obligations. We also propose to take into account the B refinement process by introducing some hierarchy in the transition systems. This representation allows exhibiting the link between data from several refinement levels. Moreover, the proposed method works by partitioning the states space of the more abstract description. This makes it possible to keep the abstract system global structure. Finally, the manuscript ends with a description of the *GénéSyst* tool, which implements the proposed method. We describe its use in the framework of the GECCOO project, in order to verify some security properties.

Keywords B Method, Transitions Systems, Refinement, Views, Development aid, Validation

Première partie

Introduction

Motivations et contexte 1

Notations are a frequent complaint... but the real problem is to understand the meaning and properties of the symbols and how they may and may not be manipulated, and to gain fluency in using them to express new problems, solutions and proofs.

C.A.R. Hoare

Sommaire

1.1 Introduction 5
1.2 Méthodes formelles 7
1.2.1 Introduction 7
1.2.2 Apports des méthodes formelles 8
1.2.3 Quelques limites des méthodes formelles 12
1.3 Première introduction à la méthode B 14
1.3.1 Utilisations industrielles 14
1.3.2 Différentes approches liées à la méthode B 15
1.3.3 Raffinement : pierre angulaire du développement selon la méthode B 15
1.4 Problématique et contributions 16
1.4.1 Problématique 16
1.4.2 Complémentarité des vues : orientées données / comportements 18
1.4.3 Visualiser le lien entre deux niveaux de description 19
1.4.4 Appliquer la méthode à la vérification de propriétés 20
1.5 Organisation de ce document 21

1.1 Introduction

Les systèmes informatiques se démocratisent et trouvent des domaines d'application de plus en plus variés. Leur coût toujours plus réduit et leur miniaturisation permettent de les intégrer dans un nombre croissant d'appareils de la vie courante. Parmi ceux-ci, nous pouvons citer les cartes à puce à microprocesseur intégré, évolution des cartes à piste, qui sont de véritables ordinateurs portatifs. Celles-ci sont notamment utilisées dans le domaine bancaire comme cartes de paiement

et dans la téléphonie mobile comme certificats d'identification du compte à débiter (cartes SIM des téléphones portables).

Un système informatique est dit *critique* si une défaillance peut avoir des conséquences humaines (blessure ou mort), sociétales (perte d'argent, dégradation de l'image de la société, etc.) ou environnementales (pollution, conséquences animales, etc.). La figure 1.1 donne deux exemples, parmi des milliers, de conséquences liées à des défaillances de systèmes informatiques critiques et montrant à quel point il est important de pouvoir y garantir un minimum de sécurité.

Les pannes de la société de l'information
UNE PANNE du réseau informatique de la SNCF a paralysé, vendredi 3 décembre, des centaines de terminaux de vente de billets aux guichets des gares. Une panne de même nature avait déjà affecté l'entreprise pendant quatre jours au mois de juillet.
Source : Le Monde – 05/12/2004

Prius hybrids dogged by software
A software problem is causing some Toyota Prius gas-electric hybrid cars to stall or shut down while driving at highway speeds, according to a published report.
Source : CNN/Money – 16/05/2005

FIG. 1.1 – Des défaillances des systèmes informatiques peuvent avoir de graves conséquences

J-C. Laprie [Lap95] définit la sécurité d'un logiciel comme *la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre.* Il propose également une décomposition en six attributs pour définir la sécurité des logiciels, aussi appelée *sûreté de fonctionnement* :

- la **disponibilité**, qui correspond à l'accessibilité sans interruption du système ;
- la **fiabilité**, qui correspond à la continuité du service ;
- la **confidentialité**, qui correspond à la non-divulgation d'informations ;
- l'**intégrité**, qui correspond à l'impossibilité d'altérer des informations ;
- la **maintenabilité**, qui correspond à pouvoir réparer et faire évoluer le système ;
- la **sécurité-innocuité**, qui correspond à l'absence de défaillances *catastrophiques.*

Les erreurs catastrophiques, qui correspondent à l'aspect sécurité-innocuité, sont celles ayant des conséquences sociétales, environnementales ou humaines.

Certaines conséquences liées à l'utilisation de logiciels critiques sont limitées par les gouvernements. Par exemple, la législation française impose que l'utilisation d'un système technologique ne conduise pas à une mortalité supérieure à celle due à des causes naturelles [LCGP89]. Ainsi, le nombre maximum de défaillances catastrophiques par heure est fixé pour chacun des systèmes critiques, en prenant en considération un certain nombre de facteurs, tels que la gravité des conséquences d'une défaillance, le nombre de personnes susceptibles d'être affectées et la durée d'exposition. Par exemple, le logiciel embarqué d'un métro ne doit pas avoir plus de 10^{-12} panne de sécurité par heure tandis que celui d'un avion civil de transport de personnes ne devra pas avoir plus de 10^{-9} panne de sécurité par heure.

Dans certains domaines, on peut essayer de quantifier le concept de *criticité*. Par exemple, la

méthodologie de l'AMDEC[1], qui est intégrée dans la norme ISO 9000 et qui porte sur la sûreté de fonctionnement dans les domaines avionique et automobile, définit la criticité en fonction de la fréquence des défaillances, de la gravité de leurs conséquences et de la probabilité de ne pas pouvoir les détecter[2] :

$$\textit{Criticité} = \textit{Fréquence} \times \textit{Gravité} \times \textit{Probabilité de non détection}$$

Toutefois, certains facteurs peuvent changer en fonction des domaines d'application.

Cependant, la sécurité d'un logiciel ne se mesure pas. On considère généralement que le niveau de confiance que l'on a en un logiciel varie en fonction du soin apporté à son développement. C'est sur ce concept que se basent les différents processus d'évaluation de la sûreté de fonctionnement des logiciels critiques. Par exemple, la norme DO-178B [SC-92] définit les principes de développement et de vérification des logiciels aéronautiques et la norme ISO/IEC 15408 [Com99a, Com05] définit des critères d'évaluation de la sécurité des technologies de l'information. Afin d'établir qu'un système critique a été correctement conçu, ces différents critères imposent l'utilisation de méthodes de génie logiciel, telles que les méthodes formelles (méthodes basées sur les mathématiques), qui permettent d'atteindre un haut niveau de confiance dans le code produit. Cependant, comme nous le détaillons dans la suite, de telles méthodes ne constituent pas la solution à tous les problèmes. Par exemple, elles nécessitent souvent une grande expertise, ce qui limite les échanges possibles entre client et concepteur.

Dans cette thèse, nous nous intéressons au développement de modèles selon la méthode formelle B. Avant de présenter cette dernière, nous proposons de nous intéresser aux principales approches formelles existantes en détaillant notamment leurs apports et leurs limites. Parmi les concepts clefs des méthodes formelles, nous nous focalisons particulièrement sur la notion de raffinement, car cette méthode de développement, basée sur une description de plus en plus en fine du modèle, est la pierre angulaire de la méthode B. Nous décrivons ensuite la méthode B, avant de terminer en détaillant nos contributions.

1.2 Méthodes formelles

1.2.1 Introduction

Le terme de génie logiciel est une traduction de l'anglais *Software Engineering* [BBH68] qui est apparu en France en 1984 [mdd84]. Il est défini dans le journal officiel de la République Française comme « *l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi* ». Il répond à un besoin de l'industrie du logiciel, puisque d'après une étude réalisée en 1994 aux États-Unis[3], 31% des logiciels étaient abandonnés pendant leur développement ou jamais utilisés, 53% des développements logiciels dépassaient largement les coûts estimés et seulement 16% des logiciels étaient réalisés selon

[1] Analyse des modes de défaillance, de leurs effets et de leur criticité
[2] http://fr.wikipedia.org/wiki/Analyse_des_modes_de_d%C3%A9faillance,_de_leurs_effets_et_de_leur_criticit%C3%A9
[3] Rapport Chaos du Standish Group : http://www.standishgroup.com/sample_research/chaos_1994_1.php

les plans. Différents cycles de vie du logiciel ont alors été proposés pour aider à structurer les développements, suivant les besoins et la taille du produit (Cycle en V, par itération, etc.). Ces cycles de développement se décomposent chacun en trois grandes étapes : l'analyse, la conception et la validation. La phase d'analyse consiste à définir le cahier des charges en partenariat avec le client, en mettant en évidence les exigences fonctionnelles et les propriétés de sécurité voulues. La conception consiste à franchir le pas qui sépare le cahier des charges et le code du système. Enfin, la phase de validation consiste à vérifier que le système développé s'exécute sans erreur, qu'il est conforme aux exigences fonctionnelles et qu'il vérifie les propriétés attendues. La figure 1.2 résume ces étapes sur l'exemple du cycle de développement en V.

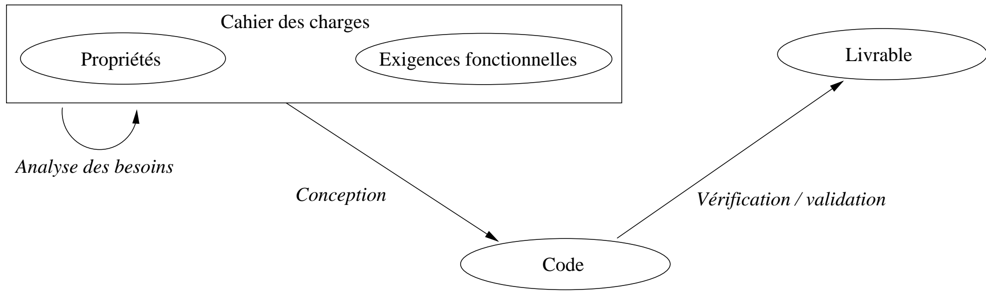


Fig. 1.2 – Principe du cycle de développement en V

Les cahiers des charges sont souvent rédigés en langue naturelle (français, anglais, etc.), laissant ainsi une très (trop ?) grande liberté d'interprétation lors des phases de conception et de validation. Pour affiner cette description et lever certaines ambiguïtés, il est alors possible d'utiliser des méthodes d'analyse et de conception semi-formelles (basées sur des langages dont la sémantique est partiellement définie) ou formelles (basées sur des langages dont la sémantique est totalement définie et non ambiguë). Pour qu'une méthode de développement soit dite formelle, elle doit fournir à l'utilisateur des concepts et des notations lui permettant de décrire le système, les propriétés ou la spécification voulue et de raisonner sur les descriptions effectuées. Il est alors nécessaire que la sémantique soit mathématiquement bien définie. De plus, il est important qu'une méthode formelle soit outillée, afin de garantir la correcte application de ses règles et faciliter la détection d'erreurs.

1.2.2 Apports des méthodes formelles

Dans cette section, nous voulons donner au lecteur une intuition des différentes approches formelles existantes. Nous n'avons pas la prétention de les décrire toutes exhaustivement, mais nous essayons de donner un maximum de pointeurs pour orienter le lecteur intéressé dans ses recherches. De plus, les garanties obtenues par l'utilisation d'une méthode formelle dépendent des techniques de vérification utilisées. C'est pourquoi, nous détaillons, dans la section 1.2.2.3, les principales d'entre elles, les garanties qu'elles peuvent apporter et leurs limitations. L'objectif de cette section est de situer la méthode B au milieu du panorama des approches existantes et de réaliser que les limitations liées à l'utilisation de cette méthode, se retrouvent dans les autres méthodes formelles.

1.2.2.1 Principales approches des méthodes formelles

Les méthodes formelles peuvent être utilisées lors de chacune des étapes du cycle de développement pour aider à structurer le raisonnement et apporter des garanties sur le développement.

Durant la phase d'analyse, l'utilisation de méthodes formelles permet d'imposer une certaine rigueur de rédaction du cahier des charges et un effort de formalisation. L'intérêt est alors d'être amené à se poser les bonnes questions, qui faciliteront ensuite le processus de développement. Cette phase consiste donc à réécrire les exigences fonctionnelles et les propriétés de sécurité du cahier des charges dans un ou des langages formels, puis à vérifier la cohérence[4] de la composition de ces descriptions. On pourra, par exemple, utiliser des formalismes basés sur des formules logiques, tels que Z [Abr80, Dil94, Spi93] ou TLA [Lam94b], des formalismes orientés modification de données, tels que B [Abr96b], des formalismes fonctionnels, tels que Coq [BC04, PM93], etc. Les modèles ainsi décrits sont également utilisables pour concevoir et valider le système.

Durant la phase de conception, certaines méthodes se basent sur une approche descendante pour faire le lien entre le modèle formel décrivant le cahier des charges et le code produit. L'objectif est alors d'avoir certaines garanties sur le code produit et donc de limiter les risques de non conformité du système par rapport au cahier des charges. Par exemple, certaines méthodes telles que Coq permettent de dériver automatiquement un programme à partir de la preuve des propriétés attendues (en utilisant les propriétés de l'isomorphisme de Curry-de Bruijn-Howard [GLT89]). D'autres méthodes, telles que Z ou B, se basent sur un processus de raffinement qui consiste à décrire les spécifications de plus en plus précisément. Il est alors possible d'atteindre un niveau de description qui répond aux contraintes machines (taille mémoire, déterminisme des instructions, etc.). Les principes du raffinement sont décrits plus en détail dans la section suivante.

Enfin, **durant la phase de validation**, d'autres méthodes se basent sur une approche ascendante, où le modèle formel n'est pas nécessairement décrit avant le code. L'objectif est alors de vérifier la conformité du programme développé par rapport au cahier des charges. Pour ce faire, il est possible soit d'exploiter les modèles décrits durant les phases d'analyse ou de conception, soit d'annoter le code avec des assertions[5] permettant de décrire les exigences fonctionnelles et les propriétés de sécurité voulues. On pourra alors utiliser, par exemple, les formalismes JML [LBR98, LBR99] ou SPEC# [BDJ+05]. Parmi ces méthodes, nous pouvons citer le test basé sur les modèles [ABC+02], la vérification dynamique de contraintes [LBR98] ou la preuve de programmes.

Dans cette thèse, nous nous intéressons à assister un concepteur durant les différentes phases du développement d'un système selon la méthode B. L'un des principaux atouts de cette méthode est l'intégration d'un processus de développement par raffinement. C'est pourquoi, nous proposons de décrire plus en détail les différents aspects de ce processus, avant de nous intéresser aux techniques de vérification.

[4] De manière générale, un modèle formel est dit *cohérent* s'il ne contient pas de contradiction (s'il en existe un modèle).

[5] Propriétés logiques insérées dans un code qui doivent être vérifiées. On s'intéresse notamment aux pré/post-conditions, assertions vraies avant et après une opération, et aux invariants, qui doivent être toujours vrais.

1.2.2.2 Une approche de développement : le raffinement

La notion de raffinement est comparable à la modification de l'échelle d'une carte routière. Par exemple, une carte routière à l'échelle 1/2 000 000 permet de considérer la France entière, mais ne permet de représenter que les grands axes (les autoroutes ou nationales principales). Pour représenter toutes les nationales, ainsi que les principales départementales, il faut alors considérer une représentation à l'échelle 1/100 000 de cette carte. Dans cette seconde représentation, le tracé des grands axes est celui décrit dans la carte à faible échelle, mais une information supplémentaire a été ajoutée : le tracé des grandes départementales. On dira donc que la carte au 1/2 000 000$^{\text{ème}}$ est une abstraction de la carte au 1/100 000$^{\text{ème}}$, qui est alors appelée un raffinement.

De la même manière, raffiner une spécification formelle consiste à décrire plus finement ses algorithmes, afin de s'approcher, par étapes, d'une description implantable. Un composant de raffinement est une spécification à part entière dans laquelle des informations supplémentaires sont fournies pour lier les données et les actions de cette description à celles d'une autre, plus abstraite. Dans le cas de notre carte routière, le lien de raffinement pourrait être le facteur d'échelle. Pour chaque niveau de raffinement on s'intéresse donc à vérifier non seulement la cohérence de la description, mais aussi sa conformité par rapport à son abstraction.

Notons qu'une surcharge est faite sur le terme de raffinement qui désigne, suivant le contexte, soit l'action de raffiner, soit un composant donnant une description plus fine d'un autre composant. On appelle alors spécification abstraite une description n'étant liée à aucune autre description plus abstraite. Chaque composant de raffinement raffine donc, soit une spécification abstraite soit un raffinement. Nous appellerons le $n^{\text{ème}}$ niveau de raffinement d'un modèle, un composant raffinant une chaîne de $n-1$ niveaux de description plus abstraits.

Le premier intérêt du raffinement se trouve dans le fait que l'on commence par décrire une abstraction du système. On y décrit uniquement le cœur du cahier des charges en y masquant les détails, ce qui simplifie la compréhension et les vérifications. Par exemple, pour trouver une route allant de Grenoble à Paris, il est plus simple d'utiliser une carte *abstraite* au 1/2 000 000$^{\text{ème}}$ plutôt qu'une carte au 1/100 000$^{\text{ème}}$, car cela permet de ne pas se noyer dans la masse d'informations et de détails.

Le second intérêt est d'introduire les détails, et donc la complexité, par étapes. En effet, se baser sur une abstraction permet de structurer l'ordre et la manière d'introduire les propriétés du cahier des charges. Le raffinement permet alors d'introduire les propriétés secondaires tout en préservant les propriétés principales issues de la spécification abstraite. Typiquement, les propriétés dites de sûreté[6] sont toujours préservées par raffinement. C'est-à-dire que si celles-ci sont vérifiées par la spécification abstraite alors elles sont nécessairement vraies dans tous ses raffinements. Par exemple, s'il existe une route menant de Grenoble à Paris sur la carte au 1/2 000 000$^{\text{ème}}$, alors il existe aussi une route sur la carte au 1/100 000$^{\text{ème}}$.

Comme le processus de raffinement permet d'introduire progressivement la complexité des modèles et des algorithmes et qu'il préserve les propriétés de sûreté, il s'ensuit que la vérification de la cohérence des modèles est simplifiée. Par exemple, considérons d'une part un modèle mo-

[6] Rien de mauvais n'arrivera jamais. Par opposition aux propriétés de vivacité (quelque chose de bien arrivera fatalement un jour).

nolithique M_m, décrit sans raffinement, et un modèle M_r équivalent, dont les différents aspects sont introduits progressivement. Dans les deux cas, nous obtenons la même garantie de cohérence. Mais dans le cas du modèle M_r, comme les informations sont introduites progressivement, il est possible de simplifier les vérifications liées au raffinement en exploitant les propriétés établies au niveau d'abstraction supérieur. Cependant, il devient nécessaire de vérifier, en plus, la conformité des différents niveaux de raffinement.

Enfin, l'approche par raffinement permet une certaine adaptabilité du processus de développement, car la remise en question d'un choix effectué au $n^{\text{ème}}$ niveau de raffinement ne remet pas en cause ses niveaux de descriptions plus abstraits.

1.2.2.3 Différentes techniques de vérification

Quelle que soit la méthode formelle utilisée, il est nécessaire de vérifier la cohérence du modèle et la conformité de ses éventuels raffinements. Cependant, la logique du premier ordre est indécidable en général. Il s'ensuit que la pertinence de la réponse, et donc les garanties obtenues, dépend de la manière de vérifier ces propriétés. Il existe alors principalement trois approches : restreindre l'expressivité du langage de description des modèles pour se ramener à de la logique décidable, raisonner sur une sous-approximation du modèle pour trouver des exemples d'erreurs ou raisonner sur une sur-approximation du modèle pour garantir sa conformité. Cependant, dans ces deux dernières approches, si le but n'est pas atteint (pas de contre-exemple ou pas de preuve de conformité), alors il n'est pas possible de conclure. Parmi les techniques de vérification les plus utilisées nous trouvons l'interprétation abstraite, le model-checking, la preuve, l'animation et le test.

Les techniques basées sur l'animation et le test ne sont pas toujours considérées comme des techniques de vérification dans le sens où elles permettent d'exhiber des erreurs mais pas d'en garantir l'absence. En effet, pour une unique trace d'exécution de longueur finie, elles peuvent toujours conclure sur le respect d'une propriété, mais le problème est alors de couvrir l'ensemble, possiblement infini, des exécutions (elles mêmes possiblement infinies).

L'interprétation abstraite [Cou00], quant à elle, consiste à raisonner statiquement sur une abstraction d'un système. La difficulté est alors de trouver cette abstraction, en exhibant ses fonctions d'abstraction et de concrétisation (connexions de Galois [CC77]), de telle sorte que la résolution du problème y soit décidable. De plus, il faut que le respect de la propriété par l'abstraction implique le respect de la propriété par le modèle. L'indécidabilité de la vérification est donc transférée dans le choix de l'abstraction.

Le model-checking, pour sa part, est généralement décidable, car il est utilisé sur des modèles ayant un nombre fini d'états. Cette technique consiste à vérifier que les traces d'exécution d'un modèle sont incluses dans les traces autorisées par une propriété. Pour traiter des modèles ayant un nombre potentiellement infini d'états, il est possible de ne s'intéresser qu'à une abstraction du système, en couplant cette approche à des techniques d'interprétation abstraite. On se heurte alors aux limites de l'interprétation abstraite et à l'indécidabilité du choix de l'abstraction.

Enfin, les techniques basées sur la preuve permettent de raisonner sur l'ensemble des traces d'exécution d'un système et de conclure si une propriété est vérifiée ou non. Le principe est de générer des obligations de preuve (des formules logiques) à partir du modèle et de la propriété, de

telle sorte que, si elles sont prouvées, alors le modèle vérifie nécessairement la propriété. La difficulté se trouve alors dans la recherche d'une preuve permettant d'établir le respect de la propriété voulue.

1.2.2.4 Bilan

Ainsi, utiliser des méthodes formelles offre un ensemble de techniques qui permettent de structurer le raisonnement et de prévoir, au plus tôt, les difficultés à venir. Cela permet donc d'améliorer la qualité du développement des logiciels et donc aussi d'augmenter la confiance que l'on a en eux. Par exemple, les Critères Communs pour la certification de systèmes d'information [Com05] nécessitent, pour les notes les plus élevées, qu'une description formelle du système soit fournie. De plus, suivant la note de certification visée, il peut être exigé que le système soit décrit en utilisant un processus de raffinement. Par exemple, pour obtenir la note la plus élevée (EAL7), il est nécessaire que 5 niveaux de description soient fournis[7] et que la preuve de la conformité du raffinement soit formellement effectuée. Cependant, un développement par raffinement n'est pas une solution à tous les problèmes. Dans la section suivante, nous mettons en évidence les principales limitations des méthodes formelles.

1.2.3 Quelques limites des méthodes formelles

1.2.3.1 Difficultés liées à la modélisation

Les méthodes formelles fournissent un cadre rigoureux de raisonnement permettant d'établir qu'une spécification vérifie certaines propriétés. Pour ce faire, il est nécessaire que la propriété soit exprimée dans un formalisme permettant la comparaison avec la spécification. Cependant, les propriétés et les exigences fonctionnelles sont généralement décrites de manière informelle en langue naturelle (français, anglais, etc.), avec parfois quelques diagrammes semi-formels. Comme cette étape de formalisation du cahier des charges n'est pas automatisable, seule l'expertise des concepteurs permet de garantir la conformité du modèle par rapport au cahier des charges. En effet, comme les modèles formels ne sont pas nécessairement très intuitifs, les clients ne peuvent que difficilement confirmer ou infirmer les choix effectués dans la réalisation des modèles [BH06]. De plus, les spécifications sont ensuite utilisées d'une part pour la conception et le développement du système et, d'autre part, pour sa validation finale. Il s'ensuit que la moindre erreur durant cette étape peut avoir de lourdes conséquences sur le développement à venir. Ainsi, l'expertise du concepteur est un facteur important dans les phases de modélisation et de validation.

1.2.3.2 Difficultés liées aux techniques de vérification

L'utilisation de techniques de vérification fait souvent appel à l'expertise du vérifieur. En effet, les différentes techniques de vérification qui ne sont pas indécidables sont généralement soit limitées à un nombre fini d'exécutions, soit restreintes à une logique décidable.

Cependant, pour qu'une technique telle que le model-checking ou l'animation puisse couvrir un ensemble plus important d'exécutions, on peut s'intéresser à vérifier une abstraction du système, en

[7] (1) Spécification globale de la cible de sécurité / (2) Spécification fonctionnelle / (3) Conception de haut niveau / (4) Conception de bas niveau / (5) Implantation

utilisant des techniques d'interprétation abstraite. On se ramène alors à un problème d'indécidabilité, lié en même temps au choix de l'abstraction et à la vérification de la propriété sur l'abstraction.

De la même manière, même si la vérification du respect d'une propriété par le modèle est décidable, une trop grande complexité peut interdire l'utilisation de vérificateurs automatiques. Pour diminuer cette complexité, il est possible de décomposer le problème soit *horizontalement*, en réalisant une description modulaire du système, soit *verticalement*, en décrivant le système par raffinement. Cependant, cette approche ne garantit pas de toujours diminuer la complexité de la vérification. En effet, il faut pour cela effectuer des choix pertinents qui permettent d'établir la propriété par parties ou sur la description abstraite.

Ainsi, on se ramène dans tous les cas à un problème d'indécidabilité dans lequel seule l'expertise du vérifieur permet d'effectuer de bons choix.

1.2.3.3 Garanties obtenues par un développement formel

Enfin, le problème final est de savoir quelles sont les garanties que l'on a sur un système réalisé selon un processus de développement formel. Par exemple, dans le cadre de la certification d'un système selon les Critères Communs, les concepteurs définissent leur cible de sécurité (noyau sécuritaire du programme) et les propriétés de sécurité qu'ils estiment devoir établir. La note issue de l'évaluation caractérise alors, en fonction des techniques mises en œuvre durant les différentes phases de développement, le degré de confiance qu'ont les évaluateurs en la sécurité de la cible d'évaluation. Le processus de certification, qui dure classiquement plusieurs mois, est alors une longue interaction où les concepteurs vont devoir convaincre les évaluateurs des garanties obtenues sur le système. Par exemple, si des concepteurs avancent qu'ils ont établi par la preuve qu'un système respecte une propriété, alors un certain nombre de questions se posent, parmi lesquelles :

- **Confiance en la formalisation :** *la propriété décrite est-elle bien celle voulue ?*
- **Confiance en le générateur d'obligations de preuve :** *l'obligation de preuve générée est-elle bien celle voulue ?*
- **Confiance en le prouveur :** *la preuve est-elle correcte ?*
- **Confiance en le compilateur :** *la preuve ayant été établie par rapport à une spécification ou un code compilable, quelles garanties a-t-on sur le code compilé ?*

1.2.3.4 Bilan

Ainsi, l'expertise du spécifieur est un facteur important durant toutes les phases du développement d'un système. C'est pourquoi, il faut lui fournir des outils permettant de l'assister dans sa tâche. Dans cette thèse, nous nous intéressons au développement de modèles selon la méthode B. Celle-ci intègre un processus de développement par raffinement et se heurte aux différentes limites que nous venons de décrire. Dans les sections suivantes, nous commençons donc par décrire cette méthode, avant d'y instancier les limitations décrites. Nous nous focalisons alors sur les problèmes traités par cette thèse, que nous présentons sous la forme de nos contributions.

1.3 Première introduction à la méthode B

Dans cette section, nous introduisons informellement la méthode B. Celle-ci est décrite plus en détail dans le chapitre 3.

Introduite en 1986 par J-R Abrial, la méthode B [Abr96b] est une méthode formelle de développement, ainsi qu'un langage de spécification basé sur l'affectation. Très inspirée du formalisme Z, la méthode B s'en distingue par la possibilité de raffiner les développements jusqu'à leur implantation et par l'utilisation d'un langage de substitutions généralisées (nom donné aux *instructions* du langage B). Certaines de ces substitutions permettent de décrire des actions de manière non-déterministe. L'un des objectifs du raffinement est alors de réduire ce non-déterminisme en précisant les algorithmes. Dans cette méthode, la vérification de la cohérence d'un modèle ou de la correction d'un raffinement se fait par le biais d'un processus de génération et de vérification d'obligations de preuve.

Nous commençons par présenter la méthode B à travers ses utilisations industrielles. Ensuite, nous décrivons le processus de raffinement qu'elle intègre. Enfin, nous détaillons le processus classique de développement selon cette méthode, en faisant le lien avec les limitations mises en avant dans la section précédente et liées à l'utilisation des méthodes formelles.

1.3.1 Utilisations industrielles

La méthode B est issue des besoins de l'industrie [BDM97, Hab01, DM94]. Le projet SACEM[8], développé par GEC Alsthom Transport, est l'un des premiers à utiliser des systèmes pour lesquels la sécurité est assurée par logiciel. C'est ensuite avec le projet de métro sans conducteur parisien *Météor*[9] [BDM98, BBFM99, Beh96], mené par Matra Transport International[10] en 1998, que la méthode B a pris sa force et a vu naître l'outil qui est aujourd'hui l'*AtelierB* [Cle01]. Depuis, d'autres métros sans chauffeur ont vu le jour [Sys] notamment à Hong-Kong (en 2002), à New-York (ligne Canarsie en 2003) et d'autres chantiers sont en cours comme la ligne 1 du métro parisien (2010), la ligne 9 du métro de Barcelone (2008), la navette *Roissy VAL* de l'aéroport Roissy Charles de Gaulle [BA05] (2006) et la ligne 2 du métro de Budapest (2008).

Par la suite, la méthode B a également été appliquée dans les systèmes embarqués avec, par exemple, des travaux sur le diagnostic de panne dans le domaine de l'automobile [PP03] ou le développement de logiciels pour des modules d'aide aux malades diabétiques sous dépendance d'insuline [PLN03]. Cependant, cette dernière expérience s'est heurtée au faible espace mémoire disponible pour l'exécution de systèmes sur de telles plateformes. Cette même limitation a été atteinte par A. Requet et G. Bossu [RB00], lors de leur tentative pour embarquer, sur carte à puce, du code traduit automatiquement depuis le langage B. Suite à ces observations, le projet RNTL BOM [BBB$^+$04, RB00, BBP$^+$03, BCR03] a été lancé avec comme objectif la création d'un traducteur B vers C optimisant l'utilisation de l'espace mémoire. L'efficacité du traducteur produit a alors permis d'embarquer sur une carte à puce un système d'exploitation JavaCard développé en B.

[8] Système d'Aide à la Conduite, à l'Exploitation et à la Maintenance
[9] Ligne 14 du métro parisien
[10] aujourd'hui renommé SIEMENS Transportation Systems

En raison de leurs besoins en sécurité, les industriels de la carte à puce sont très présents dans le domaine de la méthode B, ce qui a motivé un grand nombre de travaux. Certains portent sur la certification [Mot00, MT00], tandis que d'autres études se focalisent sur les mécanismes de transaction [SL99, SL00], les protocoles [LL98] et les exceptions [BR03]. Avec l'arrivée des nouvelles générations de cartes basées sur le système JavaCard et permettant d'installer / désinstaller plusieurs applettes sur une même carte, certains travaux se sont orientés vers le développement d'un vérificateur de bytecode embarquable sur carte à puce [BCR03, LR98, Req00, CBR02, Cas02].

Ainsi, la communauté B, y compris certains industriels, est très active. L'une des explications à cet attrait se trouve dans le processus de développement par raffinement, ainsi que dans la facilité de prise en main du langage de spécification, proche des langages classiques de programmation impérative.

1.3.2 Différentes approches liées à la méthode B

La méthode B est un cadre général dans lequel on distingue l'approche B classique, aussi appelée B logiciel, et l'approche B événementiel, aussi appelée B système. En B classique, on s'intéresse à décrire un système informatique en terme d'opérations qui peuvent être appelées par un système externe (le système est alors dit ouvert), tandis qu'en B événementiel on s'intéresse principalement à l'analyse de cahiers des charges où à la conception d'architectures de systèmes [PP03] (le système est alors dit fermé). Un modèle B événementiel est décrit par des événements qui se déclenchent spontanément. Afin d'alléger le discours, dans la suite de cette thèse, nous parlons de la méthode B, lorsque nous ne voulons pas opposer les approches B classique et B événementiel.

Chaque opération du B classique est composée d'une pré-condition et d'une action. Une opération ne peut être appelée que si sa pré-condition est vérifiée. À l'inverse, chaque événement de B système est composé d'une garde et d'une action, de telle sorte que si sa garde est vérifiée, alors son action peut être exécutée. Intuitivement, les exécutions possibles d'un modèle B classique sont définies par le composant qui appelle ses opérations, tandis que les exécutions d'un modèle B événementiel sont déterminées par ses événements. Ainsi, ces deux approches se distinguent notamment par l'externalisation (systèmes ouverts) ou l'internalisation (systèmes fermés) du contrôle. Il s'ensuit que, contrairement au B classique, il est possible, en B événementiel, de raffiner le contrôle d'un modèle. Par exemple, un événement peut se déclencher moins souvent par raffinement et de nouveaux événements peuvent être introduits.

1.3.3 Raffinement : pierre angulaire du développement selon la méthode B

Un modèle B est décrit par une spécification abstraite et une suite de raffinements. Les spécifications et chaque niveau de raffinement sont décrits par des propriétés invariantes et des actions (des opérations ou des événements). On prouve alors, pour chacun des raffinements, sa cohérence (respect des propriétés invariantes par les actions) et sa correction vis-à-vis de la spécification ou de son niveau de description directement supérieur. De plus, si le raffinement est correct, alors les propriétés invariantes des niveaux supérieurs sont préservées par raffinement.

Enfin, dans le cas du B classique, le dernier niveau de raffinement peut être une implantation, c'est-à-dire un raffinement répondant aux contraintes machines, telles que le déterminisme des programmes et la représentation des données par des structures classiques dans les langages de programmation (entiers bornés, booléen, tableaux). Une implantation peut être traduite de manière automatique en un programme dans un langage de programmation (tels que C ou ADA). La figure 1.3 résume la chaîne de développement de la méthode B.

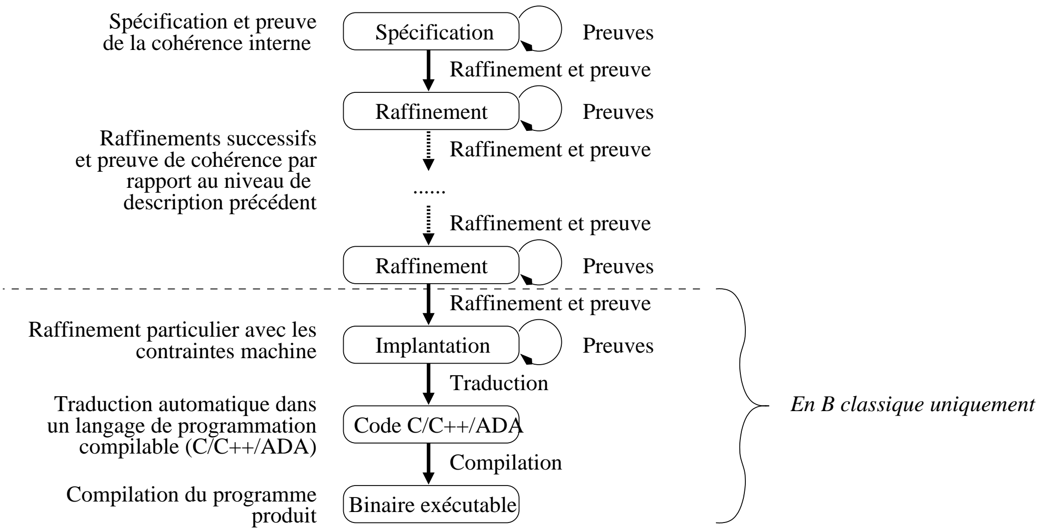


Fig. 1.3 – Processus général de développement selon la méthode B

À l'inverse du B classique, une spécification B événementiel n'est pas directement implantable, car elle est décrite par des événements qui se déclenchent spontanément et que le choix du déclenchement peut être non-déterministe. Par contre, le raffinement d'une spécification B événementiel permet de raffiner non seulement la représentation de ses données et ses algorithmes, comme en B classique, mais aussi son contrôle. Raffiner le contrôle consiste à modifier l'ensemble des comportements (des traces d'exécution) possibles du système. Pour ce faire, il est possible de renforcer les gardes des événements ou d'introduire de nouveaux événements. Cependant, pour que cette modification préserve la notion de raffinement, il est nécessaire que l'ensemble des traces d'exécution du raffinement soit inclus, modulo les nouveaux événements, dans l'ensemble des traces d'exécution de l'abstraction. L. Lamport [Lam94b] propose l'exemple d'une horloge donnant l'heure et les minutes qui peut être affinée pour donner aussi les secondes.

1.4 Problématique et contributions

1.4.1 Problématique

Comme nous l'avons vu, l'un des principaux verrous technologiques des développements logiciels en général, mais que l'on retrouve dans l'utilisation des méthodes formelles, est la vérification de la conformité d'un modèle par rapport à un cahier des charges. C'est pourquoi, nous proposons de

faciliter cette vérification, en mettant en évidence un autre aspect du modèle, complémentaire à sa description en langage B : ses comportements.

Terminologie. *Dans cette thèse, nous avons choisi d'appeler les* **comportements** *d'un modèle, l'ensemble de ses traces d'exécution ; c'est-à-dire l'ensemble des séquences d'événements ou d'opérations qu'il autorise.*

$$\begin{cases} Seq_1 = \xrightarrow{oc1}\ \xrightarrow{oc2}\ \xrightarrow{oc3}\ \xrightarrow{oc4}\ \xrightarrow{oc5}\cdots\cdots\xrightarrow{ocn} \\ Seq_2 = \xrightarrow{oc'1}\ \xrightarrow{oc'2}\ \xrightarrow{oc'3}\ \xrightarrow{oc'4}\ \xrightarrow{oc'5}\cdots\cdots\xrightarrow{oc'm} \\ \ldots \end{cases}$$

L'idée est d'aider le spécifieur en lui fournissant un autre point de vue sur son modèle. De plus, en utilisant un formalisme plus accessible aux non informaticiens, nous voulons donner la possibilité au client de vérifier lui même les principales propriétés du système. Enfin, nous proposons de prendre en compte le processus de raffinement B événementiel, qui permet de raffiner les comportements.

L'approche proposée (figure 1.4) consiste à calculer une représentation de l'ensemble des comportements d'un modèle B construit par raffinement. Cette approche a pour but l'aide à la compréhension du pas entre le formel et l'informel. Les descriptions ainsi produites peuvent donc être utilisées pour documenter ou présenter le modèle.

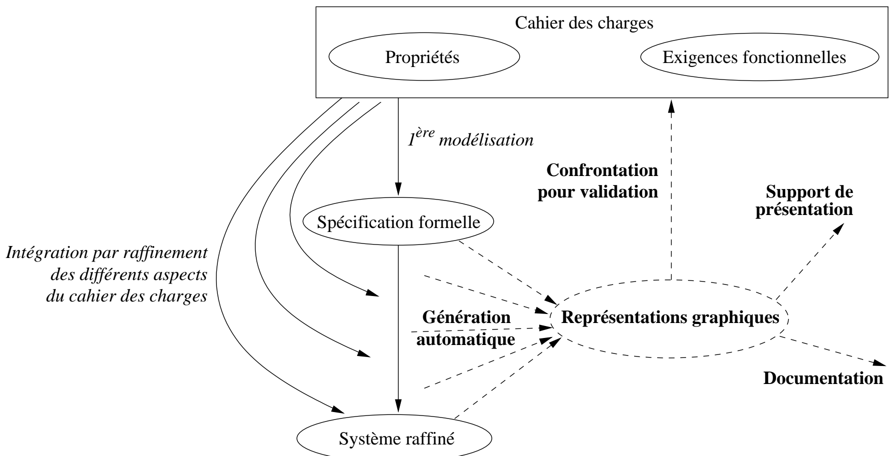


Fig. 1.4 – Intégration de notre approche dans le processus de développement

Un développement B événementiel commence généralement par la réalisation d'un diagramme informel décrivant les comportements attendus du système. Cela permet de structurer la réflexion pour formaliser les besoins et choisir les données à intégrer dans chaque niveau de description. Ce diagramme informel peut ensuite être ajouté au cahier des charges s'il est validé par le client. Il est alors intéressant de pouvoir comparer cette ébauche des comportements prévus du système avec une représentation de ceux réellement décrits. Nous nous intéressons donc aussi à la vérification de propriétés de sécurité.

Ainsi, au delà des outils, nos contributions visent les trois objectifs suivants :

– construire et représenter graphiquement les comportements d'une spécification B ;
– mettre en évidence le processus de raffinement dans les représentations d'un modèle ;
– exploiter la représentation comportementale d'un modèle pour le vérifier par rapport à une propriété.

Nous détaillons successivement chacun de ces points dans les trois sections suivantes.

1.4.2 Complémentarité des vues : orientées données / comportements

Un modèle B événementiel est décrit en termes de ses données, dans le sens où chaque événement est défini par une condition de déclenchement portant sur les valeurs des données et par une action modifiant ces données. Cependant, la sémantique d'un système B événementiel est définie par l'ensemble de ses comportements.

Bien que cette information des enchaînements possibles des événements soit sous-jacente dans tout système B événementiel, elle n'y est pas explicitement décrite. On n'écrit pas, par exemple, qu'« *après l'événement* oc_1*, l'événement* oc_2 *se déclenche* ». Classiquement, on oppose les descriptions basées sur les données aux descriptions comportementales (basées sur les événements). Les premières décrivent finement l'action effectuée par chaque événement sur les données, tandis que les secondes permettent de mettre en évidence des comportements. Les deux styles de description sont complémentaires, car ils donnent deux points de vue différents d'une même spécification.

Par exemple, les deux descriptions suivantes sont équivalentes et spécifient toutes deux que l'action S_1 est suivie de l'action S_2. Dans une description orientée comportement (Cas B.) on se contente de dire que S_1 est suivi de S_2, tandis que, dans une description orientée données (Cas A.), on *code* le comportement à travers une variable de contrôle C, qui permet de forcer l'exécution de l'action S_2 après l'exécution de S_1.

Initialisation : $C := 1$ $ev_1 \;\hat{=}\; C=1 \Longrightarrow (S_1 \parallel C:=2)$ $ev_2 \;\hat{=}\; C=2 \Longrightarrow (S_2 \parallel C:=1)$	$S_1 \;;\; S_2 \;;\; S_1 \;;\; S_2 \;;\; \ldots$
A. *Description orientée données* *(Le contrôle est codé dans C)*	**B.** *Description comportementale* *(Le contrôle est explicite)*

Intuitivement, la description orientée données est plus expressive. Elle est également intéressante pour la preuve de propriétés de sûreté, puisque l'on est amené à décrire, par des assertions, l'état du système entre deux occurrences d'événements. Cette information permet alors de décomposer les obligations de preuve, puisque l'on ne s'intéresse plus à la correction d'un enchaînement mais à celle de chacune des actions qui le composent. À l'inverse, la description comportementale permet de s'abstraire du codage (compliqué) du contrôle dans les événements. Ce second type de formalisme est donc mieux adapté à la vérification de propriétés dynamiques.

Le langage B est un formalisme de description orienté données. Dans notre approche, nous proposons donc une méthode permettant de construire une description comportementale d'un système décrit en B. Pour ce faire, nous proposons également un formalisme graphique de représentation. Il

existe de nombreux langages permettant de décrire les comportements d'un système (CSP [Hoa78], μ-calcul [Par74], etc.) ; il semble toutefois que les automates symboliques soient l'une des alternatives les mieux adaptées à notre objectif d'aide à la compréhension [Liv78]. En effet, ce formalisme graphique est universel et exhibe de manière intuitive les enchaînements d'événements qui peuvent survenir.

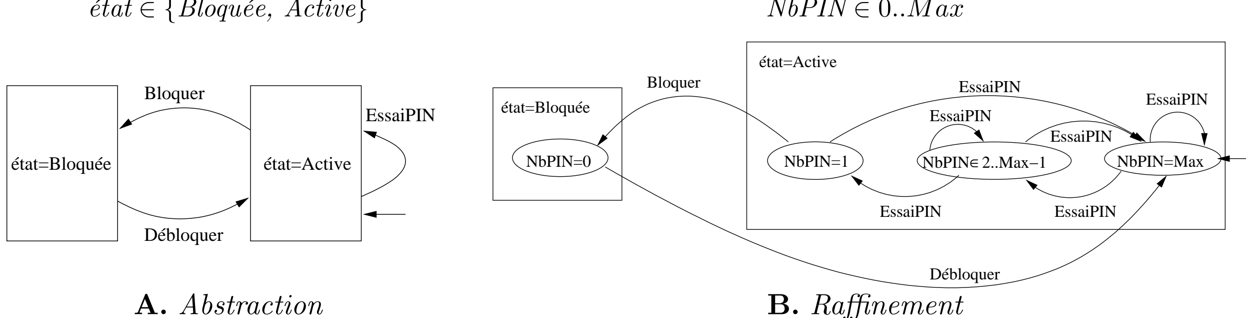
1.4.3 Visualiser le lien entre deux niveaux de description

Quelques travaux [BL07, LT05, IL06] (pour ne citer qu'eux) portent déjà sur l'aide à la compréhension de modèles B et à la validation de spécifications abstraites par rapport au cahier des charges. Cependant, aucun d'entre eux ne prend vraiment en compte le processus de raffinement.

En effet, un raffinement est une description à part entière d'un système à laquelle ont été rajoutées des informations permettant de faire le lien avec les données et les événements de son abstraction. Ce lien permet de tracer les données et leurs propriétés à travers le processus de raffinement et donc de casser la complexité des vérifications. C'est pourquoi, nous proposons de faire apparaître ce lien sur nos représentations. En utilisant des états hiérarchiques (figure 1.5), nous associons chaque niveau de hiérarchie aux données d'un niveau de raffinement. De plus, comme nous le verrons, cette approche hiérarchique conserve la structure générale du système de transitions abstrait. Il s'ensuit qu'il est possible de comparer les systèmes de transitions et donc de réutiliser les efforts de compréhension faits pour les niveaux les plus abstraits.

Par exemple, considérons deux descriptions d'une carte de paiement (figure 1.5) où l'on se focalise sur l'authentification par saisie du code PIN. Nous proposons d'observer 3 événements : *EssaiPIN* qui modélise la saisie d'un code PIN (qui peut être correct ou incorrect), *Bloquer* qui correspond à la saisie d'un code PIN faux entraînant un blocage de la carte (plus d'essai autorisé) et *Débloquer* qui permet de réactiver la carte en laissant à nouveau la possibilité de saisir un code PIN. Dans la première description (figure 1.5.A), nous ne modélisons qu'une unique variable *état*

$état \in \{Bloquée,\ Active\}$ $NbPIN \in 0..Max$



A. *Abstraction* **B.** *Raffinement*

FIG. 1.5 – Automate hiérarchique représentant les comportements d'une carte de paiement

qui peut avoir deux états (*Bloquée* ou *Active*). Seuls les événements *Bloquée* et *Débloquer* peuvent alors changer le système d'état. Dans la seconde description (figure 1.5.B), on ne s'intéresse plus à la variable *état*, mais à *NbPIN*, qui caractérise le nombre restant d'erreurs de saisie du code PIN avant blocage de la carte. L'événement *EssaiPIN* permet alors de remettre le compteur au maximum, si

le code est correct, ou sinon de décrémenter le compteur. Si le compteur était à 1 et que le code est incorrect, alors c'est l'événement *Bloquer* qui est déclenché. Cette seconde description est un raffinement de la première, car elle contient plus d'informations. La formule suivante permet de relier les variables *état* et *NbPIN* :

$$(NbPIN = 0 \Leftrightarrow état = Bloquée) \;\wedge\; (NbPIN > 0 \Leftrightarrow état = Active)$$

C'est ce lien, appelé invariant de liaison, que l'on veut expliciter sur nos automates en exploitant la hiérarchie des états. Cet invariant entre les données abstraites et raffinées peut ensuite être retrouvé dans le graphe. Il est ainsi possible de retrouver sur le système de transitions les associations suivantes, qui sont équivalentes à l'invariant de liaison donné plus haut :

$$(état{=}Active) \;\Leftrightarrow\; (NbPIN{=}\,1 \vee NbPIN{\in}\, 2..Max{-}1 \vee NbPIN{=}Max)$$
$$(état{=}Bloquée) \;\Leftrightarrow\; (NbPIN{=}0)$$

Enfin, certaines propriétés peuvent être préservées par raffinement. Par exemple, sur la description proposée en figure 1.5.A *il n'est pas possible de saisir un code PIN si la carte est bloquée.* Cette propriété est préservée dans le raffinement, comme on peut le voir sur la figure 1.5.B. Cette préservation est visible, car, grâce aux contraintes de la représentation hiérarchique proposée, la structure globale de la figure 1.5.A et conservée dans la figure 1.5.B.

1.4.4 Appliquer la méthode à la vérification de propriétés

Nous proposons trois techniques de vérification de propriétés qui se basent sur la représentation des comportements d'un modèle B. La première possibilité est une aide à la preuve d'invariant et consiste à vérifier que l'ensemble des états atteignables vérifient l'invariant.

Une seconde proposition se base sur le fait qu'une propriété de sûreté est préservée par raffinement. On s'intéresse alors à montrer que le modèle est un raffinement de la propriété. Pour ce faire, nous proposons de décrire la propriété et le modèle par des systèmes de transitions et d'établir que celui du modèle est une simulation de celui de la propriété (figure 1.6).

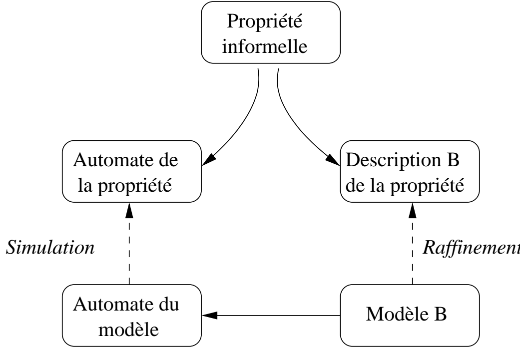


FIG. 1.6 – Deux approches de vérification basées sur le raffinement

Enfin, notre troisième proposition consiste à exprimer la propriété dans un langage logique suffisamment simple pour être décidable et suffisamment expressif pour pouvoir décrire des propriétés comportementales. Pour cela, nous introduisons des prédicats logiques permettant de décrire des

comportements élémentaires. Il suffit ensuite d'évaluer ces prédicats par lecture des comportements du modèle pour se ramener à une formule décidable et conclure si le modèle vérifie ou non la propriété.

1.5 Organisation de ce document

Dans le chapitre 2, nous présentons les différents travaux sur lesquels nous nous sommes basés pour identifier les besoins auxquels nous répondons dans cette thèse. Nous terminons ensuite cette partie avec le chapitre 3, dans lequel nous décrivons la méthode B, que nous utilisons comme support pour nos travaux.

Nous présentons ensuite nos contributions en deux temps. Dans le chapitre 4, nous décrivons notre formalisme de description graphique et nous proposons une méthode pour construire une représentation des comportements d'une spécification abstraite B événementiel. Dans le chapitre 5, nous proposons une extension de cette approche pour prendre en compte le processus de raffinement.

La partie 3, portant sur les outils et les applications, est composée de deux chapitres. Dans le chapitre 6, nous décrivons l'outil *GénéSyst* en précisant les choix d'implantation qui ont été faits pour améliorer ses performances. Dans le chapitre 7, nous proposons un exemple d'application de notre méthode dans le cadre de la vérification de propriétés de sécurité dans le domaine des cartes à puce.

Enfin, dans la dernière partie, nous concluons ces travaux en faisant le bilan de ce qui a été réalisé et en décrivant les pistes de recherche qu'il nous semble intéressant de développer.

À la fin de ce manuscrit, nous proposons trois annexes. La première est un complément sur la méthode B. La seconde est un recueil des démonstrations que nous n'avons pas voulu mettre dans le texte, afin d'alléger la lecture de ce document. Enfin, La troisième annexe est un glossaire des principaux acronymes et termes techniques utilisés.

De l'orienté données à l'orienté comportements : un état de l'art

2

Un langage de programmation est censé être une façon conventionnelle de donner des ordres à un ordinateur. Il n'est pas censé être obscur, bizarre et plein de pièges subtils (ça ce sont les attributs de la magie).

Dave Small, ST Magazine, novembre 1992

Sommaire

2.1 Langages informels de description de comportements . . . 24
2.2 Langages de description orientés comportements . . . 25
2.2.1 Systèmes de transitions concrets (**STC**) . . . 26
2.2.2 Systèmes de transitions symboliques (**STS**) . . . 27
2.2.3 Systèmes de transitions hiérarchiques . . . 30
2.3 D'une description orientée données vers une description comportementale . . . 33
2.3.1 Description explicite des comportements . . . 34
2.3.2 Extraction des comportements d'un modèle . . . 35
2.3.3 Représentation de lien de raffinement . . . 39
2.4 Vérification de propriétés comportementales . . . 39
2.5 Synthèse . . . 41

Dans ce chapitre, nous décrivons les principales approches dans le domaine de l'aide à la construction de modèles formels. En particulier, nous nous focalisons sur le passage entre cahier des charges et description formelle. L'approche classique consiste à utiliser un langage de description intermédiaire, qui soit suffisamment intuitif pour être utilisé et compris par le client et qui soit suffisamment bien défini pour permettre de faire le lien avec une spécification formelle abstraite. L'analyse du cahier des charges met en évidence un ensemble de propriétés de sécurité et d'exigences fonctionnelles qui sont ensuite décrites dans autant de modèles en langage intermédiaire. Chacun d'entre eux est une vue différente du système, et doit être présent dans le modèle formel final. Le passage par une phase de décomposition des problèmes permet de faciliter l'expression et la vérification des propriétés. De plus, la construction par raffinement du modèle permet d'intégrer ces différents aspects par étapes successives.

Il existe alors deux manières de faire ce lien entre description intermédiaire et modèle B : construire un squelette de modèle B à partir de la description intermédiaire ou valider un modèle B existant par rapport au cahier des charges. C'est cette seconde approche que nous avons adoptée. En effet, même si un modèle B est généré à partir du cahier des charges, celui-ci sera ensuite modifié et raffiné par les concepteurs. Le modèle produit doit donc être vérifié par rapport au cahier des charges. C'est pourquoi, nous pensons que l'approche par vérification est moins contraignante et permet de réduire le temps de développement.

Il semble intéressant d'exploiter un formalisme permettant de jouer sur la complémentarité des descriptions orientées données et orientées comportements. Ainsi, les deux aspects importants de la thématique de cette thèse sont l'extraction et la représentation de l'ensemble des comportements d'un modèle B. Notons toutefois que la méthode d'extraction des comportements d'un modèle peut être dépendante du formalisme de représentation choisi. C'est pourquoi, dans le tour d'horizon que nous effectuons dans cette section, nous commençons par introduire les différents formalismes de description de comportements avant de comparer les méthodes d'extraction. Dans un premier temps, nous proposons toutefois un court aparté sur des langages informels de description des comportements d'un modèle. Enfin, nous terminons ce chapitre en nous intéressant aux principales méthodes de validation d'un système qui se basent sur sa description comportementale.

Terminologie. *Afin d'alléger le discours, nous parlons, indifféremment de* ***figures****, de* ***vues graphiques****, de* ***représentations graphiques****, de* ***diagrammes****, de* ***systèmes de transitions*** *ou d'****automates****, pour désigner un diagramme représentant les comportements d'un modèle.*

*De la même manière, nous sommes amenés à parler de différents formalismes dans lesquels les modèles peuvent être décrits en termes d'événements (B événementiel), d'opérations (B), d'actions (TLA) ou de processus (UNITY). Afin d'homogénéiser la lecture, nous proposons de toujours parler d'****actions****, qui pourront être, suivant le contexte, préconditionnées ou gardées.*

2.1 Langages informels de description de comportements

Dans cette section, nous voulons mettre en évidence qu'il existe des approches informelles qui s'intéressent également à l'aide à la compréhension. Par exemple, la simulation est une approche permettant de montrer les fonctionnalités d'un système à des non experts. Par exemple, les animateurs Flash pour les outils *Brama*[1] [Ser06] et *ProB*[2] [BL07] permettent de simuler l'exécution d'une spécification sous la forme d'une animation de dessins. Chaque état de chaque composant physique modélisé est associé à un état de la description formelle et à une disposition graphique des dessins. Chaque événement du modèle est associé à une ou des animations graphiques. L'animation du modèle formel permet alors de générer une séquence de dessins animés. L'utilisation d'un animateur permet d'évaluer quelles gardes sont vérifiées dans un état donné et donc de choisir une action à activer. Cette approche permet une bonne compréhension des évolutions possibles d'un système,

[1] Développé par ClearSy
[2] Développé par J. Bendisposto et M. Leuschel de l'université de Düsseldorf.

mais nécessite un temps de développement assez long, puisqu'il faut dessiner tous les composants et prévoir les différentes transitions possibles du système.

Il existe également des approches de traduction en langue naturelle telles que l'outil *Composys* [Cle05, PPS06] de ClearSy. Cet outil, par exemple, contribue à la réalisation de la documentation d'un modèle et permet de lever totalement la barrière de la syntaxe informatique.

Dans ce type d'approches, le langage de sortie est suffisamment intuitif ou compréhensible pour que la validation puisse être effectuée par des personnes n'ayant aucune connaissance en informatique mais connaissant les besoins du modèle réalisé. Cependant, le résultat d'une telle approche ne peut pas être utilisé comme support à une réflexion mathématique, ce qui implique que cette validation ne peut pas être outillée ou automatisée.

2.2 Langages de description orientés comportements

Les comportements d'un système peuvent être décrits dans de nombreux langages. Par exemple, les diagrammes de séquences et d'activité [CSBSD01] d'UML [OMG01] permettent de représenter certains comportements d'un système [Voi04]. Cependant, les diagrammes de séquence ne décrivent qu'une classe d'exécutions et sont limités à des traces de longueur finies. Les diagrammes d'activité, quant à eux, sont une extension des diagrammes d'états-transitions, qui mettent l'accent sur la modularité d'un système et la synchronisation de ses processus. Ainsi, bien que ces langages permettent de décrire des aspects comportementaux, ils sont adaptés à des problématiques qui ne sont pas abordées ici (Unique classe d'exécution, traces finies, architecture du modèle, synchronisation des processus, etc.).

De la même manière, nous ne nous intéressons pas non plus aux langages algébriques tels que CSP [Hoa78], CCS [Mil80] ou le μ-calcul [Par74], car leur compréhension nécessite une trop grande expertise pour être utilisés dans une description faite pour un non informaticien.

Dans cette section, nous nous focalisons sur les différents types d'automates proposés dans la littérature et sur la manière de les associer aux comportements d'un modèle B. En particulier, nous distinguons les modèles ayant un nombre d'états finis, les modèles ayant une infinité d'états et les modèles paramétrés. Ces derniers sont définis modulo des paramètres et caractérisent donc une classe de systèmes. Une fois les paramètres fixés, le système peut avoir un nombre fini ou infini d'états, mais il peut exister un nombre infini de systèmes : un par valeur de paramètre. En B, par exemple, les constantes peuvent ne pas être valuées et sont alors considérées comme des paramètres. L'exemple d'authentification par code PIN, présenté à la fin du chapitre précédent, est un système B paramétré, où la constante Max n'est pas connue. Une fois celle-ci définie, le nombre d'états de ce système devient fini. Nous utilisons cet exemple pour illustrer les différents types d'automates décrits dans cette section.

Terminologie. ***Valuer*** *une donnée (variable ou constante) consiste à lui associer une valeur. Nous serons également amenés à parler d'une* ***configuration*** *ou d'un* ***état*** *d'un modèle pour désigner une valuation de l'ensemble de ses données.*

La sémantique d'un système de transitions est donnée par l'ensemble des chemins qu'il accepte.

Classiquement [GS97, BC00, Har87, LB03], on définit :

Définition 1 (Chemins d'un système de transitions) *Une séquence $e_1, \ldots, e_n$ de noms d'étiquettes est un **chemin** d'un système de transitions si et seulement si, il existe une séquence $q_1, \ldots, q_{n+1}$ d'états du système telle que :*
- *q_1 est l'état initial ;*
- *pour tout $i \in 1..n$, il existe une transition étiquetée par e_i et allant de l'état q_i à l'état q_{i+1}.*

De manière similaire, on définit l'ensemble des comportements d'un modèle par l'ensemble des séquences d'actions qu'il accepte.

Terminologie. *L'ensemble des séquences d'actions acceptées par un modèle sont appelées ses **traces d'exécution**. Un automate est une **représentation de l'ensemble des comportements d'un modèle** M si et seulement si chaque transition de l'automate est associée à l'exécution d'une action du modèle, et chaque état de l'automate est associé à une ou plusieurs configurations de M. Cette représentation est dite **exacte** si l'ensemble des traces d'exécution du modèle est égale à l'ensemble des chemins du système de transitions.*

Dans cette section, nous allons successivement décrire trois types de systèmes de transitions : les concrets, dont chaque état caractérise une valuation des variables du modèle, les symboliques, où un état peut être défini par une infinité de valuations des variables du modèle, et les hiérarchiques, où les états peuvent contenir des états et des transitions. Afin d'alléger le discours, nous nous permettrons d'appeler les systèmes de transitions des automates.

2.2.1 Systèmes de transitions concrets (STC)

Dans les systèmes de transitions concrets, chaque état correspond à une valuation possible des variables et chaque transition correspond à une occurrence possible de l'action dont le nom est dans l'étiquette. Ce formalisme permet de décrire exactement les comportements autorisés par le modèle associé, si celui-ci a un nombre fini d'états. Cependant, ce formalisme ne permet pas de décrire les comportements des systèmes paramétrés ou des systèmes ayant un nombre d'états infini.

Définition 2 (Système de transitions concret) *Un **STC** est une structure de Kripke étiquetée. Il peut être vu comme un quintuplet $(\mathbb{Q}, \mathsf{q}_{Init}, \mathbb{E}, \mathbb{R}, \mathcal{D}ef)$ tel que :*

- *$\mathbb{Q}$ est un ensemble de noms d'états,*
- *q_{Init} est un état initial ($\mathsf{q}_{Init} \in \mathbb{Q}$),*
- *$\mathbb{E}$ est un ensemble de noms d'actions,*
- *$\mathbb{R}$ est une relation de transition ($\mathbb{R} \subseteq \mathbb{Q} \times \mathbb{E} \times \mathbb{Q}$),*
- *$\mathcal{D}ef$ associe une unique valuation du système à chaque nom d'état ($\mathcal{D}ef \in \mathbb{Q} \rightarrow \mathbb{V}$).*

où $\mathbb{V}$ est l'ensemble des valuations du système.

La figure 2.1 est un exemple de représentation des comportements du modèle de carte de paiement vu dans le chapitre précédent. Pour construire ce système de transitions concret, nous avons dû valuer la constante *Max*, qui définit le nombre maximum d'erreurs successives de saisie du

code PIN autorisées avant blocage de la carte. Dans cet exemple, nous avons choisi Max=3 et l'état initial est (E_3). Il est caractérisé par une flèche ne venant d'aucun état. Enfin, cette description ne contient pas d'information sur la variable *état*, qui fait partie de la spécification abstraite.

Terminologie. *Dans cette thèse, nous choisissons de* ***nommer*** *chaque état d'un système de transitions avec un prédicat caractérisant la ou les valuations qu'il contient. De plus, pour clarifier les renvois entre le texte et le diagramme nous pouvons être amenés à compléter ce nom par un identifiant plus court ou en français. Les références à celui-ci sont alors faites, dans le texte, avec la syntaxe suivante :* (***Nom d'état***).

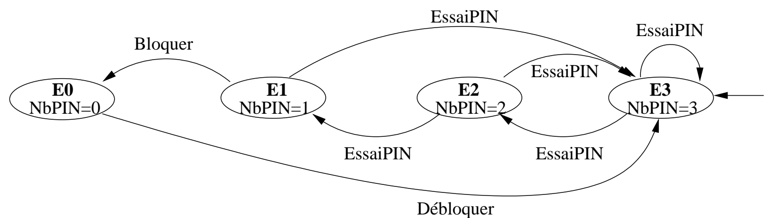


FIG. 2.1 – Exemple d'un système de transitions concret

Étant donné l'exactitude de telles descriptions, ce formalisme est utilisé dans différentes approches s'intéressant à la vérification de propriétés. Nous pouvons citer, par exemple, l'outil d'animation de spécifications *ProB* [LB03], qui utilise ce langage pour représenter les comportements d'un modèle B ayant un ensemble fini d'états. Il permet ensuite de vérifier la cohérence du modèle par model-checking ou par résolution de contraintes. Cet outil est décrit plus en détail en section 2.3.2.2.

Cependant, au vu de l'objectif d'aide à la compréhension, une représentation graphique est intéressante si le diagramme n'est pas trop grand ou bien s'il ne contient pas un entrelacement compliqué de transitions [LT05]. C'est pourquoi, nous proposons de nous tourner vers les systèmes de transitions symboliques.

2.2.2 Systèmes de transitions symboliques (STS)

Afin de diminuer le nombre d'états d'un automate, il est possible de les regrouper et de former un système de transitions symbolique, tel que les *abstract states machines* (ASM) [Gur85]. Les systèmes de transitions symboliques sont des automates dont chaque état est défini par un prédicat logique et caractérise un ensemble possiblement infini de valuations des variables. Il est ainsi possible de représenter les comportements de modèles paramétrés ou ayant un nombre infini d'états.

Il existe de nombreuses définitions de systèmes de transitions symboliques. Dans cette section nous présentons deux extrêmes : d'une part une définition simple, où les transitions ne sont étiquetées ni par une condition de franchissement, ni par une affectation, et d'autre part une définition plus fournie, où les transitions sont étiquetées par une condition de franchissement et une affectation.

Systèmes de transitions symboliques sans conditions de franchissement ou affectation

Formellement, il est possible de définir un **STS** comme un **STC** dont la fonction de définition des états $\mathcal{D}ef$ associe un prédicat, et non plus une unique valuation, à chaque nom d'état.

Définition 3 (Système de transitions symbolique simple) *Un **STS** ($\mathbb{Q}$, q_{Init}, $\mathbb{E}$, $\mathbb{R}$, $\mathcal{D}ef$) est un **STC** dont les états sont définis par des prédicats :*

- $\mathbb{Q}$ *est un ensemble de noms d'états,*
- q_{Init} *est un état initial* ($\mathsf{q}_{Init} \in \mathbb{Q}$)*,*
- $\mathbb{E}$ *est un ensemble de noms d'actions,*
- $\mathbb{R}$ *est une relation de transition* ($\mathbb{R} \subseteq \mathbb{Q} \times \mathbb{E} \times \mathbb{Q}$)*,*
- $\mathcal{D}ef$ *associe un prédicat de définition à chaque nom d'état* ($\mathcal{D}ef \in \mathbb{Q} \rightarrow \mathbb{P}$)*.*

où $\mathbb{P}$ *est l'ensemble des prédicats logiques.*

Ce type de formalisme est classiquement utilisé pour décrire les comportements d'un système décrit par ailleurs. Ils sont notamment utilisés dans des travaux visant à représenter l'ensemble des comportements d'une spécification abstraite dans le but de vérifier le respect de propriétés par model-checking. Nous pouvons notamment citer les travaux de S. Graf et H. Saïdi [GS97] et ceux de D. Bert et F. Cave [BC00], qui utilisent les systèmes de transitions symboliques pour décrire respectivement les comportements d'un système spécifié en UNITY et les comportements d'un système spécifié en B événementiel.

La figure 2.2 est un exemple de représentation symbolique des comportements du modèle de carte de paiement. Notons que dans ce formalisme, il est possible de ne pas valuer la constante *Max*.

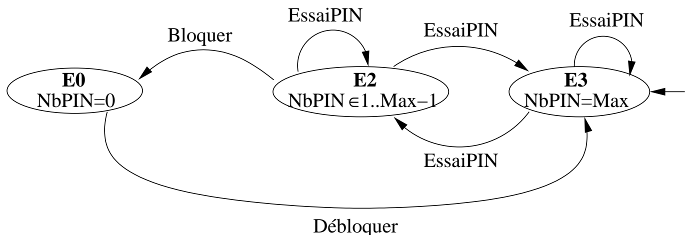


FIG. 2.2 – Exemple d'un système de transitions symbolique

Dans un système de transitions symbolique simple représentant l'ensemble des comportements d'un modèle M, l'ensemble des chemins peut être un sur-ensemble des traces d'exécution de M. Par exemple, sur la figure 2.2, les chemins commençant par la séquence *EssaiPIN, Bloquer,*... ne sont des traces d'exécution du modèle que si $Max = 2$. On distingue notamment les transitions toujours franchissables et partiellement franchissables.

Terminologie. *Une transition est dite **toujours franchissable** si elle est franchissable depuis toutes les valuations de son état d'origine. Sinon elle est dite **partiellement franchissable**. Sur la figure 2.2, la transition étiquetée par Bloquer est partiellement franchissable, tandis que celle étiquetée par Débloquer est toujours franchissable.*

Systèmes de transitions symboliques avec conditions de franchissement et affectations

Il est également possible de compléter la définition des systèmes de transitions en ajoutant, sur les transitions, des conditions de franchissement, ainsi que des affectations. Ces dernières correspondent à la modification de l'état du système qui est effectuée lors du franchissement de la transition.

Définition 4 (Système de transitions symbolique avec conditions de franchissement et affectations) *Un* ***STS*** $(\mathbb{V}, \mathbb{Q}, \mathsf{q}_{Init}, \mathbb{E}, \mathbb{R}, \mathcal{D}ef, Aff, G)$ *avec conditions de franchissement et affectations est tel que :*

- $(\mathbb{V}, \mathbb{Q}, \mathsf{q}_{Init}, \mathbb{E}, \mathbb{R}, \mathcal{D}ef)$ *est un* ***STS*** *simple,*
- *Aff décrit les affectations de chaque transitions* ($Aff \in \mathbb{R} \rightarrow \mathbb{F}(\mathbb{A})$)*,*
- *G décrit la garde de chaque transition* ($G \in \mathbb{R} \rightarrow \mathbb{P}$)*,*

où $\mathbb{P}$ *est l'ensemble des prédicats logiques et* $\mathbb{A}$ *l'ensemble des affectations possibles des variables du système (où* $\mathbb{F}(\mathbb{A})$ *est l'ensemble fini des parties de* $\mathbb{A}$*).*

Cette définition permet de donner plus d'informations sur les transitions partiellement franchissables, avec l'introduction des *conditions de franchissement*. La condition de franchissement d'une transition, aussi appelée garde, est un prédicat logique caractérisant depuis quelle sous-partie de son état de départ cette transition peut être franchie. Les transitions toujours franchissables ont une garde toujours vraie.

L'autre apport de cette définition est la description de la modification de l'état du système. Cependant, cette fonctionnalité est intéressante principalement si le système n'est pas associé à un modèle décrit par ailleurs ou si chaque action du modèle n'est décrit que par des affectations (sans structures de contrôle, telles que la conditionnelle, la boucle ou les choix non-déterministes). C'est pourquoi nous ne l'exploiterons pas dans notre approche où nous considérons que le système de transitions est extrait à partir d'un modèle B événementiel.

Par exemple, rien sur la figure 2.2 ne permet de déduire que la transition étiquetée par *Bloquer* et menant en E_0 depuis E_2 n'est franchissable que depuis la valeur $NbPIN = 1$. Cette information est en revanche disponible sur la figure 2.3, dans la garde la transition. Cependant, les conditions de franchissement ne suffisent pas pour avoir une notion plus précise de l'ensemble des chemins. En effet, comme on ne connaît pas la sous-partie de l'état que l'on atteint par une transition donnée, alors on ne sait pas, dans la séquence de deux transitions, depuis quelle sous-partie de l'état de départ le franchissement de la première transition permet de satisfaire la garde de la deuxième transition. Par exemple, par lecture de la figure 2.3, on ne sait pas que les chemins commençant par la séquence *EssaiPIN, Bloquer,...* ne sont des traces d'exécution du modèle que si $Max = 2$. Les figures 2.2 et 2.3 ont donc le même ensemble de chemins. La description des affectations permet, quant à elle, d'affiner partiellement cet ensemble de chemins.

Une seconde sémantique [GBJ06], plus rarement utilisée, peut alors être attachée à un système de transitions symbolique associé à un modèle. Dans cette sémantique, on explicite le lien entre l'automate et le modèle. En fait cette sémantique associe les chemins du système de transitions aux traces d'exécution du modèle. De cette manière, un système de transitions décrit exactement

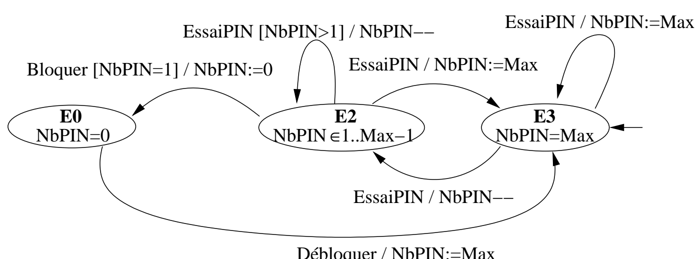


FIG. 2.3 – Ajout de gardes sur les transitions partiellement franchissables

l'ensemble des traces d'exécution du modèle.

Définition 5 (Chemins associés à une trace d'exécution) *Soit S un modèle et T un système de transitions symbolique avec conditions de franchissement et affectations. Une séquence $e_1, \ldots, e_n$ de noms d'actions de S est un* **chemin** *de T si et seulement si, il existe une séquence $q_1, \ldots, q_{n+1}$ d'états de T et une séquence $v_1, \ldots, v_{n+1}$ de valuations de S telles que :*
- *chaque v_i vérifie le prédicat de définition de l'état q_i ;*
- *q_1 est l'état initial ;*
- *pour tout $i \in 1..n$:*
  - *il existe une transition étiquetée par la garde g_i, l'action e_i et l'affectation a_i et allant de l'état q_i à l'état q_{i+1} ;*
  - *la valuation v_i vérifie la garde g_i ;*
  - *l'action e_i peut atteindre la valuation v_{i+1} depuis la valuation v_i.*
  - *l'affectation a_i exécutée depuis la valuation v_i peut atteindre la valuation v_{i+1}.*

Cette seconde sémantique permet de décrire l'ensemble exact des comportements d'un modèle, même s'il est paramétré ou s'il a une infinité d'états. Cependant, elle nécessite de connaître la spécification des actions. Par exemple, considérons la définition suivante de l'action *EssaiPIN* :

$$EssaiPIN \;\hat{=}\; NbPIN > 1 \Longrightarrow ((NbPIN := NbPIN - 1) \;[\!]\; (NbPIN := Max))$$

Cette action n'est déclenchable que si $NbPIN > 1$. Elle peut alors, de manière non déterministe, soit décrémenter le compteur *NbPIN*, soit l'affecter avec la valeur *Max*. À partir de cette spécification et en utilisant la seconde sémantique des chemins, nous pouvons conclure que les séquences d'actions commençant par *EssaiPIN, Bloquer,...* ne sont des chemins du système de transitions, et donc des traces d'exécution du modèle, que si $Max = 2$.

Dans les chapitres suivants, nous utilisons cette définition pour définir formellement la sémantique des systèmes de transitions que nous introduisons. Cependant, comme dit précédemment, nous ne considérerons pas la notion d'affectations associées à des transitions.

2.2.3 Systèmes de transitions hiérarchiques

Pour aider à mieux structurer la représentation d'un système de transitions symbolique, D. Harel a introduit le concept de hiérarchie, définissant ainsi les StateCharts [Har87]. Le principe est de regrouper des états à l'intérieur d'*états hiérarchiques*. On définit alors l'ensemble des comportements

décrit par un StateChart comme celui du système de transitions symbolique composé de ses états-feuille.

Terminologie. *Les* ***états hiérarchiques****, aussi appelés* ***états composés*** *ou* ***clusters****, sont des états contenant des états et des transitions. À l'inverse, des* ***états-feuille*** *sont des états non hiérarchiques. Enfin, un état hiérarchique est le* ***super-état*** *de chacun de ses* ***sous-états****.*

Les StateCharts intègrent de nombreuses autres caractéristiques, telles que la parallélisation des sous-systèmes, les états d'historique ou les affectations dans les transitions, que nous ne décrirons pas ici, car nous ne les utilisons pas dans la suite de cette thèse. Cependant, il serait intéressant d'étudier leur prise en compte dans l'approche présentée ici.

Par exemple, la figure 2.4 est une description du modèle de carte de paiement où deux niveaux de description sont pris en compte. Le premier, la spécification abstraite, porte sur la variable *état*, qui peut être *Bloquée* ou *Active*. Le second, quant à lui, est un raffinement qui porte sur la variable *NbPIN* et qui introduit la constante *Max*. Le lien entre les données abstraites et les données raffinées est caractérisé par l'inclusion des états. L'état **Active** est défini comme initial par une flèche qui l'atteint depuis un point noir (méta-état caractérisant l'ensemble des états sources). Les états hiérarchiques eux aussi contiennent des sous-états initiaux, indiqués de la même manière. Un sous-état initial matérialise le point d'entrée usuel de l'état composé. Syntaxiquement, toute transition menant sur le bord d'un état composé mène, en fait, à son sous-état initial.

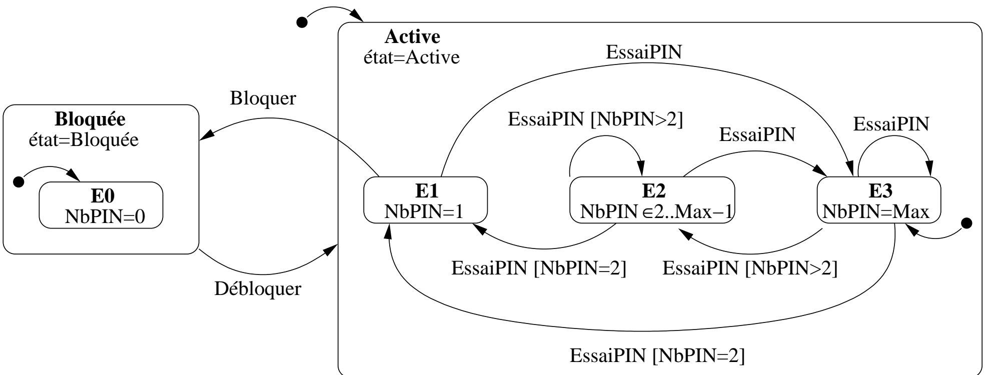
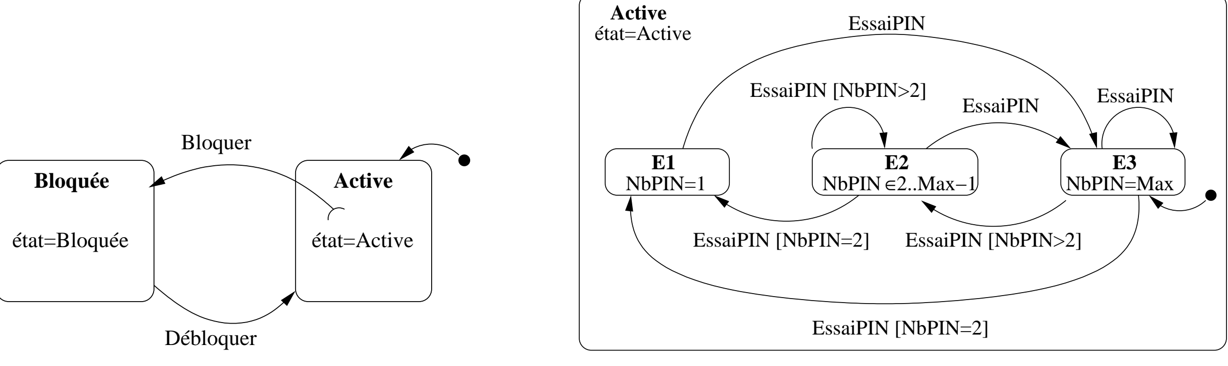


Fig. 2.4 – Exemple de StateCharts

D. Harel présente la hiérarchie comme une méthode d'abstraction permettant de diminuer la complexité apparente d'un système de transitions. En effet, un état hiérarchique est considéré comme un StateCharts à part entière. On peut alors masquer ou afficher les StateCharts internes selon le niveau de granularité recherché. Par exemple, les figures 2.5.A et 2.5.B sont deux vues de la figure 2.4 représentant respectivement les interactions globales entre les super-états (StateCharts internes masqués) et les comportements internes à un état composé (interactions globales masquées). Si une transition n'est déclenchable que depuis une partie des sous-états d'un état hiérarchique dont le StateCharts interne est masqué, alors la transition prend son origine au milieu de l'état et est *bouchonnée* par un arc de cercle (Transition *Bloquer* sur la figure 2.5.A). Sinon, elle est factorisée sur le super-état (Transition *Débloquer* sur la figure 2.5.A).



A. *Interactions globales* **B.** *Zoom sur l'état état=Active*

FIG. 2.5 – Choix de la granularité de représentation d'un StateCharts

De plus, comme il est possible de choisir le niveau de granularité d'une représentation, la hiérarchie permet de comprendre un système par parties. Cependant, cela nécessite de réduire autant que possible le nombre de transitions externes. Par exemple, les transitions externes menant au sous-état initial peuvent être simplifiées et n'atteindre que la limite extérieure du cluster, sans que cela change sa sémantique. De même, une transition partant d'un état composé est franchissable depuis chacun des sous-états, permettant ainsi de factoriser certaines transitions.

Terminologie. *Si E est un état composé, alors une* ***transition externe*** *est une transition reliant un sous-état de E à un état extérieur à E.*

Les diagrammes d'états-transitions d'UML [OMG01, Boo94] sont une extension des StateCharts adaptée à la problématique de la représentation de programmes orientés objet. Du point de vue de la problématique de cette thèse, la principale différence entre ces formalismes est l'introduction du concept de sous-état final. Ceux-ci permettent de caractériser l'état de fin usuel d'un processus (matérialisé par un sous-StateChart). Syntaxiquement, toute transition partant du bord d'un état hiérarchique part en fait de son sous-état final. La figure 2.6 est une représentation UML de la figure 2.4. Les sous-états (E_0) et (E_1) sont les sous-états finaux. Ils sont caractérisés par une flèche qui en part et qui atteint un point noir (méta-état caractérisant l'ensemble des états destination).

Dans les StateCharts et les diagrammes d'états-transitions, les états sont caractérisés par leur nom et non pas par une définition formelle. C'est-à-dire qu'ils ne sont pas définis par un prédicat logique et n'ont pas de domaine de définition. Les transitions, quant à elles, sont décorées par le nom de l'événement permettant de les franchir, d'une éventuelle condition de franchissement pouvant cependant être décrite en langue naturelle et d'une ou plusieurs affectations.

Par contre, le concept de hiérarchie, qui permet de matérialiser un lien d'inclusion d'états et de caractériser la notion de profondeur des états, semble parfaitement adaptée à la représentation d'un lien de raffinement entre des données abstraites et des données raffinées. Intuitivement, nous voudrions dire que chaque niveau de hiérarchie correspond à un niveau de raffinement et que chaque état hiérarchique est composé des états qui le caractérise dans le niveau de raffinement suivant.

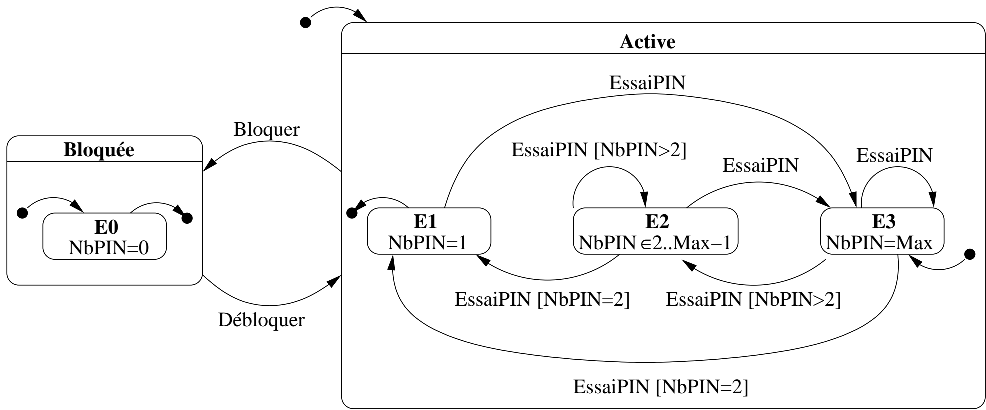


FIG. 2.6 – Diagramme d'état transitions utilisant des sous-états finaux

Dans la section suivante, nous ne nous intéressons plus aux formalismes de représentation des comportements d'un modèle, mais aux méthodes permettant d'extraire ces comportements à partir d'une spécification formelle.

2.3 D'une description orientée données vers une description comportementale

Nous voulons faire ressortir les aspects dynamiques d'un modèle décrit dans un langage orienté données, tel que B, de telle sorte que l'utilisateur puisse intuiter ce qui arrive dans le système. Globalement, ce lien entre un modèle B est ses comportements peut être fait de deux manières :

- Dériver un modèle B à partir d'une description des comportements voulus ;
- Extraire les comportements d'un modèle B existant.

Dans cette section, nous décrivons successivement ces deux approches. Nous détaillons particulièrement la seconde, qui s'intègre dans le processus de développement que nous proposons. Pour diminuer la complexité de cette extraction, nous pouvons être amenés à construire une approximation des comportements du modèle. Suivant les propriétés que l'on veut vérifier, on s'intéresse alors à des sur-approximations ou à des sous-approximations.

Terminologie. ***Sur-approximer*** *les comportements d'un modèle M consiste à caractériser un ensemble de séquences d'actions contenant au moins les traces d'exécution de M. À l'inverse,* ***sous-approximer*** *ses comportements consiste à caractériser un ensemble de séquences d'actions inclus dans les traces d'exécution de M.*

Sur-approximer les comportements d'un modèle garantit la complétude de l'ensemble des comportements décrits par rapport à ceux du modèle. Cela permet de vérifier la non-existence d'une trace ou la non-atteignabilité d'un état. À l'inverse, sous-approximer les comportements d'un modèle garantit la correction de l'ensemble des comportements décrits par rapport à ceux du modèle. Il

s'ensuit qu'une trace de cet ensemble est nécessairement une exécution possible du modèle. On peut alors s'intéresser à des problèmes d'atteignabilité d'un état ou d'existence d'une trace finie.

2.3.1 Description explicite des comportements

Différents travaux s'intéressent au développement de modèles B dont le contrôle est décrit dans un autre formalisme, tel que UML, CSP ou EB[3]. Ce faisant, il est possible soit de générer un système B événementiel intégrant ce contrôle, soit de contrôler les exécutions possibles d'une machine B.

E. Meyer et J. Souquières [MS99] présentent une méthode pour traduire un modèle semi-formel, décrit par des diagrammes de classes et d'états transitions OMT (méthode intégrée dans UML), en une spécification B. La traduction est effectuée par application de règles de réécriture et permet d'utiliser les modèles B générés comme point de départ pour un développement formel. Ces travaux ont été ensuite étendus par Ninh Thuan Truong et Jeanine Souquières [TS06]. Ils proposent d'utiliser cette méthode de réécriture, non plus pour l'aide au développement de modèles B, mais pour la vérification de la cohérence de modèles UML pouvant contenir des contraintes OCL.

F. Gervais, M. Frappier et R. Laleau [GFL07] proposent, pour leur part, d'utiliser le langage EB[3] à la place des diagrammes UML, pour générer des composants B. EB[3] est un langage dédié à la description de systèmes d'informations qui se base, d'une part, sur une description des traces d'exécution autorisées, et d'autre part, sur une description de la structure de données (Diagrammes ER[3] -*Entity-Relationship*). L'originalité de cette approche est que le processus de raffinement B est exploité pour prendre en compte la structure de données décrite dans la spécification EB[3], ainsi que des propriétés invariantes précisées par le développeur. Cela permet alors de séparer les vérifications liées aux aspects contrôle et données.

Enfin, différents travaux ont porté sur l'intégration du langage B et du langage CSP [Hoa78] (Communicating Sequential Processes). En effet, la notation CSP permet de définir explicitement l'ordre dans lequel des actions doivent s'exécuter. Par exemple, il est possible de définir l'ensemble des traces d'exécution du modèle de carte de paiement, par le processus CARTE décrit par induction comme suit :

$$CARTE \;=\; (EssaiPIN \rightarrow CARTE) \,\Box\, (Bloquer \rightarrow Débloquer \rightarrow CARTE)$$

où *Débloquer*, *Bloquer* et *EssaiPIN* sont des actions telles qu'une occurrence de *Bloquer* ne soit jamais suivie par *EssaiPIN*. Notons qu'en CSP, la notation $\Box$ caractérise un choix non-déterministe et $\rightarrow$ caractérise un enchaînement gardé.

L'approche CSP2B [But00] proposée par M. Butler consiste à décrire les comportements du modèle sous la forme d'une spécification CSP. Celle-ci est ensuite dérivée en un squelette de système B événementiel dans lequel ce contrôle est encodé par des variables et des gardes. Cette spécification peut être raffinée pour préciser les algorithmes et rendre les données plus concrètes. Les approches CSP——B [ST05] et CSP+B [BL05], quant à elles, proposent de définir un modèle comme la composition d'une machine B et d'un modèle CSP. Ce dernier permet de définir le contrôle du modèle

[3] sous ensembles des diagrammes de classe UML.

global, tandis que la machine B permet de définir sa structure de données et ses actions. Les deux approches diffèrent principalement par leur manière de vérifier la cohérence du modèle global.

Ces différentes approches s'intègrent donc dans un processus de développement descendant où le concepteur décrit explicitement les comportements attendus du modèle. Pour notre part, nous nous intéressons à l'approche inverse, basée sur l'extraction des comportements d'un modèle B existant. Il est alors possible de valider le modèle vis-à-vis du cahier des charges.

2.3.2 Extraction des comportements d'un modèle

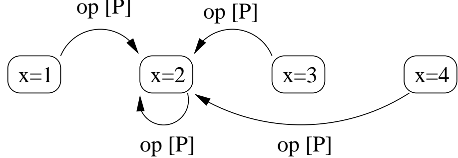
Nous nous intéressons maintenant aux principales méthodes d'extraction des comportements d'un modèle : par règles de réécriture, par animation ou par la preuve.

2.3.2.1 Extraction par règles de réécriture

La construction des comportements possibles d'un modèle formel peut être effectuée de manière syntaxique, par le biais de règles de réécriture. Le principal intérêt d'une telle approche est son coût d'exécution très faible. Cependant, l'ensemble de comportements ainsi construit peut contenir de nombreuses transitions non franchissables, puisque les conditions de franchissement sont calculées de manière syntaxique.

J-C. Voisinet et B. Tatibouet [VT03, Voi04] proposent une approche pour générer un diagramme d'états-transitions représentant l'ensemble des comportements d'un modèle B par applications successives de règles de réécriture. Pour ce faire, ils se restreignent à une sous-partie du langage B, dans laquelle les substitutions primitives sont **select**, **pre**, **choice**, **any** et **:=**, et dans lequel il n'est pas possible d'imbriquer les substitutions introduisant une condition (**select**, **pre** et **any**).

Leur méthode consiste à construire les comportements d'une action par rapport à une variable, appelée variable observée. Chaque état contient une unique valuation de la variable observée. Comme il n'y a pas de contrainte sur les autres variables, les états sont symboliques. De plus, la variable observée ne doit être affectée qu'avec des constantes explicites et non des expressions ou des variables. Par exemple, il est possible d'écrire $x := 4$ mais pas $x := 2 + 1$ ou $x := y$. L'état d'arrivée d'une action peut ainsi être déterminé de manière syntaxique, indépendemment de l'état initial de la variable. Par exemple, les comportements d'une action *op* décrite par **select** P **then** x **:=** 2 **end** pourront être construits sur les états $\{x=1, x=2, x=3, x=4\}$ comme suit :



La condition de franchissement P est également déterminée syntaxiquement à partir de l'action *op*, puisque c'est le prédicat de garde de la substitution **select**. Si plusieurs variables sont observées, alors chacune d'elles donne lieu à la construction d'un système de transitions. Les systèmes obtenus sont ensuite composés par produit cartésien pour construire une représentation dont les états explicitent la valuation de plusieurs variables.

Cette approche syntaxique amène à construire un grand nombre de transitions infranchissables (dont les conditions sont trivialement fausses depuis leur état de départ). Par exemple, dans le cas de l'opération *op* décrite précédemment, si la condition P de la substitution **select** est $x = 3$, alors seule l'une des quatre transitions construites est franchissable : celle allant de $x = 3$ à $x = 2$. C'est pourquoi, les auteurs intègrent un mécanisme simple de résolution de contraintes basé sur l'évaluation de certains connecteurs logiques ($=$, $\in$, $\wedge$, $\vee$ et $\neg$).

Enfin, l'ensemble des comportements calculés selon cette méthode est complet et la construction est moins coûteuse que les techniques d'animation ou de preuve. Cependant, les restrictions sur le langage d'entrée sont très fortes :

- Les variables observées doivent être entières ou typées par un ensemble énuméré ;
- L'espace d'états est nécessairement une énumération de toutes les valeurs des variables observées ;
- Les substitutions ne peuvent contenir que des affectations (pas d'imbrication des substitutions) ;
- Les variables observées ne doivent pas être affectées par des variables ou des expressions.

Cette approche est donc particulièrement intéressante dans le cas de modèles dont le contrôle est explicitement décrit à l'aide de variables d'états.

2.3.2.2 Extraction par animation

Animer un modèle consiste à déterminer, pour un état de départ donné, les états atteints par l'exécution d'une suite d'opérations. Animer un modèle permet donc de construire une sous-approximation de l'ensemble de ses comportements. Si cet ensemble est fini et suffisamment petit, alors l'animation permet de construire l'ensemble des comportements du modèle. Ce processus peut être dirigé manuellement, par un utilisateur, ou de manière automatique.

L'une des difficultés liées à l'animation est la prise en compte du non-déterminisme [LAB+01]. En effet, l'animation de spécifications non-déterministes ne peut pas toujours être effectuée pas à pas, comme c'est le cas usuellement pour les programmes. Par exemple, pour caractériser les valeurs possibles d'une variable définie par un prédicat logique P, il est nécessaire d'évaluer chacune des contraintes décrites par P. Il est alors nécessaire de faire appel à un solveur de contraintes.

Classiquement il existe deux grandes familles de techniques d'animation : par énumération [LB03] ou par résolution de contraintes [BDL05].

Animation par énumération des valeurs des variables

L'énumération est une technique d'animation où chaque état du système de transition construit est caractérisé par une valuation de l'ensemble des variables. Par exemple, l'outil *ProB* [LB03] permet d'animer une spécification B par énumération et fournit en résultat un système de transitions concret décrivant un ensemble de traces d'exécution du modèle. Comme les états sont concrets, ils peuvent être trivialement comparés, et les boucles sont donc détectées. Cette approche permet de construire un système de transitions complet, si tous les états successeurs ont été traités. Sinon, la génération s'arrête lorsqu'une borne, fixée par l'utilisateur, est atteinte.

Cet outil permet de valider la cohérence d'un modèle. Il intègre pour cela un model-checker et un solveur de contraintes. Le premier permet de vérifier que les états atteints respectent l'invariant du modèle. Si ce n'est pas le cas, alors la trace d'exécution menant à cet état est mise en évidence comme étant un contre-exemple. Le solveur de contraintes, quant à lui, permet d'établir si l'exécution d'une opération depuis un état vérifiant l'invariant permet de violer cet invariant.

Enfin, les systèmes de transitions produits peuvent être trop grands pour permettre d'aider à la compréhension du modèle. C'est pourquoi, *ProB* intègre deux algorithmes [LT05] de regroupement des états qui permettent de représenter l'ensemble des traces d'exécution sous la forme d'un système de transitions symbolique. La première méthode proposée consiste à abstraire les opérations de leurs paramètres, puis à rendre déterministe le système obtenu, en regroupant les états qui sont atteints par des transitions ayant la même origine et la même étiquette. La seconde méthode consiste à regrouper les états ayant la même signature, la signature d'un état étant définie par l'ensemble des noms des événements qui y sont déclenchables.

Animation par résolution de contraintes

Il est également possible d'animer une spécification de manière symbolique. Chaque état est défini par un prédicat et animer une opération consiste à caractériser, par résolution de contraintes, l'ensemble des valeurs qu'elle peut atteindre depuis un état donné. Cette méthode a l'avantage de pouvoir construire les comportements de modèles ayant une infinité d'états.

Par exemple, les outils *BZ-TT* [ABC+02] et *JML-TT* [BDLU05b, BDLU05a] utilisent un solveur de contraintes [BLP02] pour simuler l'exécution d'une opération depuis un état symbolique. Les états du système sont alors calculés *en avant* et caractérisent l'ensemble des valeurs atteignables, depuis l'état initial, par une séquence finie d'opérations. Partant d'un modèle ayant un nombre fini d'états, ces outils permettent donc d'exhiber les traces d'exécutions existantes. Notons que, pour l'instant, il n'y a pas de détection des valuations déjà atteintes et donc pas de détection de boucles dans les traces d'exécution.

Cependant, ces outils ont pour objectif la génération de tests et non pas la représentation des comportements calculés. C'est pourquoi, il est délicat d'exploiter cette approche pour l'aide à la compréhension.

2.3.2.3 Extraction par la preuve

Étant donné un espace d'états symboliques $\mathbb{Q} = \{E_1, \ldots, E_n\}$, construire une représentation des comportements d'un modèle pour l'ensemble $\mathbb{Q}$ consiste à caractériser l'ensemble des transitions reliant ces états. La résolution d'obligations de preuve permet de déterminer, pour chaque couple d'états (E_i, E_j), si une action permet d'aller à E_j depuis E_i.

Dans cette approche, l'une des motivations est de conserver toute l'expressivité des langages formels (ensembles infinis, non-déterminisme, etc.), ce qui implique des problèmes d'indécidabilité pouvant introduire des approximations dans la représentation des comportements du modèle.

C'est ce problème que pointe particulièrement le travail de S. Graf et H. Saïdi [GS97], s'inspirant de la méthode d'interprétation abstraite par partitionnement, présentée dans la thèse de

D. Dams [Dam96]. Partant d'une spécification en UNITY [CM88] et d'un ensemble d'états décrits par des prédicats, ils proposent une méthode systématique pour abstraire le système. En exploitant les propriétés des connexions de Galois [CC77], il leur suffit alors de montrer qu'il n'existe pas de transition entre deux états abstraits pour en conclure qu'il n'existe pas non plus de transition entre les états correspondants du système et inversement, de montrer qu'il existe toujours une transition entre deux états abstraits pour en conclure qu'il existe toujours une transition entre les états correspondants du système. Si une transition n'est ni jamais, ni toujours franchissable, alors elle est considérée comme partiellement franchissable. Cette partialité peut avoir deux significations : soit seulement une partie des valuations de l'état initial peut franchir cette transition, soit il y a eu un défaut de preuve.

Terminologie. *On appelle **défaut de preuve** le cas où un démonstrateur termine en n'ayant pas établi la véracité d'une formule vraie. On parle aussi de **faux négatif**.*

Syntaxiquement, les auteurs ne font pas de distinction entre les transitions toujours et partiellement franchissables. Toutefois, ils proposent des critères de choix de l'espace d'états permettant de limiter l'introduction de transitions partiellement franchissables. Tout d'abord, comme chaque action n'est décrite que par une garde et une ou plusieurs affectations, sans autre structure de contrôle[4], il est possible de choisir un ensemble d'états tel que le déclenchement des événements ne soit pas conditionné. Par exemple, il suffit de définir l'ensemble des gardes des actions comme l'ensemble des états du système. Ensuite, les états doivent être choisis pour réduire le non déterminisme du système de transitions construit. Dans la mesure du possible, chaque action ne doit donc mener que dans un seul état depuis un état donné.

Cette approche est notamment étendue par D. Cansell, D. Méry et S. Merz [CMM00a, CMM00b] qui proposent une méthode similaire de construction d'un diagramme de prédicats-actions [Lam95] représentant les comportements d'un modèle décrit en TLA [Lam94b]. Leur objectif est alors de permettre la vérification de propriétés de sûreté et de vivacité. Pour ce faire, ils ajoutent des décorations d'équité et d'ordre bien-fondé sur les transitions, ce qui leur permet de se ramener, par réécriture, à de la vérification par model-checking. De plus, ils mettent en évidence les transitions partiellement franchissables, qu'ils appellent *maybe transitions*. En effet, le sous-ensemble du langage TLA pris en compte est plus riche que le sous-ensemble de UNITY considéré dans les travaux décrits précédemment. Il peut alors être difficile de choisir un espace d'états tel que toutes les transitions soient toujours déclenchables. Il suffit alors de construire l'ensemble des transitions toujours franchissables et celui des transitions jamais franchissables dans le modèle abstrait pour en déduire les transitions partiellement franchissables.

Enfin, la même année et dans la même conférence, Didier Bert et Francis Cave [BC00] ont proposé une comparaison de différentes méthodes de construction des comportements d'un système B événementiel. Ils proposent notamment de ne pas nécessairement passer par un processus d'interprétation abstraite. Ils introduisent alors différentes méthodes de choix de l'espace d'états telles que l'énumération des valeurs des variables ou la décomposition de l'invariant du modèle B en une disjonction de prédicats. Ces différentes approches sont détaillées en section 4.4.2.

[4] En particulier pas de structure conditionnée

2.3.3 Représentation de lien de raffinement

Peu de travaux se sont intéressés à la prise en compte du processus de raffinement dans le contexte de la représentation des comportements de modèles formels. En effet, l'approche classique [CMM01, Voi04] consiste à considérer un raffinement comme une spécification à part entière, et à construire l'ensemble de ses comportements sans faire le lien avec ceux de son abstraction, ni faire le lien entre les données abstraites et les données raffinées.

D. Cansell, D. Méry et S. Merz [CMM01] se sont intéressés à simplifier la vérification de propriétés temporelles par model-checking en définissant une notion de raffinement sur les diagrammes de prédicats-actions. Ils distinguent alors le raffinement structurel et le raffinement de données, permettant respectivement d'affiner le contrôle ou de changer la représentation des données, par le biais d'un invariant de collage. Cependant, cet invariant de collage n'est pas représenté. C'est-à-dire que l'ensemble des comportements de la spécification abstraite et ceux du raffinement sont représentés par 2 diagrammes séparés et que les auteurs ne font pas de lien entre les états des ces diagrammes. Ainsi, un lecteur voulant faire le lien entre ces deux vues devra retrouver par lui-même le lien de raffinement entre ces deux vues.

De la même manière, J-C. Voisinet et B. Tatibouet [VT03, Voi04] proposent une méthode automatique de génération d'un diagramme d'états-transitions à partir d'un raffinement B. Pour cela, ils considèrent le composant de raffinement comme une spécification à part entière et exécutent leur algorithme de génération par application des règles de réécriture présentées précédemment (Section 2.3.2.1). Cette approche ne permet pas non plus de mettre en évidence les liens entre les données abstraites et les données raffinées, ni entre les comportements de l'abstraction et ceux du raffinement.

Dans cette thèse, nous étendons la démarche d'aide à la compréhension de modèles formels par explicitation des comportements, à des développements construits par raffinement. En effet, comme nous l'avons introduit dans le premier chapitre, le raffinement est un outil de modélisation important pour maîtriser la complexité des systèmes à modéliser, tout en assurant une certaine traçabilité. Pour ce faire, nous adoptons une approche basée sur la preuve et la vérification de propriétés.

2.4 Vérification de propriétés comportementales

Nous proposons d'appliquer les résultats de cette thèse à la vérification de propriétés de sûreté. Parmi celles-ci, on distingue les propriétés statiques (propriétés invariantes sur les états) et les propriétés comportementales [Sha93] (aussi appelées dynamiques). Ce sont ces dernières qui nous intéressent particulièrement, car la méthode B ne permet pas de les exprimer directement, bien qu'elles soient importantes pour définir la notion de raffinement.

Lors du développement d'un système B événementiel, le spécifieur est amené à valider ses choix en confrontant son modèle au cahier des charges initial. Cette validation peut être effectuée par construction ou *a posteriori*. Dans le premier cas, il est possible de construire un modèle M à partir d'une propriété ϕ, en utilisant des règles de réécriture garantissant la conformité de M par rapport

à ϕ. Si l'on considère qu'une propriété comportementale est décrite par un automate, alors cette approche correspond, par exemple, aux approches présentées en section 2.3.1, où le modèle abstrait est généré à partir de ses comportements attendus.

Il est également possible de valider un modèle *a posteriori*. L'utilisation d'un outil permettant de construire une vue comportementale du modèle B permet alors de mettre en évidence les aspects dynamiques du modèle. Il existe ensuite différentes méthodes permettant d'automatiser la confrontation de cette représentation avec les propriétés voulues, ou au moins de lui fournir un cadre de réflexion formel. Pour valider un modèle M par rapport à une propriété comportementale ϕ, il existe globalement deux manières de procéder :

1. par model-checking,
2. par preuve.

La première approche se base sur une description comportementale de M et de ϕ. Vérifier la conformité de M par rapport à ϕ revient alors à comparer l'ensemble des traces d'exécution du système à celui des traces d'exécution autorisées par la propriété. Cependant, cette approche nécessite de construire une description comportementale concrète du modèle et de la propriété. B. Parreaux a étudié dans sa thèse [Par00] différentes techniques pour vérifier, par model-checking, qu'un modèle B événementiel respecte une propriété dynamique, exprimée en logique PLTL[5]. L'algorithme classique consiste à calculer l'automate complémentaire à celui définissant les traces autorisées par la propriété, puis à établir que son intersection avec les traces d'exécution du système est vide. Pour retarder l'arrivée des problèmes liés à l'explosion combinatoire, l'auteur explore un certain nombre de techniques telles que l'interprétation abstraite, le model-checking symbolique, les BDD[6], la compression mémoire, les méthodes à la volée, etc. Dans la continuité de ces travaux, P.A. Masson, H. Mountassir et J. Julliand [MMJ00] proposent de diminuer la complexité de la vérification en décomposant la spécification en modules. Après avoir introduit leur notion de module, les auteurs définissent la classe BA_2 des formules P qui peuvent être vérifiées de manière modulaire, c'est-à-dire telles que « *si P est vrai dans tous les modules alors P est vérifiée par le modèle entier* ». En particulier, les auteurs s'intéressent à vérifier pour trois schémas de propriétés temporelles qu'ils peuvent être vérifiés de manière modulaire.

Enfin, la seconde approche consiste à ramener la propriété comportementale ϕ à un ensemble de formules de logique du premier ordre, que l'on peut ensuite vérifier à l'aide de prouveurs. Cette méthode a été initiée, dans le cadre de la méthode B, par J-R. Abrial et L. Mussat [AM98] qui étendent le langage B événementiel avec des contraintes dynamiques, exprimées dans une logique temporelle. Les propriétés ainsi exprimables sont soit des propriétés basées sur l'évolution des variables par une unique exécution d'un événement (opérateur next de la logique temporelle), soit des propriétés qui caractérisent l'atteignabilité d'un état Q depuis un état P. Dans ce dernier cas, les obligations de preuve sont inspirées de l'obligation de preuve de boucle et nécessitent notamment de fournir un variant[7] caractérisant le plus long chemin entre P et Q. Ces travaux ont

[5] Propositional Linear Temporal Logic.

[6] Binary Decision Diagram. Structure de données utilisée pour représenter des fonctions booléennes, basée sur un graphe orienté ayant une racine et seulement deux feuilles (0 et 1).

[7] Par exemple, une expression entière positive ou nulle et qui décroît strictement à chaque exécution d'événement.

ensuite été étendus notamment par H. Ruìz Barradas et D. Bert [BB02, BB06], qui fournissent une méthode de vérification basée sur un logique inductive inspirée de UNITY [Cha88]. En faisant l'hypothèse d'une équité faible [Fra86], ils permettent de diminuer les restrictions sur les propriétés vérifiables. La notion d'équité faible peut se traduire par « *si un jour un événement e devient déclenchable et le reste pour toujours, alors cet événement sera infiniment souvent lancé* ». Grâce à cette hypothèse d'équité, ils autorisent le bégaiement. C'est-à-dire que le variant peut ne pas toujours strictement décroître, mais stagner, à condition que les événements qui s'exécutent pendant ce temps ne modifient pas l'état courant du système. En effet, il s'ensuit que nécessairement un jour un événement permettant de faire décroître le variant sera déclenché et donc de conclure sur le respect de la propriété par le modèle. Enfin, J. Groslambert [Gro07] propose d'exploiter les mécanismes introduits par Abrial et Mussat, pour prendre en compte l'ensemble de la logique temporelle linéaire (LTL). Son approche consiste à construire un système B événementiel M_ϕ à partir d'une propriété ϕ et de la spécification M, de telle sorte que la vérification des obligations de preuve de cohérence de M_ϕ garantissent le respect de la propriété ϕ par le modèle M. Grâce à ce codage de la vérification de propriétés, dans le langage B événementiel étendu par J-R. Abrial et L. Mussat [AM98], l'auteur permet de prendre en compte toute la logique LTL, et en particulier, l'imbrication des modificateurs modaux.

Comme cela a été introduit en section 1.4.4, nous proposons, dans cette thèse, d'exploiter notre approche d'explicitation des comportements pour aider l'utilisateur à vérifier des propriétés comportementales. Pour ce faire nous proposons trois techniques qui diffèrent par le langage d'expression de la propriété, mais pour lesquelles nous proposons une méthode syntaxique de vérification.

2.5 Synthèse

Dans ce chapitre, nous avons décrit les principaux travaux en rapport avec notre objectif d'aide à la compréhension, au développement et à la validation de modèles B. Nous avons également présenté différentes formes d'automates, ainsi que les principales méthodes d'extraction des comportements déjà proposées.

Dans cette thèse, nous proposons un formalisme de système de transitions symbolique qui permet de décrire finement les comportements d'un modèle. Après en avoir défini la sémantique sur les systèmes B événementiel, nous étendons la proposition de D. Bert et F. Cave [BC00] pour calculer les comportements d'un modèle B en caractérisant des conditions de franchissement des transitions. Par la suite, nous ajoutons la notion de hiérarchie dans les systèmes de transitions pour représenter les comportements de modèles B événementiel construits par raffinement. Cela permet d'une part de visualiser aisément le lien entre les variables raffinées et les variables abstraites et d'autre part de faciliter le lien entre la représentation des comportements de l'abstraction et celle du raffinement, car la structure générale du système de transitions est préservée. De plus, nous décrivons comment exploiter les propriétés du raffinement pour réduire le nombre d'obligations de preuve, et donc de maîtriser le risque de défaut de preuve. Ces méthodes ont été en grande parties implantées dans l'outil *GénéSyst*.

Pour terminer la description du contexte de cette thèse, le chapitre suivant décrit le formalisme

que nous utilisons par la suite : la méthode B.

La méthode B

3

program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Edsger Dijkstra

Sommaire

3.1 Notions ensemblistes **44**

3.2 Description par l'exemple d'un composant B **44**

3.3 Les substitutions généralisées **48**

3.3.1 Substitutions primitives 49

3.3.2 Substitutions dérivées 50

3.3.3 Forme normalisée d'une substitution 51

3.4 Calcul des obligations de preuve **52**

3.5 Raffinement **54**

3.5.1 Exemple de raffinement B 54

3.5.2 Le raffinement en B 56

3.6 L'approche B événementiel **58**

3.6.1 Introduction 59

3.6.2 Exemple de système B événementiel 59

3.6.3 Calcul des obligations de preuve 60

3.6.4 Le raffinement en B événementiel 61

3.6.5 Exemple de raffinement B événementiel 61

3.7 Outils **63**

3.8 Synthèse **63**

Afin de comprendre les fondements théoriques des travaux décrits dans ce manuscrit, il est nécessaire d'avoir une connaissance minimale de la méthode B et de ses principes. Dans ce chapitre, nous décrivons les principales caractéristiques de ce formalisme. Pour ce faire, nous commençons par décrire l'approche B classique, afin de mettre plus en évidence les particularités de l'approche B événementiel.

Dans ce chapitre, nous illustrons nos propos en introduisant, en section 3.2, un exemple de modèle décrit en B classique. Nous introduisons alors les substitutions généralisées dont nous donnons la sémantique en terme du calcul de la plus faible pré-condition. Nous donnons ensuite les

obligations de preuve associées à la validation d'un composant, et nous présentons le principe du raffinement par partie qui est utilisé en B. Enfin, nous terminons en mettant en évidence les particularité de l'approche B événementiel ainsi que les principaux outils du domaine. Mais en premier lieu, nous faisons un rappel succinct de quelques définitions et notations ensemblistes utilisées en B.

3.1 Notions ensemblistes

Le langage B permet de manipuler des données de type entier ou booléen, mais également des données caractérisées par des ensembles. Il est alors possible d'exprimer des propriétés logiques sur ces données. Dans cette section, nous proposons de faire un rappel des notations et des définitions ensemblistes utilisées en B. En particulier, une *relation* est un ensemble de couples et une *fonction* est une relation n'associant au plus qu'une image à chaque élément de son domaine. Si tous les éléments du domaine ont une image alors la fonction est dite *totale*, sinon elle est *partielle*.

Le tableau 3.1 décrit la syntaxe particulière de certains opérateurs ensemblistes que nous utiliserons par la suite. Le tableau 3.2 décrit, quant à lui, les principaux constructeurs de relations et de couples, tandis que le tableau 3.3 définit quelques opérateurs sur les relations dont nous aurons besoin pour décrire les exemples donnés dans cette thèse. Finalement, le tableau 3.4 définit les principaux types de fonctions.

Opérateur	Nom
$\emptyset$	Ensemble vide
$\mathbb{P}(P)$	Ensemble des sous-ensembles de P
$\mathbb{F}(P)$	Ensemble des sous-ensembles finis de P
$P_1 \times P_2$	Produit cartésien de P_1 par P_2

où P est un ensemble.

Tab. 3.1 – Opérateurs ensemblistes

Opérateur	Nom	Définition
$a \mapsto b$	Couple (a, b)	(a, b)
$A \leftrightarrow B$	Ensemble des relations entre A et B	$\mathbb{P}(A \times B)$
$\mathsf{id}(A)$	Relation d'identité sur A	$\{a \mapsto a \mid a \in A\}$

où A et B sont des ensembles et a et b des éléments de A et B.

Tab. 3.2 – Constructeurs de relations ou de couples

3.2 Description par l'exemple d'un composant B

Un composant B peut être une spécification abstraite (appelée *machine*), un raffinement ou une implantation. La description d'une machine commence par son nom (clause **machine**) et finit

Opérateur	Nom	Définition
$\mathsf{dom}(R)$	Domaine	$\{x \mid \exists y \cdot (x \mapsto y \in R)\}$
$\mathsf{ran}(R)$	Codomaine	$\{y \mid \exists x \cdot (x \mapsto y \in R)\}$
$R[A]$	Image	$\{y \mid \exists x \cdot (x \mapsto y \in R \ \wedge \ x \in A)\}$
R^{-1}	Inverse	$\{y \mapsto x \mid x \mapsto y \in R\}$
$R_1 \ ; R_2$	Composition	$\{x \mapsto z \mid \exists y \cdot (x \mapsto y \in R_1 \ \wedge \ y \mapsto z \in R_2)\}$
R^+	Fermeture transitive	$R \cup (R \ ; \ R) \cup (R \ ; \ R \ ; \ R) \cup \dots$ où $R \in A \leftrightarrow A$
R^*	Fermeture réflexive et transitive	$\mathsf{id}(A) \cup R^+$ où $R \in A \leftrightarrow A$

où A est un ensemble, R une relation et x, y et z des éléments du domaine ou du codomaine de R.

TAB. 3.3 – Opérateurs définis sur les relations

Opérateur	Nom	Définition
$A \mapsto B$	Ensemble des fonctions partielles	$\{f \mid f \in A \leftrightarrow B \ \wedge \ \forall(a,b,c) \cdot (a \mapsto b \in f \ \wedge \ a \mapsto c \in f \Rightarrow b = c)\}$
$A \rightarrow B$	Ensemble des fonctions totales	$\{f \mid f \in A \mapsto B \ \wedge \ \mathsf{dom}(f) = A\}$

où f est une fonction, A et B sont deux ensembles et a, b c des éléments de A ou de B.

TAB. 3.4 – Opérateurs de définition des fonctions

par **end**. Entre ces deux bornes se trouvent un ensemble de clauses décrivant les *données* (clauses **sets**, **constants** et **variables**), les *propriétés* (clauses **properties**, **invariant** et **assertions**) et les *traitements* (clauses **initialisation** et **operations**) du composant. Enfin, les opérations B ont la forme générale suivante :

$$\textit{Résultats} \longleftarrow \textit{NomOpération}(\textit{Paramètres}) \mathrel{\hat{=}} \textbf{pre } P \textbf{ then } S \textbf{ end}$$

où *Paramètres* et *Résultats* sont les listes des noms des paramètres entrant et sortant, le prédicat logique P décrit la pré-condition de l'opération (condition en dehors de laquelle l'opération ne peut pas être appelée) et la substitution généralisée S définit son corps.

Notons également que la substitution *simultanée* ($S_1 \ || \ S_2$) permet de définir l'exécution des deux substitutions S_1 et S_2 sans que l'ordre d'exécution ne soit défini (S_1 puis S_2 ou S_2 puis S_1). Pour introduire la notion d'ordre on utilise la substitution séquence : S_1 **;** S_2.

L'exemple ci-après est une machine *GestionObjets*, qui est inspirée de l'étude de cas réalisée dans le cadre du projet RNTL BOM. Elle modélise le processus de gestion dynamique des classes et de l'héritage intégré dans la machine virtuelle JavaCard. Cette machine fournit deux fonctionnalités (deux opérations) : charger une classe en mémoire et dire si deux classes présentes en mémoire héritent l'une de l'autre. Pour ce faire, il est nécessaire de définir une structure de données permettant de mémoriser les classes présentes en mémoire à un instant donné, ainsi que les données relatives à leur héritage. Les sources complètes dont est issu cet exemple sont disponibles sur in-

ternet[1].

La première opération (*InstanceOf*) prend deux classes C_1 et C_2 en paramètre et renvoie vrai si et seulement si la classe C_1 hérite de C_2. Pour pouvoir calculer ce résultat, nous avons donc besoin de maintenir une variable (*Super*), qui est une fonction renvoyant la classe mère de toute classe. Ainsi, le résultat renvoyé est vrai si, et seulement si, il existe un chemin de C_1 à C_2 dans la fermeture transitive $Super^+$.

La seconde opération (*Charger*) permet de charger une classe en mémoire. Afin de maintenir le lien d'héritage, deux paramètres sont requis : l'identificateur C_1 de la classe à charger et l'identificateur C_2 de sa classe mère. Pour que ce chargement soit possible, il est nécessaire que C_2 soit déjà chargée en mémoire, que C_1 ne le soit pas encore et que le nombre maximum de classes chargeables ne soit pas atteint. Cette opération renvoie alors vrai si et seulement si le chargement a pu être effectué.

Pour définir ces opérations nous avons besoin d'introduire l'ensemble des identificateurs de classes (*IdfClasses*), l'ensemble des identificateurs des classes chargées en mémoire (*LesClasses*), ainsi que le nombre de classes chargeables en mémoire simultanément (*ClassesMax*).

[1] http://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/?ZoomSur=ProjetBOM#ProjetBOM

Spécification 3.1 (*Machine de gestion des classes*) :

machine *GestionObjets*
sets *IdfClasses*
constants *ClassesMax*, *Object*
properties
$ClassesMax \in \mathsf{NAT}_1$ *(1)*
$\wedge\ Object \in IdfClasses$ *(2)*
variables *LesClasses*, *Super*
invariant
$LesClasses \subseteq IdfClasses$ *(3)*
$\wedge\ \mathsf{card}(LesClasses) \leq ClassesMax$ *(4)*
$\wedge\ Super \in (LesClasses - \{Object\}) \rightarrow LesClasses$ *(5)*
$\wedge\ Object \in LesClasses$ *(6)*
$\wedge\ \forall cc \cdot (cc \in \mathsf{dom}(Super) \Rightarrow (cc \mapsto Object) \in Super^+)$ *(7)*
$\wedge\ Super^+ \cap \mathsf{id}(IdfClasses) = \emptyset$ *(8)*
initialisation
$LesClasses := \{Object\} \parallel Super := \emptyset$
operations
$bb \longleftarrow InstanceOf(C_1, C_2) \mathrel{\hat=}$ **pre** $C_1 \in IdfClasses\ \wedge\ C_2 \in IdfClasses$ **then**
$bb :=$ **bool**$(C_1 \mapsto C_2 \in Super^+)$
end ;
$rr \longleftarrow Charger(C_1, C_2) \mathrel{\hat=}$ **pre** $C_1 \in IdfClasses\ \wedge\ C_2 \in IdfClasses$ **then**
if $C_1 \in LesClasses\ \vee\ C_2 \notin LesClasses\ \vee\ \mathsf{card}(LesClasses) = ClassesMax$
then $rr := \mathsf{false}$
else $rr := \mathsf{true} \parallel LesClasses := LesClasses \cup \{C_1\} \parallel Super := Super \cup \{C_1 \mapsto C_2\}$
end
end
end

L'ensemble *IdfClasses* des identificateurs de classes est un ensemble non vide dont la structure des éléments n'est pas définie. Nous le déclarons donc dans la clause **sets**.

L'ensemble des classes chargées en mémoire à un moment donné (*LesClasses*) est un sous-ensemble de *IdfClasses* *(3)* dont la taille est bornée par l'entier positif *ClassesMax* *(4)*. Comme la valeur de cette constante dépend de l'espace mémoire disponible sur la carte, nous ne lui donnons pas de valeur pour l'instant.

Pour modéliser l'arbre d'héritage, nous introduisons la variable *Super*, qui est une fonction de *LesClasses* vers *LesClasses*. Cet arbre a pour racine la classe *Object* *(2)* qui est la seule classe qui n'hérite de rien *(5)* et de laquelle héritent toutes les autres classes *(7)*. Celle-ci étant toujours présente en mémoire *(6)*, cela contraint *ClassesMax* a être différent de zéro *(1)*. Comme la fonction *Super* est totale et que tous les éléments de son domaine sont liés, par fermeture transitive, à *Object*, alors il n'est pas nécessaire de préciser qu'il n'y a pas de cycle dans la fonction *Super*. Toutefois, pour l'exemple, nous conservons cette propriété dans l'invariant *(8)*.

Enfin, seule la classe *Object* est chargée en mémoire à l'état initial. La fonction *Super* est donc vide tandis que l'ensemble *LesClasses* ne contient que *Object*.

Cet exemple nous sert dans les sections suivantes à illustrer la définition des substitutions généralisées et le calcul de la plus faible pré-condition. Ces notions sont importantes pour ensuite introduire la génération des obligations de preuve en B.

3.3 Les substitutions généralisées

La logique de Hoare [Hoa69], aussi appelée logique des programmes, permet de vérifier qu'un programme respecte des assertions logiques (prédicat logique). Les formules ont la forme générale :

$$\{P\}\ S\ \{R\}$$

et signifient que si l'assertion P est vraie et que le programme S termine, alors l'assertion R est vérifiée après exécution de S. Les prédicats P et R sont respectivement appelés la *pré-condition* et la *post-condition* de S. On dit d'une instruction qu'elle termine si son exécution s'arrête dans un temps fini et sans erreur (exemple : les boucles infinies et les divisions par 0 ne terminent pas).

E. Dijkstra [Dij76] a ensuite étendu cette approche en introduisant une méthode de calcul de la plus faible pré-condition P nécessaire pour que le programme S termine et que la post-condition R soit vérifiée. C'est ce calcul, appelé $\mathcal{WP}$ (pour Weakest Precondition), que l'on utilise en B.

En effet, dans le langage B, on définit la sémantique des instructions utilisées pour la spécification et le développement de programmes, appelées des substitutions généralisées, en terme du calcul de $\mathcal{WP}$. L'assignation $x := E$ est la substitution simple et correspond à l'axiome du calcul de $\mathcal{WP}$, noté $[x := E]R$, où E est une expression et R un prédicat (Exemple 3.1). Cette transformation est exactement la substitution de x par E dans R, c'est-à-dire le remplacement des occurrences libres[2] de x par E dans R.

Exemple 3.1 (*Substitution simple*) :

(1) $[LesClasses := LesClasses \cup \{C_1\}](Object \in LesClasses) \quad \equiv \quad Object \in (LesClasses \cup \{C_1\})$

(2) $[x := E](x > 0 \ \wedge\ \forall x \cdot P(x)) \quad \equiv \quad E > 0 \ \wedge\ \forall x \cdot P(x)$

Le calcul de $\mathcal{WP}$ est un transformateur de prédicat (il prend un prédicat et renvoie un prédicat) et peut donc être composé. Ainsi, la plus faible pré-condition d'une séquence de substitutions[3] peut être définie comme suit :

$$[S_1 \ ;\ S_2]R \quad \equiv \quad [S_1][S_2]R$$

Implicitement, cela signifie que la plus faible précondition pour que la substitution $S_1\ ;\ S_2$ termine et mène à R est la plus faible pré-condition telle que S_1 termine, que S_2 termine depuis les valeurs atteignables par S_1 et que S_1 vérifie la post-condition $[S_2]R$.

[2] L'occurrence d'une variable x est dite *libre* dans un prédicat P si elle est présente dans P et qu'elle n'est pas sous la portée un quantificateur ($\exists$, $\forall$, λ, $\{x \mid \ldots\}$). À l'inverse, x est dite *non libre* dans R si toutes les occurrences de x dans R sont introduites par un quantificateur (occurrences *liées*) ou si x est absente de R.

[3] Notée ' ; ', comme la composition de relations.

3.3.1 Substitutions primitives

Pour chacune des substitutions primitives[4], deux notations sont définies. La première notation, dite *mathématique*, est utilisée pour raisonner sur les prédicats logiques. La seconde, dite *syntaxique*, est proche de celle des instructions des langages de programmation "classiques". Elle est utilisée pour décrire les modèles. Le tableau 3.5 présente les substitutions primitives avec leurs deux notations.

Notation mathématique	Nom	Notation syntaxique
$x := E$	Substitution simple	$x := E$
$x, y := E, F$	Substitution multiple simple	$x, y := E, F$
skip	Substitution sans effet	**skip**
$P \mid S$	Substitution pré-conditionnée	**pre** P **then** S **end**
$P \Longrightarrow S$	Substitution gardée	**select** P **then** S **end**
$S_1 [] S_2$	Substitution choix borné	**choice** S_1 **or** S_2 **end**
$@ z \cdot S$	Substitution choix non borné	**var** z **in** S **end**
$S_1 \; ; \; S_2$	Substitution séquence	$S_1 \; ; \; S_2$

Dans ce tableau, x, y et z sont des variables, E et F sont des expressions, P est un prédicat logique et S est une substitution généralisée.

TAB. 3.5 – Substitutions primitives

Notons que certaines de ces substitutions sont l'apanage des spécifications et sont interdites dans les implantations, qui doivent pouvoir s'exécuter sur une machine physique. C'est notamment le cas de la substitution multiple simple, qui effectue les différentes substitutions en simultané, et des substitutions de choix (borné ou non), qui ne sont pas déterministes.

Enfin, les axiomes du calcul de $\mathcal{WP}$ sont donnés en définition 6.

Définition 6 (Axiomes du calcul de $\mathcal{WP}$ sur les substitutions primitives)

Cas de substitution	*Réduction*	*Condition*
$[x := E]R$	*Remplacement par E des occurrences de x libres dans R*	
$[x, y := E, F]R$	$[z := F][x := E][y := z]R$	$z \backslash E, F, R$
$[\textbf{skip}]R$	R	
$[P \mid S]R$	$P \wedge [S]R$	
$[P \Longrightarrow S]R$	$P \Rightarrow [S]R$	
$[S_1 [] S_2]R$	$[S_1]R \wedge [S_2]R$	
$[@ z \cdot S]R$	$\forall z \cdot [S]R$	$z \backslash R$
$[S_1 \; ; \; S_2]R$	$[S_1][S_2]R$	

Terminologie. *$z \backslash R$ signifie que la variable z n'est pas libre dans R.*

[4] Substitutions ne pouvant pas se décomposer en d'autres substitutions.

3.3.2 Substitutions dérivées

Les substitutions qui s'expriment en terme des substitutions primitives sont dites *dérivées*. Le tableau 3.6 décrit les substitutions dérivées les plus courantes.

Notation	Définition
$x := E \mid\mid y := F$	$x, y := E, F$
if P **then** S_1 **else** S_2 **end**	$(P \Longrightarrow S_1) \mathbin{[\!]} (\neg\ P \Longrightarrow S_2)$
if P **then** S **end**	$(P \Longrightarrow S) \mathbin{[\!]} (\neg\ P \Longrightarrow \textbf{skip})$
$x := \textbf{bool}(P)$	**if** P **then** $x := \mathsf{true}$ **else** $x := \mathsf{false}$ **end**
choice S_1 **or** S_2 ... **or** S_3 **end**	$S_1 \mathbin{[\!]} S_2 \ldots \mathbin{[\!]} S_3$
any z **where** P **then** S **end**	$@\, z \cdot (P \Longrightarrow S)$
$x :\in E$	**any** z **where** $z \in E$ **then** $x := z$ **end** *Avec x et z deux variables différentes et $z \backslash E$*

Tab. 3.6 – Quelques substitutions dérivées

À partir du modèle de gestion des classes présenté en section 3.2, nous avons extrait un exemple de calcul de $\mathcal{WP}$ (exemple 3.2) consistant à vérifier une propriété invariante.

Exemple 3.2 (*Calcul de $\mathcal{WP}$*) :

Soit la substitution S suivante :

$S \mathrel{\hat{=}} \textbf{if } \mathsf{card}(\mathit{LesClasses}) \neq \mathit{ClassesMax} \textbf{ then } \mathit{LesClasses} := \mathit{LesClasses} \cup \{C_1\} \textbf{ end}$

où C_1 n'appartient pas à *LesClasses*. Quelle propriété doit on avoir avant l'exécution de S pour que la propriété $\mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax}$ soit établie après l'exécution de S ?

$[S](\mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax})$

$\equiv_{\text{Par définition du if}}$

$$\left[\begin{array}{l}(\mathsf{card}(\mathit{LesClasses}) = \mathit{ClassesMax} \Longrightarrow \textbf{skip}) \mathbin{[\!]} \\ (\mathsf{card}(\mathit{LesClasses}) \neq \mathit{ClassesMax} \Longrightarrow \mathit{LesClasses} := \mathit{LesClasses} \cup \{C_1\})\end{array}\right](\mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax})$$

$\equiv_{\text{Par calcul de } \mathcal{WP}}$

$$\begin{array}{l}\quad (\mathsf{card}(\mathit{LesClasses}) = \mathit{ClassesMax} \Rightarrow \mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax}) \\ \wedge\ (\mathsf{card}(\mathit{LesClasses}) \neq \mathit{ClassesMax} \Rightarrow \mathsf{card}(\mathit{LesClasses} \cup \{C_1\}) \leq \mathit{ClassesMax})\end{array}$$

$\equiv_{\text{Par simplification logique et comme } C_1 \notin \mathit{LesClasses}}$

$(\mathsf{card}(\mathit{LesClasses}) = \mathit{ClassesMax} \ \vee\ \mathsf{card}(\mathit{LesClasses}) + 1 \leq \mathit{ClassesMax})$

$\equiv$

$\mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax}$

Il est donc nécessaire et suffisant que $\mathsf{card}(\mathit{LesClasses}) \leq \mathit{ClassesMax}$ soit vrai avant exécution de S pour l'être après.

3.3.3 Forme normalisée d'une substitution

Pour finir cette section sur les substitutions généralisées, nous présentons deux prédicats permettant d'introduire une forme normalisée pour toute substitution [Abr96b]. En particulier, cette forme permet de simplifier les preuves de la méta-théorie[5], car elle permet de traiter les substitutions de manière générique, sans avoir à effectuer une démonstration par induction sur l'ensemble des substitutions primitives.

La terminaison d'une substitution S (notée $\mathsf{trm}(S)$) est la condition sous laquelle S permet d'établir une quelconque post-condition R, tandis que le prédicat avant/après (noté $\mathsf{prd}_x(S)$) caractérise l'évolution des variables x entre les états d'avant et ceux après une substitution S. Si x désigne un ensemble de variables, alors le prédicat avant-après de la substitution S est l'ensemble des couples (x, x') tels que x désigne les valeurs de l'état avant exécution de S et x' désigne celles après exécution de S. On définit ces deux prédicats de la manière suivante :

Définition 7 (Terminaison et prédicat avant/après d'une substitution)

Symbole	*Définition*	*Définition mathématique*
$\mathsf{trm}(S)$	*La terminaison de la substitution S*	$[S](x = x)$
$\mathsf{prd}_x(S)$	*Le prédicat avant-après*	$\langle S\rangle(x' = x)$

Les résultats de ces calculs sur les principales substitutions généralisées sont décrits en annexes A.1 et A.2.

La notation $\langle S\rangle R$ est équivalente à $\neg[S]\neg R$. Celle-ci a été introduite par C. Morgan [WADJ90], puis adaptée en B par M. Butler [But00]. Elle désigne le conjugué du calcul de $\mathcal{WP}$ (noté $\mathcal{WP}^{cg}$). Cette notation étant fréquemment utilisée par la suite, nous décrivons les résultats de ce calcul en annexe A.3.

La forme normalisée d'une substitution généralisée est alors définie comme suit :

Définition 8 (Forme normalisée d'une substitution) *Si S est une substitution généralisée, alors elle peut toujours être écrite ainsi :*

$$S = \mathsf{trm}(S) \mid @\, x' \cdot (\mathsf{prd}_x(S) \implies x := x') \qquad \textit{si } x' \backslash \mathsf{trm}(S)$$

Cette définition permet de décrire de manière systématique un prédicat logique caractérisant toute substitution S, établissant ainsi le lien entre les substitutions généralisées et les approches basées sur la logique telles que Z [Abr80, Dil94, Spi93], VDM [Jon86] ou TLA [Lam94a].

Voici un exemple de calcul de forme normalisée :

[5] Preuves portant sur la sémantique des substitutions et non sur leur utilisation.

Exemple 3.3 (*Forme normalisée*) :

Soit la substitution suivante : $x > 0 \Longrightarrow x := x + 1$
Sa terminaison et son prédicat avant/après sont :

$\mathsf{trm}(x > 0 \Longrightarrow x := x + 1) \Leftrightarrow \mathsf{btrue}$

$\mathsf{prd}_x(x > 0 \Longrightarrow x := x + 1) \Leftrightarrow (x > 0 \ \wedge\ x' = x + 1)$

Et sa forme normalisée est donc : $\mathsf{btrue} \mid @\, x' \cdot (x > 0 \ \wedge\ x' = x + 1 \ \Longrightarrow\ x := x')$

Terminologie. *Dans le langage B, les notations* btrue *et* bfalse *symbolisent les prédicats respectivement toujours vrai et toujours faux, par opposition aux valeurs booléennes* true *et* false.

3.4 Calcul des obligations de preuve

En B, les obligations de preuve associées aux machines ont pour but de garantir, par construction, que l'invariant est toujours établi. Elles consistent à vérifier que l'initialisation établit l'invariant et que si l'invariant est vrai alors il est préservé par l'appel d'une opération.

machine	M	*/* Nom de la machine */*
sets	R ;	*/* Ensembles abstraits */*
	$T = \{a, b\}$	*/* Ensembles énumérés */*
constants	c	*/* Constantes */*
properties	P	*/* Spécification des constantes */*
variables	v	*/* Variables */*
invariant	I	*/* Spécification des variables */*
initialisation	U	*/* Initialisation des variables */*
operations		*/* Liste d'opérations */*
$Res \longleftarrow Op(Params) \mathrel{\hat=}$ **pre** Q **then** S **end**		*/* Forme générale d'une opération */*
end		

FIG. 3.1 – Forme générale d'une machine abstraite B

Par la suite, nous considérons que toute machine B a la forme générale donnée en figure 3.1. Le B-Book décrit également un mécanisme de paramétrage des composants, mais celui-ci étant peu utilisé dans la pratique, nous laissons le lecteur intéressé se référer à [Abr96b, pp. 236]. Notons $\mathcal{B}$ le prédicat décrivant les propriétés des constantes (P), des ensembles abstraits (R) et des ensembles énumérés (T), tel que :

$$\mathcal{B} \equiv P \ \wedge\ R \in \mathbb{P}_1(\mathsf{INT}) \ \wedge\ T \in \mathbb{P}_1(\mathsf{INT}) \ \wedge\ T = \{a, b\} \ \wedge\ a \neq b$$

où le prédicat $\mathbb{P}_1(E)$ caractérise l'ensemble des sous-parties non vides de E et INT est l'ensemble des entiers relatifs compris entre minint et maxint[6]. Il est intéressant de noter que, les ensembles abstraits et les ensembles énumérés sont codés par des ensembles sur les entiers. À partir de ces

[6] Les constantes minint et maxint caractérisent l'ensemble des entiers utilisables nativement par une machine.

simplifications d'écriture, nous pouvons exprimer les obligations de preuve du composant de la figure 3.1 comme suit :

Conditions de validité d'une machine	
Initialisation	$\mathcal{B} \Rightarrow [U]I$
Pour chaque opération	$I \wedge \mathcal{B} \wedge Q \Rightarrow [S]I$

Avant l'initialisation, l'invariant n'est pas établi. Il s'ensuit que seules les propriétés sur les constantes sont en hypothèse, tandis qu'avant chaque appel d'opération on a comme hypothèse que l'invariant et les propriétés sur les constantes sont établis, de même que la pré-condition de l'opération. En effet, c'est lors de l'appel d'une opération que l'on vérifie que l'on établit sa pré-condition. Celle-ci est donc une hypothèse pour l'exécution de l'opération. Dans les deux obligations de preuve, l'exécution de la substitution, depuis les hypothèses, doit nécessairement établir l'invariant.

Pour finir cette présentation du calcul de $\mathcal{WP}$, voici un exemple de vérification d'invariant tiré de la spécification 3.1. Nous voulons vérifier que l'opération *Charger* préserve l'invariant :

$$Super \in (LesClasses - \{Object\}) \rightarrow LesClasses$$

Obligation de preuve 1 : Préservation de l'invariant par *Charger (Spécification 3.1)*

Hypothèses :

Invariant $I = Super \in (LesClasses - \{Object\}) \rightarrow LesClasses$
Aucune propriété $\mathcal{B}$ *sur les constantes n'est nécessaire.*
Pré-condition $Q = C_1 \in IdfClasses \wedge C_2 \in IdfClasses$

But :

$$\begin{bmatrix} \textbf{if } C_1 \in LesClasses \vee C_2 \notin LesClasses \vee \mathsf{card}(LesClasses) = ClassesMax \\ \textbf{then } rr := \mathsf{false} \\ \textbf{else } rr := \mathsf{true} \mid\mid LesClasses := LesClasses \cup \{C_1\} \\ \quad\quad \mid\mid Super := Super \cup \{C_1 \mapsto C_2\} \\ \textbf{end} \end{bmatrix} \begin{pmatrix} Super \in \\ (LesClasses - \{Object\}) \rightarrow LesClasses \end{pmatrix}$$

Or rr n'apparaît pas dans la post-condition et le but est présent dans l'invariant, qui est en hypothèse. Le cas **then** *est donc trivialement vrai et il reste donc :*

$$\equiv \left[\begin{pmatrix} C_1 \notin LesClasses \wedge \\ C_2 \in LesClasses \wedge \\ \mathsf{card}(LesClasses) \neq ClassesMax \end{pmatrix} \Longrightarrow \begin{pmatrix} LesClasses := LesClasses \cup \{C_1\} \\ \mid\mid Super := Super \cup \{C_1 \mapsto C_2\} \end{pmatrix} \right] \begin{pmatrix} Super \in \\ (LesClasses - \{Object\}) \rightarrow LesClasses \end{pmatrix}$$

Par simplification et application des règles de $\mathcal{WP}$ *on obtient :*

$$\equiv \begin{pmatrix} C_1 \notin LesClasses \wedge C_2 \in LesClasses \\ \wedge \mathsf{card}(LesClasses) \neq ClassesMax \end{pmatrix} \Rightarrow \begin{pmatrix} (Super \cup \{C_1 \mapsto C_2\}) \in \\ ((LesClasses \cup \{C_1\}) - \{Object\}) \rightarrow (LesClasses \cup \{C_1\}) \end{pmatrix}$$

Dans un premier temps, nous pouvons établir que $Super \cup \{C_1 \mapsto C_2\}$ *est une fonction, car :*

1. *Super est une fonction (d'après l'invariant* I*) ;*
2. $\{C_1 \mapsto C_2\}$ *est aussi une fonction ;*
3. *Les domaines de Super et* $\{C_1 \mapsto C_2\}$ *sont disjoints* $(\mathsf{dom}(Super) \subseteq LesClasses$ *et* $C_1 \notin LesClasses)$

Enfin, cette fonction est totale, car $C_1 \neq Object \ \wedge \ \{C_1\} \cup \mathsf{dom}(Super) = LesClasses \cup \{C_1\} - \{Object\}$*. Cette opération préserve donc la propriété* $Super \in (LesClasses - \{Object\}) \rightarrow LesClasses$*.*

□

3.5 Raffinement

La notion de raffinement a été initialement introduite en 1976 par E. Dijkstra [Dij76], puis étendue par R. J. Back [Bac78]. Cette notion est définie comme la vérification de la préservation de la correction d'un programme vis-à-vis de sa spécification.

Le raffinement est une partie importante de la méthode B, qui consiste à rendre plus concret un modèle abstrait en précisant les algorithmes et en modifiant la représentation des données. Le raffinement est défini par la notion d'observateur externe : un raffinement doit pouvoir se substituer au modèle qu'il raffine sans que cela soit perceptible de l'extérieur.

Dans cette section, nous définissons formellement ce qu'est un raffinement en B, en explicitant les différentes obligations de preuve qui permettent d'en garantir la correction. Pour introduire cette notion, nous commençons par un exemple concret de raffinement.

3.5.1 Exemple de raffinement B

Prenons l'exemple de la machine *GestionObjets*. Nous proposons de modifier sa structure de données pour aller vers une implantation. La variable *SommetClasses* caractérise le nombre de classes (*Object* mise à part) chargées en mémoire à un instant donné *(1)*. Nous introduisons également un codage permettant de représenter chaque classe chargée en mémoire par un unique entier. L'ensemble *LesClasses* est alors remplacé par *CodeVersClasses*, qui est une fonction bijective[7] associant chaque classe chargée en mémoire à un code numérique *(2)*. Nous remplaçons également la fonction *Super* par la fonction *TabSuper* où l'on ne considère plus le nom des classes mais leur code numérique associé *(4 et 5)*. Ces deux fonctions sont utilisées comme des piles. Seules les $SommetClasses+1$ premières cases de *CodeVersClasses* et les *SommetClasses* premières cases de *TabSuper* sont utilisées *(2 et 4)*. Ainsi, le code numérique d'une classe est choisi à son chargement comme étant l'index de la première case vide de la pile (numéro $SommetClasses+1$). Seule la classe *Object* conserve le même code numérique 0 *(3)*, pour toutes les exécutions possibles du système.

[7] Une fonction f est dite *bijective* si et seulement si chaque élément de son image est associé à exactement 1 antécédent de son domaine. On note $f \in A \rightarrowtail\!\!\!\!\twoheadrightarrow B$ une fonction bijective entre les ensembles A et B.

Spécification 3.2-1 (*Structure de données du raffinement proposé*) :

variables $SommetClasses, CodeVersClasses, TabSuper$
invariant
$SommetClasses \in 0..ClassesMax - 1$ (1)
$\wedge\ CodeVersClasses \in 0..SommetClasses \rightarrowtail\!\!\!\!\twoheadrightarrow LesClasses$ (2)
$\wedge\ (0 \mapsto Object) \in CodeVersClasses$ (3)
$\wedge\ TabSuper \in 1..SommetClasses \rightarrow 0..SommetClasses$ (4)
$\wedge\ Super = (CodeVersClasses^{-1}\ ;\ TabSuper\ ;\ CodeVersClasses)$ (5)
end

Cet invariant décrit les nouvelles variables en fonction des variables abstraites, ce qui permet de conserver certaines propriétés telles que l'absence de cycle. On doit alors définir une initialisation compatible avec l'initialisation abstraite :

Spécification 3.2-2 (*Initialisation raffinée*) :

initialisation
$SommetClasses := 0$ **;** $CodeVersClasses := \{0 \mapsto Object\}$ **;** $TabSuper := \emptyset$

Enfin, les opérations doivent être redéfinies en fonction des nouvelles variables, tout en restant compatibles d'un point de vue observationnel. C'est-à-dire que tous les résultats possibles, pour une entrée donnée, doivent correspondre à des résultats dans l'abstraction. Nous proposons les opérations suivantes :

Spécification 3.2-3 (*Opérations InstanceOf et Charger raffinées*) :

$bb \longleftarrow InstanceOf(C_1, C_2) \mathrel{\hat{=}}$ **pre** $C_1 \in IdfClasses \ \wedge\ C_2 \in IdfClasses$ **then**
$bb :=$ **bool**$(C_1 \mapsto C_2 \in (CodeVersClasses^{-1}\ ;\ TabSuper^{+}\ ;\ CodeVersClasses)$
end **;**

$rr \longleftarrow Charger(C_1, C_2) \mathrel{\hat{=}}$ **pre** $C_1 \in IdfClasses \ \wedge\ C_2 \in IdfClasses$ **then**
if $C_1 \in \mathsf{ran}(CodeVersClasses) \vee C_2 \notin \mathsf{ran}(CodeVersClasses) \vee SommetClasses + 1 = ClassesMax$
then $rr := \mathsf{false}$
else
$rr := \mathsf{true}$ **;**
$SommetClasses := SommetClasses + 1$ **;**
$CodeVersClasses(SommetClasses) := C_1$ **;**
$TabSuper(SommetClasses) := CodeVersClasses^{-1}(C_2)$
end
end

Dans la section suivante, nous décrivons plus spécifiquement les principes du raffinement B et les obligations de preuve associées. Nous illustrons alors nos propos avec l'exemple que nous venons de présenter.

3.5.2 Le raffinement en B

Dans un premier temps, nous définissons le raffinement des substitutions généralisées avant d'introduire le raffinement de composants B et les obligations de preuve associées. Le calcul de ces dernières et les garanties que leur vérification apporte sont exploités dans les chapitres suivants pour valider nos choix liés à la génération de systèmes de transitions.

3.5.2.1 Raffinement des substitutions généralisées

On appelle *invariant de liaison* un prédicat L mettant en relation les variables x d'une spécification avec les variables y de son raffinement. Pour la suite, nous admettrons, que ces ensembles x et y sont toujours disjoints. Si ce n'était pas le cas, il suffirait de renommer, dans le raffinement, chaque variable v appartenant à x et y (en $v_{Renommée}$ par exemple) et de rajouter le lien $v = v_{Renommée}$ dans L. On définit alors le raffinement d'une substitution généralisée de la manière suivante :

Définition 9 (Raffinement des substitutions) *Une substitution S_A est raffinée par une substitution S_R suivant un invariant de liaison L si S_R peut au moins traiter les mêmes données que S_A et que tous les résultats possibles de S_R ont été prévus dans S_A :*

$$L \ \wedge\ \mathsf{trm}(S_A) \Rightarrow [S_R]\langle S_A\rangle L$$

Terminologie. *L'assertion « S_A est raffinée par S_R suivant L » est notée $S_A \sqsubseteq_L S_R$.*

Cette définition du raffinement a été initialement proposée par D. Gries et J. Prins [GP85] puis étendue à la méthode B par [Abr96b]. Notons que l'équivalence de cette définition avec le raffinement des données de la théorie du raffinement dans le cas du *forward data refinement* [GM93] a été prouvée dans [CU89] et appliquée à B dans [Rou99].

Exemple 3.4 (*Raffinement de substitutions*) :

La substitution $S_R \mathrel{\hat=} y \in \mathbb{N} \mid y := y + 2$
raffine la substitution $S_A \mathrel{\hat=} x \in 0..3 \mid (x := x + 1 \ [\!]\ x := 8)$
suivant l'invariant de liaison $L \mathrel{\hat=} y = x * 2$

Pour vérifier cette assertion, posons :

$L \ \wedge\ \mathsf{trm}(S_A) \Rightarrow [S_R]\langle S_A\rangle L$
$\equiv L \ \wedge\ \mathsf{trm}(S_A) \Rightarrow [S_R]\langle x \in 0..3 \mid (x := x + 1 \ [\!]\ x := 8)\rangle(y = x * 2)$
$\equiv L \ \wedge\ \mathsf{trm}(S_A) \Rightarrow [y \in \mathbb{N} \mid y := y + 2](x \in 0..3 \Rightarrow (y = (x + 1) * 2 \ \vee\ y = 8 * 2))$
$\equiv L \ \wedge\ \mathsf{trm}(S_A) \Rightarrow y \in \mathbb{N} \Rightarrow (x \in 0..3 \Rightarrow (y + 2 = x * 2 + 2 \ \vee\ y + 2 = 16))$
$\equiv L \ \wedge\ \mathsf{trm}(S_A) \ \wedge\ y \in \mathbb{N} \ \wedge\ x \in 0..3 \Rightarrow (y = x * 2 \ \vee\ y = 14)$

or $\mathsf{trm}(x \in 0..3 \mid (x := x + 1 \ [\!]\ x := 8)) \equiv x \in 0..3$, on a donc :

$\equiv y = x * 2 \ \wedge\ y \in \mathbb{N} \ \wedge\ x \in 0..3 \Rightarrow (y = x * 2 \ \vee\ y = 14)$

Ce qui est trivialement vrai. On a donc effectivement :

$$(x \in 0..3 \mid (x := x + 1 \ [\!]\ x := 8)) \quad \sqsubseteq_{y=x*2} \quad (y \in \mathbb{N} \mid y := y + 2)$$

3.5.2.2 Raffinement d'un composant B

Dans la méthode B, le raffinement est effectué par parties (*partwise refinement*). C'est-à-dire que, contrairement au cas général où l'on cherche à prouver que toute suite d'appels d'opérations est correctement raffinée, il suffit que l'initialisation et chaque opération soient correctement raffinées. Il est alors nécessaire de montrer que l'initialisation $Init_R$ raffine l'initialisation $Init_A$ et chaque opération Op_A est raffinée par une opération Op_R plus concrète.

machine M **sets** R ; $T = \{a, b\}$ **constants** c **properties** P **variables** v **invariant** I **initialisation** U **operations** $Res \longleftarrow Op(Params) \hat{=}$ **pre** Q **then** S **end** **end**	**refinement** N **refines** M **sets** R_R ; $T_R = \{c, d\}$ **constants** c_R **properties** P_R **variables** v_R **invariant** I_R **initialisation** U_R **operations** $Res \longleftarrow Op(Params) \hat{=}$ **pre** Q_R **then** S_R **end** **end**

FIG. 3.2 – Définition d'une machine M et de son raffinement N

Notons $\mathcal{B}$ les propriétés sur les constantes et les ensembles de la machine et $\mathcal{B}_r$ celles du raffinement, que nous définissons comme suit :

$$\mathcal{B} \equiv P \ \wedge \ R \in \mathbb{P}_1(\mathsf{INT}) \ \wedge \ T \in \mathbb{P}_1(\mathsf{INT}) \ \wedge \ T = \{a, b\} \ \wedge \ a \neq b$$
$$\mathcal{B}_R \equiv P_R \ \wedge \ R_R \in \mathbb{P}_1(\mathsf{INT}) \ \wedge \ T_R \in \mathbb{P}_1(\mathsf{INT}) \ \wedge \ T_R = \{c, d\} \ \wedge \ c \neq d$$

Si l'on considère la forme générale d'une machine M et de son raffinement N, comme montré dans la figure 3.2, alors l'invariant de liaison L est défini par $I \ \wedge \ I_R$. En nous basant sur la définition 9, nous pouvons donc déduire les obligations de preuve présentées en tableau 3.7 et qui permettent d'établir que N est un raffinement correct de M.

Initialisation	$\mathcal{B} \wedge \mathcal{B}_R \Rightarrow [U_R]\langle U\rangle I_R$
Pré-condition	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge Q \Rightarrow Q_R$
Opération sans résultat	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge Q \Rightarrow [S_R]\langle S\rangle I_R$
Opération avec résultat	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge Q \Rightarrow [[Res := Res_R]S_R]\langle S\rangle(I_R \wedge Res = Res_R)$ *Si* $Res_R \backslash Res, S_R, S, I_R$

Notons que la substitution $[Res := Res_R]S_R$ *est définie [Abr96b] comme le remplacement des occurrences libres de* Res *par* Res_R *dans* S_R.

TAB. 3.7 – Obligations de preuve à vérifier pour prouver la correction d'un raffinement

Notons que, contrairement à la définition 9, ces formules consistent à établir la préservation de l'invariant I_R du raffinement et non de l'invariant L de liaison. En effet, le mécanisme de

preuve des raffinements B se base sur le fait que les obligations de preuve associées à l'abstraction sont vérifiées par ailleurs. Nous considérons donc que, pour toute opération abstraite $Q \mid S$, alors $\mathcal{B} \wedge I \wedge Q \Rightarrow [S]I$ est établi. Ce qui permet de simplifier les obligations de preuve comme présenté en tableau 3.7.

De plus, dans le cas d'une opération avec résultat le but I_R est complété par $Res = Res_R$, où Res est le résultat de l'opération abstraite et Res_R le résultat de l'opération raffinée. En effet, le raffinement est une action qui doit être invisible pour un observateur extérieur. Il faut donc que, pour une entrée donnée, les résultats du raffinement aient été prévus dans l'abstraction.

Pour terminer cette section, voici un exemple, consistant à vérifier le raffinement de l'opération *InstanceOf*, introduite en spécification 3.1 et raffinée en spécification 3.2-3.

Obligation de preuve 2 : Cas de l'opération *InstanceOf*

Sous l'hypothèse des constantes abstraites et raffinées ($\mathcal{B} \wedge \mathcal{B}_R$), des invariants abstraits et raffinés ($I \wedge I_R$) et de la précondition de l'opération abstraite (Q), on doit montrer que :

$$[[bb := bb_R]InstanceOf_R]\langle InstanceOf\rangle(I_R \wedge bb = bb_R)$$

Ce qui se ramène à :

$$\equiv [[bb := bb_R]InstanceOf_R]\langle bb := \mathbf{bool}(C_1 \mapsto C_2 \in Super^+)\rangle(I_R \wedge bb = bb_R)$$

$$\equiv \left[bb_R := \mathbf{bool}\left(\begin{array}{c} C_1 \mapsto C_2 \in \\ (CodeVersClasses^{-1} ; TabSuper^+ ; CodeVersClasses) \end{array}\right)\right](I_R \wedge \mathbf{bool}(C_1 \mapsto C_2 \in Super^+) = bb_R)$$

$$\equiv (I_R \wedge ((C_1 \mapsto C_2 \in Super^+) \Leftrightarrow (C_1 \mapsto C_2 \in (CodeVersClasses^{-1} ; TabSuper^+ ; CodeVersClasses)))$$

Ce qui est vrai sous l'hypothèse de l'invariant I_R du raffinement, qui contient notamment :

$$Super = (CodeVersClasses^{-1} ; TabSuper ; CodeVersClasses)$$
$$\wedge\ CodeVersClasses \in 0..SommetClasses \rightarrowtail\!\!\!\!\twoheadrightarrow LesClasses$$

□

3.6 L'approche B événementiel

Dans les sections précédentes, nous avons décrit l'approche originelle de la méthode B, aussi appelée B classique ou B logiciel. Celle-ci permet de décrire des composants en terme de données et de traitement. L'extension B événementiel, que nous allons présenter dans cette section, permet de décrire et de raffiner également le contrôle du système. C'est J-R. Abrial, qui a, le premier, travaillé sur l'intégration du contrôle dans un modèle B [Abr96a, AM98]. D'autres travaux ont ensuite été menés avec d'autres approches, parmi lesquelles nous pouvons citer la proposition de M. Butler consistant à décrire les comportements voulus en CSP, puis à les traduire en B [But00].

Dans le cadre de cette thèse, nous nous intéressons à l'extraction et à la représentation de l'ensemble des comportements d'un modèle B. Cette extraction n'a de sens que dans le cadre du B événementiel puisque dans le cas du B classique le contrôle est contenu dans le composant appelant. Nous verrons par la suite qu'il est tout de même possible de s'intéresser à la représentation des comportements d'un modèle B classique en nous ramenant à du B événementiel.

3.6.1 Introduction

En B événementiel, on ne décrit pas des opérations qui s'exécutent lorsqu'on les appelle, mais des événements qui se déclenchent spontanément. Chaque événement est composé d'une garde G et d'un corps S et a la forme générale suivante :

$$ev \mathrel{\hat{=}} \textbf{select } G \textbf{ then } S \textbf{ end}$$
ou
$$ev \mathrel{\hat{=}} \textbf{any } v \textbf{ where } G \textbf{ then } S \textbf{ end}$$

Si la garde d'un événement est vérifiée, alors son corps est exécuté. Si plusieurs événements sont déclenchables, alors l'un d'entre eux est choisi de manière non déterministe. Si dans un état donné aucun événement n'est déclenchable, alors le système se bloque. On parle de *deadlock*. À l'inverse, si dans un état donné un événement peut prendre la main indéfiniment, alors on parle de *livelock*. Enfin, notons que, en B événementiel, toute substitution termine ($\mathsf{trm}(S) \equiv \mathsf{true}$). En particulier, la substitution pré-conditionnée n'est pas autorisée en B événementiel.

Contrairement au cas du B classique, lors du développement d'un modèle B événementiel, le concepteur est amené à coder le contrôle des événements. En effet, contrairement aux pré-conditions du B classique qui permettent de caractériser les paramètres et de garantir la terminaison de l'opération, les gardes des événements B ne servent qu'à définir depuis quel état du système l'événement est déclenchable. Cela correspond à un codage du contrôle par des variables. C'est pourquoi, un développement B événementiel commence souvent par un dessin, fait au brouillon, et décrivant les ordonnancements possibles, afin d'aider le concepteur dans son choix du codage du contrôle.

Dans la section suivante, nous proposons un exemple de modèle B événementiel en décrivant la démarche menant à sa conception.

3.6.2 Exemple de système B événementiel

En B événementiel, le composant de spécification le plus abstrait est appelé un système (par opposition à la machine du B classique) et les événements d'un composant B événementiel sont réunis dans la clause **events** (au lieu de la clause **operations**).

Pour illustrer l'approche B événementiel, nous proposons de détailler la démarche de conception d'un modèle. Partant de l'idée de modéliser un canal de communication, un concepteur B événementiel sera généralement amené à en représenter informellement les comportements (figure 3.3). Il souhaite, dans un premier temps, considérer 3 actions principales : l'envoi d'un message, son traitement et l'abandon d'un traitement. Il choisit par exemple de caractériser l'événement *Envoyer*, qui émet, en une fois, un message composé de *TailleEnvoi* éléments, qui sont ensuite reçus un par un par l'événement *Traiter*. Une réception peut être annulée à tout moment par l'événement *Reset*.

La phase de modélisation B peut alors commencer. La spécification 3.3 est un exemple de spécification répondant au cahier des charge initial. Cet exemple sera utilisé dans les chapitres suivants pour illustrer nos travaux portant sur le B événementiel.

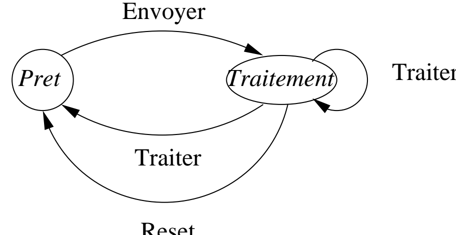


FIG. 3.3 – Dessin informel représentant les comportements attendus

Spécification 3.3 (*Canal de communication*) :

system *Canal_De_Communication*
variables *TailleEnvoi*
invariant $TailleEnvoi \in \mathbb{N}$
initialisation $TailleEnvoi := 0$
events
$Envoyer \mathrel{\hat{=}}$ **select** $TailleEnvoi = 0$ **then** $TailleEnvoi :\in \mathbb{N}_1$ **end ;**
$Traiter \mathrel{\hat{=}}$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := TailleEnvoi - 1$ **end ;**
$Reset \mathrel{\hat{=}}$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := 0$ **end**
end

Dans cet exemple, si la variable *TailleEnvoi* n'est pas nulle, donc s'il reste des données envoyées et non traitées, alors les gardes des deux événements *Traiter* et *Reset* sont vérifiées. Le choix de l'événement à exécuter est alors fait de manière non-déterministe. Pour mémoire, la substitution *devient élément de*, notée $:\in$, est présentée dans le tableau 3.6, section 3.3.2.

3.6.3 Calcul des obligations de preuve

De même qu'en B classique, les obligations de preuve associées à un système B événementiel permettent de vérifier que l'initialisation établit l'invariant et que chacun des événements préserve l'invariant. Nous ne parlons pas ici des cas particuliers liés aux livelocks et aux deadlocks, car on peut vouloir les autoriser dans le cas d'une spécification abstraite. Nous ne nous soucions donc pas de la vérification de leur absence.

Si l'on note I l'invariant du système et $\mathcal{B}$ les propriétés des constantes et des ensembles, alors les obligations de preuve à vérifier sont :

Conditions de validité d'un système B événementiel	
Initialisation U	$\mathcal{B} \Rightarrow [U]I$
Pour chaque événement $G \Longrightarrow S$	$I \wedge \mathcal{B} \wedge G \Rightarrow [S]I$

3.6.4 Le raffinement en B événementiel

En B événementiel, le raffinement découle des mêmes principes que le raffinement B classique. Il permet donc d'affiner les algorithmes et de changer la représentation des structures de données. Cependant, comme les événements sont gardés et non pas pré-conditionnés, les obligations de preuve sont modifiées. En effet, à l'inverse des pré-conditions qui ne peuvent être qu'affaiblies par raffinement, les gardes peuvent être renforcées. Cette différence permet alors d'affiner le contrôle, notamment en introduisant de nouveaux événements durant le processus de raffinement.

La raison de cette différence est qu'en B classique une opération raffinée doit pouvoir être appelée avec n'importe quelle entrée autorisée dans l'abstraction, tandis qu'en B événementiel le système est autonome. Cela permet de scinder les événements par raffinement et ainsi d'introduire de nouveaux événements. Cependant, ces derniers doivent raffiner **skip**. C'est l'équivalent du *stuttering* ou *bégaiement* en TLA [Lam94b]. Les nouveaux événements ne doivent pas pouvoir prendre la main indéfiniment. On parle alors de *livelock*. Pour cela, une nouvelle clause a été introduite : le **variant**. Cette clause contient une expression entière qui doit décroître strictement lors de l'exécution d'un nouvel événement, tout en restant positive.

Le tableau suivant résume les obligations de preuve associées au raffinement d'un composant B événementiel, où I et $\mathcal{B}$ sont respectivement l'invariant et les propriétés sur les constantes et les ensembles du système abstrait, où I_R et $\mathcal{B}_R$ sont respectivement l'invariant et les propriétés sur les constantes et les ensembles du système raffiné et où L est l'invariant de liaison définit par $L = I \wedge I_R$.

Obligations de preuve liées à la correction d'un raffinement B événementiel	
Initialisation	$\mathcal{B} \wedge \mathcal{B}_R \Rightarrow [U_R]\langle U\rangle I_R$
Garde	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge G_R \Rightarrow G_A$
événement déjà existant	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge G_R \Rightarrow [S_R]\langle S_A\rangle I_R$
Nouvel événement $ev_N \hat{=} G_N \Longrightarrow S_N$	
Correction du raffinement	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \wedge G_N \Rightarrow [S_N]I_R$
Pas d'introduction de livelock	$\mathcal{B} \wedge \mathcal{B}_R \wedge L \Rightarrow [n := V][ev_N](V < n)$ *Si* $n \backslash \mathcal{B}, \mathcal{B}_R, L, ev_N, V$

3.6.5 Exemple de raffinement B événementiel

Enfin, pour terminer cette section, voici une proposition de raffinement de la spécification 3.3. Celle-ci introduit un buffer[8] dans le canal de communication de telle sorte que le nombre de messages en attente d'être traités est borné. C'est le nouvel événement *EnvoyerSuite* qui émet alors les messages un par un, permettant ainsi un traitement au fur et à mesure des émissions.

[8] Une zone de mémoire tampon.

Spécification 3.4 (*Raffinement de la spécification 3.3*) :

refinement *Canal_De_Communication_R*
refines *Canal_De_Communication*

constants *TailleBuff*
properties $TailleBuff \in \mathbb{N}_1$

variables *DansBuffer*, *AEnvoyer*
invariant $AEnvoyer \in \mathbb{N} \ \wedge \ DansBuffer \in 0..TailleBuff \ \wedge \ DansBuffer + AEnvoyer = TailleEnvoi$
variant *AEnvoyer*

initialisation $DansBuffer := 0 \ || \ AEnvoyer := 0$
operations
$Envoyer \mathrel{\hat{=}}$ **select** $AEnvoyer = 0 \ \wedge \ DansBuffer = 0$ **then** $AEnvoyer :\in \mathbb{N}_1$ **end ;**

$EnvoyerSuite \mathrel{\hat{=}}$ **select** $AEnvoyer > 0 \ \wedge \ DansBuffer < TailleBuff$ **then**
$AEnvoyer := AEnvoyer - 1 \ || \ DansBuffer := DansBuffer + 1$
end ;

$Traiter \mathrel{\hat{=}}$ **select** $DansBuffer > 0$ **then** $DansBuffer := DansBuffer - 1$ **end ;**

$Reset \mathrel{\hat{=}}$ **select** $AEnvoyer > 0 \ \vee \ DansBuffer > 0$ **then** $AEnvoyer := 0 \ || \ DansBuffer := 0$ **end**
end

Prenons l'exemple de l'opération *Envoyer*, et vérifions les obligations de preuve qui lui sont associées : renforcement de la garde (1) et correction du raffinement (2).

$$\begin{cases} (1) & \mathcal{B}_R \ \wedge \ L \ \wedge \ G_{Envoyer_R} \Rightarrow G_{Envoyer_A} \\ (2) & \mathcal{B}_R \ \wedge \ L \ \wedge \ G_{Envoyer_R} \Rightarrow [S_{Envoyer_R}]\langle S_{Envoyer_A}\rangle I_R \end{cases}$$

Obligation de preuve 3 : Renforcement de la garde (1)
La première formule se vérifie aisément car elle se ramène à :

$\mathcal{B}_R \ \wedge \ L \ \wedge \ AEnvoyer = 0 \ \wedge \ DansBuffer = 0 \Rightarrow TailleEnvoi = 0$

Ce qui est trivialement vrai, puisque l'invariant I_R du raffinement précise :
$$DansBuffer + AEnvoyer = TailleEnvoi$$
□

Obligation de preuve 4 : Correction du raffinement (2)
De la même manière, sous les hypothèses : $\mathcal{B}_R \ \wedge \ L \ \wedge \ (AEnvoyer = 0 \ \wedge \ DansBuffer = 0)$
la seconde formule se ramène à :

$$[S_{Envoyer_R}]\langle S_{Envoyer_A}\rangle \begin{pmatrix} AEnvoyer \in \mathbb{N} \ \wedge \ DansBuffer \in 0..TailleBuff \\ \wedge \ DansBuffer + AEnvoyer = TailleEnvoi \end{pmatrix}$$

$\equiv$ *Par application du $\mathcal{WP}$ sur $S_{Envoyer_A}$ et utilisation de l'hypothèse DansBuffer=0*

$$[S_{Envoyer_R}] \exists TailleEnvoi' \cdot \begin{pmatrix} TailleEnvoi' \in \mathbb{N}_1 \ \wedge \ AEnvoyer \in \mathbb{N} \ \wedge \ 0 \in 0..TailleBuff \\ \wedge \ 0 + AEnvoyer = TailleEnvoi' \end{pmatrix}$$

$\equiv$ *Par application du $\mathcal{WP}$ sur $S_{Envoyer_R}$ et comme $TailleBuff \geq 0$*

$$\forall AEnvoyer' \cdot \left(AEnvoyer' \in \mathbb{N}_1 \Rightarrow \exists TailleEnvoi' \cdot \begin{pmatrix} TailleEnvoi' \in \mathbb{N}_1 \ \wedge \ AEnvoyer' \in \mathbb{N} \\ \wedge \ AEnvoyer' = TailleEnvoi' \end{pmatrix} \right)$$

Ce qui est trivialement vrai, car $\mathbb{N}_1 \subseteq \mathbb{N}$.

□

3.7 Outils

Avant de conclure ce chapitre sur la méthode B, précisons qu'il existe des outils commerciaux supportant cette méthode. En particulier, l'*AtelierB* [Cle01] de ClearSy est un outil à vocation industrielle supportant les différentes phases de développement et offrant notamment un vérificateur syntaxique et sémantique, un générateur d'obligations de preuve, un prouveur automatique, un prouveur interactif et un ensemble de traducteurs vers des langages compilables (C, C++ ou ADA). L'outil *B-Toolkit* de la société B-Core, basée en Angleterre, fournit également de telles fonctionnalités.

Des alternatives moins coûteuses existent également. En particulier, notons l'existence de l'outil *B4free* [CA+04], également développé par ClearSy, qui est diffusé gratuitement aux universitaires et possesseurs de l'*AtelierB*. Ce logiciel intègre les mêmes fonctionnalités que l'*AtelierB*, à l'exception des traducteurs et de l'interface graphique. Cette dernière peut alors être remplacée par la *Balbulette* [AC03] (aussi appelé *Click'n Prove*) développé par J-R Abrial et D. Cansell.

Enfin, il existe également un certain nombre d'outils universitaires permettant de travailler avec des modèles B. Parmi eux, nous pouvons citer :

- le traducteur *CSP2B* [But00] de l'université de Southampton ;
- l'environnement de développement *ABTools* [Bou03] développé à l'université de Compiègne ;
- le parseur *jBTools* [VTH02] développé au LIFC (Besançon) ;
- la boîte à outils B [Châ01, Sto02] (BoB) du laboratoire LIG à Grenoble ;
- le générateur de test *BZ-TT* [ABC+02] développé au LIFC (Besançon) ;
- le générateur d'obligations de preuve *Barvey* [CDD+04, CDGR04] du LIFC-LORIA utilisant le prouveur *haRVey* [RD03] du LORIA ;
- l'animateur de spécification *ProB* [LB03] développé conjointement par les universités de Düsseldorf et Southampton.

Au terme de cette thèse, il nous faut également ajouter à cette liste d'outils le générateur de systèmes de transitions étiquetées *GénéSyst* [MPS04], décrit au chapitre 6.

3.8 Synthèse

Nous avons vu que la méthode B permet de couvrir l'intégralité du cycle de développement d'un système logiciel en partant d'un modèle formel et en utilisant d'éventuels niveaux de raffinement intermédiaires. Son processus de génération d'obligations de preuve permet de garantir que les invariants ne sont jamais violés et, par transitivité du raffinement, que l'implantation est un raffinement correct du modèle abstrait.

Nous avons également décrit l'approche événementielle, permettant d'intégrer le contrôle dans les modèles. L'utilisation d'événements au lieu d'opérations autorise alors l'introduction de nouveaux événements par raffinement, ce qui permet d'affiner le contrôle du système. Dans les parties suivantes, nous nous intéressons à mettre en évidence ce contrôle.

Deuxième partie

Contributions

Calcul et représentation des comportements d'un système B événementiel

4

Pictures aid understanding. A simple flowchart is easier to understand than the equivalent pro text. However, complex pictures are confusing. A large, spaghetti-like flowchart is harder to understand than a properly structured program text.

L. Lamport

Sommaire

4.1 Choix d'un formalisme de description des comportements . . . 69
- 4.1.1 Motivations . . . 69
- 4.1.2 Systèmes de transitions étiquetées symboliques . . . 70
- 4.1.3 Forme normalisée d'un événement B . . . 72

4.2 Extraction des comportements d'un système B . . . 73
- 4.2.1 Définition de l'espace d'états . . . 73
- 4.2.2 Construction de la relation de transition . . . 75

4.3 Lien entre un STES et le système B dont il est issu . . . 80
- 4.3.1 Sémantique d'un système B événementiel . . . 81
- 4.3.2 Sémantique d'un **STES** (Système de transitions étiquetées symbolique) . . . 81
- 4.3.3 Égalité des traces . . . 82

4.4 Critères de choix des états . . . 83
- 4.4.1 Exemple de mise en évidence d'une propriété . . . 83
- 4.4.2 Techniques de choix d'états . . . 84
- 4.4.3 Bilan . . . 88

4.5 Application au B classique . . . 88
- 4.5.1 Traduction d'une machine B en système B événementiel . . . 89
- 4.5.2 Sémantique de la transformation proposée . . . 90
- 4.5.3 Externalisation des paramètres . . . 91

4.6 Synthèse . . . 95

Classiquement, un système B événementiel met en jeu deux aspects orthogonaux : la modification des données et l'évolution du système (le contrôle). La séparation entre ces deux aspects est principalement conceptuelle. On peut, par exemple, programmer un analyseur syntaxique à partir d'un automate d'états finis en représentant la relation de transition dans une structure de données,

ou, à l'opposé, coder les transitions possibles par des GOTO. La représentation par des données fournit un cadre homogène pour raisonner, alors que la mise en évidence du contrôle peut permettre de mieux comprendre le fonctionnement d'un système.

L'approche B événementiel est principalement basée sur les données, ce qui, dans certains cas, oblige à introduire des variables de contrôle pour définir l'ordonnancement des événements [Nah01, Leb00]. En partant d'un système B événementiel, l'objectif est ici de proposer des visualisations de son contrôle par des diagrammes.

Dans ce chapitre, nous commençons par définir un formalisme de représentation avant de proposer une méthode de construction de l'ensemble des comportements d'un modèle B événementiel. Celle-ci consiste à calculer l'ensemble des transitions existantes sur un espace d'états, qui est donné par l'utilisateur. Cet espace doit cependant vérifier certaines propriétés de correction et de complétude, que nous caractérisons en section 4.2.1. Nous mettons ensuite l'accent sur le calcul des transitions entre les états, en section 4.2.2, qui se base sur un processus de génération d'obligations de preuve. Pour affiner les résultats obtenus, nous faisons l'hypothèse que le modèle B utilisé préserve son invariant (obligations de preuve décrites en section 3.7), ce qui nous permet d'établir, en section 4.3, une équivalence sémantique entre les systèmes de transitions étiquetées symboliques et l'ensemble des comportements des systèmes B événementiel dont ils sont issus.

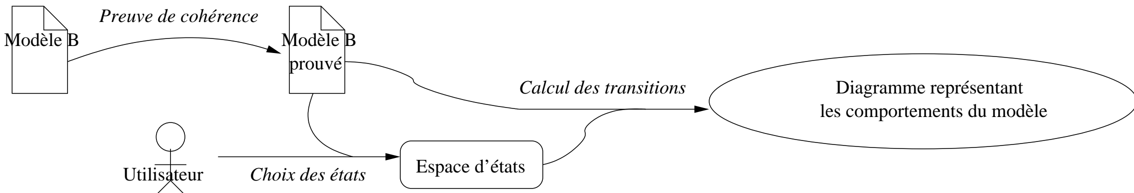


FIG. 4.1 – Étapes menant à la construction des comportements d'un modèle B

Nous terminons ce chapitre par deux discussions portant respectivement sur des méthodes de choix d'un espace d'états et sur l'application de l'approche proposée pour des modèles B classique. Enfin, nous illustrons nos propos par la spécification ci-dessous, déjà présentée dans le chapitre précédent (Spec. 3.3).

Spécification 4.1 (*Canal de communication présenté en section 3.6.2*) :

system *Canal_De_Communication*
variables *TailleEnvoi*
invariant $TailleEnvoi \in \mathbb{N}$
initialisation $TailleEnvoi := 0$
events
  $Envoyer \mathrel{\hat{=}}$ **select** $TailleEnvoi = 0$ **then** $TailleEnvoi :\in \mathbb{N}_1$ **end ;**
  $Traiter \mathrel{\hat{=}}$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := TailleEnvoi - 1$ **end ;**
  $Reset \mathrel{\hat{=}}$ **select** $TailleEnvoi > 0$ **then** $TailleEnvoi := 0$ **end**
end

4.1 Choix d'un formalisme de description des comportements

4.1.1 Motivations

Dans la section 2.2, nous avons décrit un certain nombre de formalismes permettant de représenter des ensembles de comportements. Notre objectif est d'aider à la compréhension et au développement de systèmes complexes en présentant une vue de leurs comportements. Ainsi, un diagramme trop grand pour être visualisé en entier sur un écran ou une feuille, ou bien contenant un entrelacement compliqué des transitions, n'est pas intéressant, car il n'est pas suffisamment intuitif pour être compris rapidement.

Nous proposons d'utiliser un formalisme symbolique, tel que les machines abstraites à états ou les diagrammes d'états-transitions, ce qui permet de représenter un ensemble potentiellement infini de valeurs par un nombre fini d'états. Dans une telle représentation, le choix des états permet d'exhiber différentes vues d'un système et de mettre en évidence certaines de ses propriétés. Par exemple, la figure 4.2 n'a qu'un seul état et n'apporte rien à la compréhension, tandis qu'il est possible de voir, sur la figure 4.3, que deux occurrences de l'événement *Envoyer* ne peuvent pas se suivre sans être séparées au moins par une occurrence de *Reset* ou de *Traiter*. Sur nos représentations, l'événement *Init* correspond à l'initialisation du modèle B et l'état q_{Init} est l'état des variables avant l'exécution de la clause d'initialisation du système. Nous avons choisi de mettre cet état en évidence, car il est possible que l'initialisation atteigne plusieurs états. Ainsi, les transitions *Init* ont une origine commune.

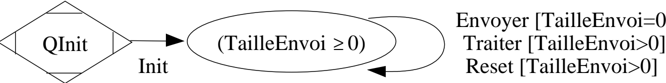


FIG. 4.2 – Représentation symbolique à un seul état des comportements du canal de communication

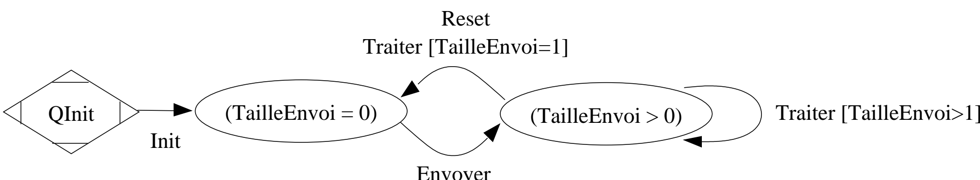


FIG. 4.3 – Seconde représentation avec un autre espace d'états

Dans la section suivante, nous nous basons sur le formalisme de systèmes de transitions étiquetées défini sur les systèmes B événementiel par D. Bert et F. Cave [BC00] que nous étendons pour rajouter des conditions de franchissement.

4.1.2 Systèmes de transitions étiquetées symboliques

Classiquement, les transitions d'un système symbolique sont étiquetées par le nom de l'événement permettant de les franchir. Certains formalismes proposent également de préciser une condition de franchissement des transitions. Dans notre approche, nous proposons d'affiner cette garde en la scindant en deux parties : la condition de *déclenchabilité* et la condition d'*atteignabilité*. Cette représentation a pour objectif d'améliorer la compréhension que l'on a des comportements du système en décrivant plus finement, dans le cas des transitions partiellement franchissables, les conditions sur les états de départs et d'arrivée. Intuitivement, un événement ev a une même condition de déclenchabilité depuis chaque état E. Si l'événement ev peut mener à plusieurs états, c'est la condition d'atteignabilité qui va caractériser depuis quelles valuations ces états peuvent être atteints par ev.

Dans l'étiquette $[D]$ $[A]$ ev d'une transition, D et A sont des prédicats logiques et ev est un événement du système B associé, tels que :

- ev est le **nom d'un événement** du système B ;
- D est la **condition de déclenchabilité** : condition caractérisant les valuations de l'état de départ permettant de déclencher l'événement ev ;
- A est la **condition d'atteignabilité** : condition caractérisant, parmi les valuations de l'état de départ qui permettent de déclencher ev, celles permettant à ev d'atteindre l'état d'arrivée.

Terminologie. *La conjonction des conditions de déclenchabilité et d'atteignabilité correspond à la condition de franchissement des StateCharts. Par abus de langage ces deux conditions sont aussi appelées les* ***conditions de franchissement*** *d'une transition.*

Dans les StateCharts, ces deux conditions existent implicitement. En effet, une transition partant du bord d'un état peut être déclenchée depuis tout l'état, tandis qu'une transition partant depuis l'intérieur de l'état ne peut être déclenchée que depuis une sous-partie de celui-ci [Har87, page 241, figure 16.b]. Ainsi, la garde d'une transition partant du bord d'un état correspond implicitement à sa condition d'atteignabilité. Sinon, elle correspond à la conjonction des conditions de déclenchabilité et d'atteignabilité [Har87, page 253, figure 33].

Nous définissons les systèmes de transitions étiquetées symboliques de la manière suivante :

Définition 10 (Système de transitions étiquetées symbolique) *Un système de transitions étiquetées symbolique est un graphe orienté ayant une unique origine, des étiquettes sur les transitions et des états définis par des prédicats. Il peut être vu comme un septuplet* $(\mathbb{V}, \mathbb{E}, \mathbb{Q}, \mathsf{q}_{Init}, \mathcal{D}ef, \mathbb{L}, \mathbb{R})$ *tel que :*

- $\mathbb{V}$ *est un ensemble de variables,*
- $\mathbb{E}$ *est un ensemble de noms d'événements,*
- $\mathbb{Q}$ *est un ensemble de noms d'états,*
- q_{Init} *est l'état initial* ($\mathsf{q}_{Init} \in \mathbb{Q}$)*,*
- $\mathcal{D}ef$ *associe un prédicat de définition, portant sur* $\mathbb{V}$*, à chaque nom d'état* ($\mathcal{D}ef \in \mathbb{Q} \rightarrow \mathbb{P}$)*,*
- $\mathbb{L}$ *est un ensemble d'étiquettes* ($\mathbb{L} \subseteq \mathbb{P} \times \mathbb{P} \times \mathbb{E}$)*,*
- $\mathbb{R}$ *est une relation de transition* ($\mathbb{R} \subseteq \mathbb{Q} \times \mathbb{L} \times \mathbb{Q}$)*.*

où $\mathbb{P}$ *est l'ensemble des prédicats portant sur l'ensemble de variables* $\mathbb{V}$*.*

Terminologie. *Afin d'alléger le texte, nous pourrons désigner un système de transitions étiquetées symbolique en parlant simplement de systèmes de transitions ou en utilisant l'acronyme* ***STES****.*

De plus, nous notons $(E, (D, A, ev), F)$*, dans le texte, la transition de l'état E vers l'état F par l'événement ev sous la condition de déclenchabilité D et la condition d'atteignabilité A. Dans les représentations graphiques, les étiquettes sont notées sous la forme* $[D]$ $[A]$ *ev.*

La figure 4.4 est une description des comportements de l'exemple du canal de communication, où les conditions de déclenchabilité et d'atteignabilité des transitions sont explicitées. Cette notation permet notamment de mettre en évidence que l'événement *Envoyer* n'est déclenchable depuis l'état **E1** que si *TailleEnvoi* $= 0$. De plus, le non déterminisme interne de cet événement est mis en évidence par le fait qu'il peut mener dans chacun des deux états **E1** et **E2**, avec la même condition d'atteignabilité true. À l'inverse, l'événement *Traiter* est toujours déclenchable depuis l'état **E2**, mais son atteignabilité est conditionnée. Celle-ci caractérise les valuations à partir desquelles *Traiter* mène dans l'état **E1** ou bien dans l'état **E2**.

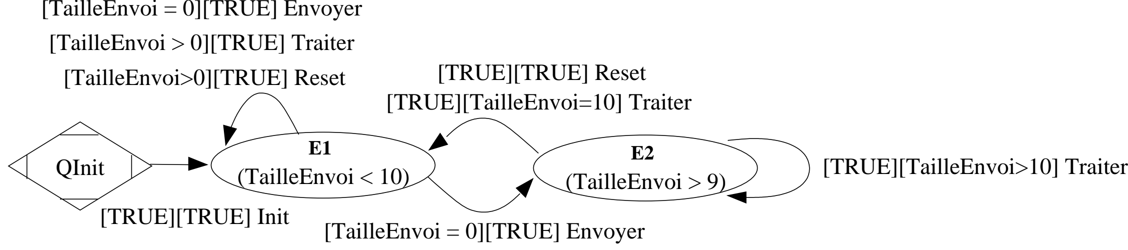


Fig. 4.4 – Comportements du canal de communication avec conditions de déclenchabilité et d'atteignabilité.

Avant de décrire comment construire un système de transitions à partir d'un modèle B, nous introduisons quelques résultats relatifs au B événementiel.

4.1.3 Forme normalisée d'un événement B

Dans [AM98], la garde logique d'un événement ev, par opposition à sa garde syntaxique, est décrite comme étant sa faisabilité, c'est-à-dire l'ensemble des valuations des variables pour lesquelles il existe une valuation après ev (pour lesquelles ev peut calculer un état d'après). Dans cette section, nous introduisons une forme normalisée des événements B mettant en évidence, de manière syntaxique, cette condition de faisabilité. Pour ce faire, nous nous plaçons dans le cas particulier du B événementiel, qui garantit syntaxiquement la terminaison.

Définition 11 (Faisabilité d'une substitution) *La faisabilité d'une substitution généralisée S (notée $\mathsf{fis}(S)$) se définit de la manière suivante :*

$$\mathsf{fis}(S) \equiv \langle S \rangle \mathsf{btrue}$$

Notons également l'équivalence suivante, que nous pouvons être amenés à utiliser :

$$\mathsf{fis}(S) \equiv \exists x' \cdot (\mathsf{prd}_x(S))$$

Si $ev \mathrel{\hat{=}} S$ est un événement, alors nous notons $\mathsf{Garde}(ev)$ sa garde logique et $\mathsf{Action}(ev)$ son corps. Conformément à [AM98], nous définissons $\mathsf{Garde}(ev) = \mathsf{fis}(S)$. La définition de l'action, quant à elle, est définie par $\mathsf{Action}(ev) = S$. On peut alors définir la forme normalisée des événements comme suit :

Définition 12 (Forme normalisée d'un événement) *Tout événement $ev \mathrel{\hat{=}} S$ peut se mettre sous une forme normalisée $\mathcal{F}(ev)$:*

$$\mathcal{F}(ev) \mathrel{\hat{=}} \mathsf{Garde}(ev) \Longrightarrow \mathsf{Action}(ev)$$

où $\mathsf{Garde}(ev) = \mathsf{fis}(S)$ et $\mathsf{Action}(ev) = S$.

L'intérêt de cette forme est que la garde syntaxique est aussi la garde logique. Il est aisé de vérifier que tout événement $ev \mathrel{\hat{=}} S$ est équivalent à sa forme normalisée $\mathcal{F}(ev)$ en montrant que, pour tout prédicat R, on a $[S]R \Leftrightarrow [\mathcal{F}(ev)]R$. Cette démonstration est décrite en annexe B.1.

Exemple 4.1 (*Forme normalisée d'un événement*) :

Soit l'événement suivant :

$$ev \mathrel{\hat{=}} (P_1 \Longrightarrow x := E_1) \parallel (P_2 \Longrightarrow x := E_2)$$

Sa faisabilité est :

$$\mathsf{fis}(ev) \equiv \langle P_1 \Longrightarrow x := E_1 \rangle \mathsf{btrue} \ \vee \ \langle P_2 \Longrightarrow x := E_2 \rangle \mathsf{btrue} \equiv P_1 \ \vee \ P_2$$

ev peut donc être mis sous la forme normalisée suivante :

$$\mathcal{F}(ev) = (P_1 \ \vee \ P_2) \Longrightarrow ((P_1 \Longrightarrow x := E_1) \parallel (P_2 \Longrightarrow x := E_2))$$

4.2 Extraction des comportements d'un système B

Dans cette partie, nous décrivons notre méthode de construction d'un système de transitions représentant l'ensemble des comportements d'un système B événementiel. Le choix de l'espace d'états est très important, car c'est lui qui conditionne la vue que l'on a du système. C'est pourquoi, nous avons choisi de laisser ce choix à l'utilisateur, qui peut vouloir mettre en évidence certaines propriétés du système. Toutefois nous donnons des heuristiques possibles pour effectuer ce choix en section 4.4. Construire la relation de transition consiste alors à calculer, pour chaque couple d'états et chaque événement, si la transition est franchissable et sous quelle condition. Nous faisons ici l'hypothèse que les obligations de preuve garantissant la préservation de l'invariant du système ont été prouvées (section 3.6.3).

Étant donné un espace d'états, nous voulons construire un système de transition dont l'ensemble des chemins soit une sur-approximation aussi petite que possible de l'ensemble des comportements du modèle. Pour ce faire, nous définissons certains critères de qualité comme le fait que les états soient non-vides ou que toute transition représentée soit au moins partiellement franchissable.

4.2.1 Définition de l'espace d'états

Dans notre approche, le spécifieur doit fournir un ensemble de prédicats caractérisant les états désirés. Dans cette section, nous introduisons d'abord les propriétés qui doivent être vérifiées par cet ensemble, puis nous décrivons la manière de construire, à partir de ce dernier, l'espace d'états utilisé pour le calcul de la relation de transition.

4.2.1.1 Propriétés d'un espace d'états

Un espace d'états est caractérisé par un ensemble de noms d'états et une fonction $\mathcal{D}ef$ de définition des états (Définition 10). Dans notre approche, nous imposons que l'espace d'états vérifie certaines propriétés vis-à-vis de l'invariant du système.

Tout d'abord, il est évident que, pour pouvoir construire l'ensemble des comportements d'un système, toutes ses configurations atteignables doivent nécessairement être représentées. Cependant, la caractérisation de l'ensemble des configurations atteignables se ramène au problème de l'arrêt. C'est pourquoi, nous simplifions le problème en considérant l'invariant B du système, qui est un sur-ensemble des états atteignables. Ainsi, s'il est prouvé que les opérations préservent l'invariant, alors il est suffisant d'utiliser un espace d'états complet par rapport à l'invariant. D'où la condition suivante :

Condition 1 (Complétude de l'espace d'états)
L'espace d'états caractérisé par l'ensemble de noms d'états $\{E_1, \dots, E_n\}$ *et la fonction de définition* $\mathcal{D}ef$ *est dit* ***complet*** *vis-à-vis d'un invariant* I *si et seulement si :*

$$I \Rightarrow \left(\bigvee_{i=1}^{n} \mathcal{D}ef(E_i) \right)$$

De plus, nous imposons que l'espace d'états soit correct vis-à-vis de l'invariant du système. En

effet, la garde des événements est définie sous hypothèse de l'invariant. Il est donc possible que certains événements soient déclenchables depuis des valeurs ne vérifiant pas l'invariant. Cependant, nous avons fait l'hypothèse que l'utilisateur a établi que l'invariant est préservé par les événements. Il s'ensuit que les valuations, à partir desquelles ces transitions sont franchissables, ne sont pas atteignables. Si celles-ci étaient représentées dans l'espace d'états alors nous afficherions des transitions non-atteignables dans le système de transitions. Afin de limiter la présence de telles transitions, il suffit donc que l'espace d'états soit correct par rapport à l'invariant. D'où la condition suivante :

Condition 2 (Correction de l'espace d'états)
L'espace d'états caractérisé par l'ensemble de noms d'états $\{E_1, \ldots, E_n\}$ *et la fonction de définition* $\mathcal{D}ef$ *est dit* ***corrects*** *vis-à-vis d'un invariant* I *si et seulement si :*

$$\left(\bigvee_{i=1}^{n} \mathcal{D}ef(E_i)\right) \Rightarrow I$$

Enfin, notons qu'il n'est théoriquement pas nécessaire que les états soient non-vides, c'est-à-dire que, sous l'hypothèse de l'invariant, leur prédicat de définition ne soit pas équivalent à bfalse. En effet, l'algorithme que nous proposons dans les sections suivantes ne devrait pas permettre d'atteindre des états vides. Cependant, comme la méthode proposée se base sur la vérification d'obligations de preuve, elle est donc sensible au risque de défaut de preuve. C'est pourquoi, il est intéressant d'établir la condition suivante pour chaque état :

Condition 3 (États non-vides)
Un état E *défini par le prédicat de définition* $\mathcal{D}ef(E)$ *est dit* ***non-vide*** *vis-à-vis d'un invariant* I *si et seulement si :*

$$\exists x \cdot (\mathcal{D}ef(E) \ \wedge \ I)$$

où x *désigne les variables du système.*

4.2.1.2 Construction d'un espace d'états

Concrètement, pour construire un espace d'états, il suffit que l'utilisateur fournisse une liste de prédicats $P_1, \ldots, P_n$ qui soient corrects, complets et non-vides par rapport à l'invariant I du système. Chacun de ces prédicats P_i est alors considéré comme le prédicat de définition d'un état, dont le nom E_i peut être construit de manière automatique. L'état initial q_{Init}, quant à lui, est construit automatiquement. Cet état caractérise l'ensemble des valuations possibles du système lorsque les variables ne sont pas encore initialisées. Il n'est donc jamais atteignable et l'invariant n'y est pas établi, mais l'initialisation y est toujours déclenchable. Nous choisissons donc d'attribuer le prédicat btrue à q_{Init} :

$$\left\{\begin{array}{rl} & \mathbb{Q} = \{E_1, \ldots, E_n\} \cup \{\mathsf{q}_{Init}\} \\ \wedge & \bigwedge_{i=1}^{n}(\mathcal{D}ef(E_i) = P_i) \\ \wedge & \mathcal{D}ef(\mathsf{q}_{Init}) = \mathsf{btrue} \end{array}\right.$$

4.2.2 Construction de la relation de transition

Avant de décrire l'algorithme général, nous devons introduire la notion de validité d'une transition.

4.2.2.1 Validité d'une transition

Intuitivement, pour qu'une transition soit valide, il suffit qu'elle soit partiellement franchissable. Cette notion est indépendante de l'atteignabilité de l'état d'origine de la transition. Par contre, elle nécessite de caractériser finement les propriétés qui caractérisent les conditions de déclenchabilité et d'atteignabilité d'une transition.

Si E et F sont deux noms d'états dont $\mathcal{D}ef(E)$ et $\mathcal{D}ef(F)$ sont les prédicats de définition, alors

$$\{x \mid \mathcal{D}ef(E) \ \wedge \ \langle ev \rangle \mathcal{D}ef(F)\}$$

caractérise l'ensemble des valuations de E depuis lesquelles l'événement ev peut atteindre l'état F. Par passage à la forme normalisée des événements (Définition 12) et décomposition selon le $\mathcal{WP}$, ce prédicat peut se réécrire de la manière suivante :

$$\mathcal{D}ef(E) \ \wedge \ \mathsf{Garde}(ev) \ \wedge \ \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$$

Pour caractériser les propriétés des conditions de déclenchabilité et d'atteignabilité d'une transition, nous proposons de décomposer ce prédicat comme suit :

Définition 13 (Caractérisation des conditions) *Soit $(E, (D, A, ev), F)$ une transition. Alors les conditions D et A doivent vérifier les propriétés suivantes :*

$$\mathcal{D}ef(E) \Rightarrow (D \Leftrightarrow \mathsf{Garde}(ev)) \quad (13.1)$$

$$\mathcal{D}ef(E) \ \wedge \ D \Rightarrow (A \Leftrightarrow \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)) \quad (13.2)$$

Cette définition permet de dissocier les conditions de déclenchabilité et d'atteignabilité. De cette manière, nous pouvons réduire la condition de déclenchabilité à btrue si ev est toujours déclenchable depuis E et à bfalse s'il ne l'est jamais. De la même manière, pour que la condition d'atteignabilité soit réductible à btrue, il suffit que toutes les valuations de E qui vérifient D permettent de mener à F par ev.

Terminologie. *Une condition de déclenchabilité ou d'atteignabilité est* ***réductible à*** *un prédicat P (Classiquement* btrue *ou* bfalse*) si celui-ci vérifie l'équation caractéristique associée à la condition (Respectivement les équations* 13.1 *et* 13.2*, définition 13).*

Nous pouvons en déduire la définition suivante de la validité d'une transition :

Définition 14 (Validité d'une transition) *Une transition $(E, (D, A, ev), F)$, où D et A vérifient la définition 13, est dite* ***valide*** *si et seulement si :*

$$\exists x \cdot (\mathcal{D}ef(E) \ \wedge \ D \ \wedge \ A)$$

avec x les variables du système.

Terminologie. *On note Valide(t) la validité de la transition t.*

4.2.2.2 Algorithme de construction de la relation de transition

Pour construire l'ensemble des transitions d'un **STES**, il suffit de calculer, pour chaque couple d'états (E, F) et chaque événement ev, s'il existe une transition valide allant de E à F par ev. Pour ce faire, nous avons besoin de calculer les conditions de franchissement d'une transition. Ce point sera détaillé dans la section suivante. Afin de limiter l'introduction de transitions non atteignables, nous proposons de construire la relation de transition par induction sur les états de $\mathbb{Q}$ qui sont atteignables à partir de l'état initial. L'ensemble $\mathbb{Q}_{Nouv}$ caractérise l'ensemble des états atteignables pour lesquels on n'a pas encore calculé l'ensemble des transitions. À l'inverse, la variable $\mathbb{Q}_{Traité}$ caractérise l'ensemble des états atteints déjà traités.

Notons que la notion d'atteignabilité utilisée ici est faible, car, pour qu'un état de $\mathbb{Q}$ soit dit atteignable, il suffit qu'il existe une séquence de transitions partant de l'état initial et menant à cet état. Il suffit que alors chaque transition soit valide, pas la séquence.

Étant donné les ensembles d'états $\mathbb{Q}$ et d'événements $\mathsf{Interface}(S)$, l'algorithme suivant, permet de calculer la relation de transition $\mathbb{R}$. À l'état initial, le système se trouve dans l'état q_{Init} et seule l'initialisation peut s'exécuter. Comme sa déclenchabilité est toujours vraie depuis cet état, c'est la condition d'atteignabilité qui permet de déterminer si la transition vers un état est valide ou pas. Enfin, pour simplifier les notations nous accédons aux éléments de l'ensemble $\mathbb{Q}_{Nouv}$ par la substitution non-déterministe $:\in$.

Algorithme 1 (*Calcul de la relation de transition*) :

Paramètres en entrée : $\mathbb{Q}$ et $\mathsf{Interface}(S)$

Avant initialisation du système : $\mathbb{Q}_{Trait\acute{e}} = \emptyset \parallel \mathbb{Q}_{Nouv} = \emptyset \parallel \mathbb{R} = \emptyset$.

Transitions depuis q_{Init} (Recherche des états atteignables par l'initialisation) :

```
Pour chaque F tel que F ∈ ℚ − {q_Init}
    Trouver la condition A de la transition (q_Init, (btrue, A, Init), F) ; /* Section 4.2.2.4 */
    Si Valide(q_Init, (btrue, A, Init), F) alors :
        ℚ_Nouv := ℚ_Nouv ∪ {F}  || /* F est ajouté, car il ne peut pas avoir été traité */
        ℝ := ℝ ∪ {(q_Init, (btrue, A, Init), F)}
    /* Sinon il n'y a pas de transition de q_Init vers F par Init */
```

Construction de $\mathbb{R}$ par induction sur $\mathbb{Q}_{Trait\acute{e}}$:

```
Tant que ℚ_Nouv ≠ ∅
    E :∈ ℚ_Nouv ;  /* On choisit un état E atteint mais non traité */
    Pour chaque ev et F tels que ev ∈ Interface(S) ∧ F ∈ ℚ − {q_Init}
        Trouver les conditions D et A de la transition (E, (D, A, ev), F) ; /* Section 4.2.2.4 */
        Si Valide(E, (D, A, ev), F) alors :
            ℚ_Nouv := (ℚ_Nouv ∪ {F}) − ℚ_Traité || /* F est ajouté s'il n'a pas déjà été traité */
            ℝ := ℝ ∪ {(E, (D, A, ev), F)}
        /* Sinon il n'y a pas de transition de E vers F par ev */
    ℚ_Traité := ℚ_Traité ∪ {E} ;
    ℚ_Nouv := ℚ_Nouv − {E}
```

Résultat : $\mathbb{R}$

Le point délicat de cet algorithme consiste à trouver des conditions de déclenchabilité et d'atteignabilité, garantissant la validité des transitions.

4.2.2.3 Calcul des conditions de franchissement d'une transition

Il est possible de construire syntaxiquement les conditions de franchissement comme suit :

$$\begin{cases} D \mathrel{\hat{=}} \mathsf{Garde}(ev) \\ A \mathrel{\hat{=}} \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F) \end{cases}$$

Bien que cette solution soit correcte (Définition 13), elle peut être améliorée pour caractériser notamment les transitions infranchissables. Dans cette section, nous proposons une méthode de calcul de ces conditions qui se base sur la résolution d'obligations de preuve.

En nous inspirant des *maybe transitions* introduites par D. Cansell, D. Méry et S. Merz [CMM00a] (section 2.3.2.3), nous proposons donc de nous intéresser aux trois formes de conditions suivantes : btrue, bfalse et conditionné (*maybe*). Ce choix permet de construire un algorithme de recherche des conditions de déclenchabilité et d'atteignabilité qui se base sur la preuve, tout en mettant en évidence les cas où les conditions sont toujours vraies ou fausses.

Dans le tableau 4.1, nous proposons des obligations de preuve permettant de déterminer la valeur de la condition de déclenchabilité. La formule (1) permet d'établir que toutes les valuations

de l'état E vérifient la garde de l'événement ev et donc que la condition de déclenchabilité est btrue. À l'inverse, la formule (2) permet d'établir qu'aucune des valeurs de l'état E ne vérifie la garde de l'événement ev et donc que la condition de déclenchabilité vaut bfalse. Enfin, la formule (3) consiste à vérifier que la garde de ev est établie par au moins une valeur de E et qu'une autre valeur de E ne vérifie pas cette garde.

Formules à vérifier		Valeur de D si la formule associée est établie	
(1)	$\forall x \cdot (\mathcal{D}ef(E) \Rightarrow \mathsf{Garde}(ev))$	$D \mathrel{\hat{=}} \mathsf{btrue}$	*Toujours déclenchable*
(2)	$\forall x \cdot (\mathcal{D}ef(E) \Rightarrow \neg \mathsf{Garde}(ev))$	$D \mathrel{\hat{=}} \mathsf{bfalse}$	*Non déclenchable*
(3)	$\exists x \cdot (\mathcal{D}ef(E) \wedge \mathsf{Garde}(ev)) \wedge$ $\exists x \cdot (\mathcal{D}ef(E) \wedge \neg \mathsf{Garde}(ev))$	$D \mathrel{\hat{=}} \mathsf{Garde}(ev)$	*Partiellement déclenchable*

Tab. 4.1 – Obligations de preuve de déclenchabilité

De la même manière, nous proposons, dans le tableau 4.2, des obligations de preuve permettant de déterminer les conditions d'atteignabilité. Dans ces obligations de preuve on s'intéresse d'une part aux configurations de l'état E qui vérifient la garde de l'événement ($\mathcal{D}ef(E) \wedge D$) et d'autre part aux configurations qui permettent d'aller dans F par ev ($\langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$). Les trois obligations de preuve permettent donc de déterminer si ces deux ensembles sont inclus, disjoints ou avec une intersection non nulle.

La formule (4) permet d'établir que toutes les valeurs de l'état E vérifiant la garde de l'événement ev permettent d'atteindre F par l'exécution de l'événement ev et donc que la condition d'atteignabilité vaut btrue. La formule (5) permet d'établir qu'aucune des valeurs de l'état E qui vérifient la garde de l'événement ev ne permet d'atteindre F par l'exécution de l'événement ev et donc que la condition d'atteignabilité vaut bfalse. Enfin, la formule (6) consiste à vérifier qu'au moins l'une des valeurs de E vérifiant la garde de ev permet d'atteindre F par ev et qu'une autre valeur de E vérifiant la garde de ev ne permet pas d'atteindre F par ev.

Formules à vérifier		Valeur de A si la formule associée est établie	
(4)	$\forall x \cdot (\mathcal{D}ef(E) \wedge D \Rightarrow \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F))$	$A \mathrel{\hat{=}} \mathsf{btrue}$	*Toujours atteignable*
(5)	$\forall x \cdot (\mathcal{D}ef(E) \wedge D \Rightarrow \neg \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F))$	$A \mathrel{\hat{=}} \mathsf{bfalse}$	*Non atteignable*
(6)	$\exists x \cdot (\mathcal{D}ef(E) \wedge D \wedge \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)) \wedge$ $\exists x \cdot (\mathcal{D}ef(E) \wedge D \wedge \neg \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F))$	$A \mathrel{\hat{=}} \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$	*Partiellement atteignable*

Tab. 4.2 – Obligations de preuve d'atteignabilité

Notons que, si un état F est vide, alors il ne peut pas être atteint si E est non-vide. En effet, dans ce cas, son prédicat de définition $\mathcal{D}ef(F)$ est réductible à bfalse et $\langle \mathsf{Action}(ev)\rangle \mathcal{D}ef(F)$ est donc toujours faux. La formule (5) est trivialement vérifiée et la formule (4) ne l'est jamais. Ainsi, en utilisant l'algorithme 1, ni l'initialisation, ni les transitions ne peuvent atteindre d'état vide.

4.2.2.4 Problème de l'interaction avec un prouveur

Pour prouver les formules présentées dans les tableaux 4.1 et 4.2, il est possible d'utiliser un logiciel de preuve soit interactif, soit automatique. Dans le premier cas, c'est l'utilisateur qui dirige le processus de preuve. C'est lui qui choisit si la preuve semble fausse ou pas et donc s'il faut corriger la spécification ou s'il faut persévérer. À l'inverse, dans cas d'un démonstrateur automatique, il y a un risque de défaut de preuve. Il est donc nécessaire de prévoir un quatrième cas, *Défaut de preuve*, pour chacune des deux conditions (Tableau 4.3). Celui-ci caractérise alors le fait que l'on ne sache rien dire sur une condition : ni toujours vraie, ni toujours fausse, ni partiellement vraie. Les conditions de franchissement qui se trouvent dans ce cas, sont définies par le tableau 4.3 et sont représentées, sur le système de transition par un $[X]$.

Formules à vérifier		Valeur de la condition si la formule associée est établie	
$(3')$	———	$D \mathrel{\hat=} \mathsf{Garde}(ev)$	*Défaut de preuve*
$(6')$	———	$A \mathrel{\hat=} \langle \mathsf{Action}(ev)\rangle \mathcal{D}ef(F)$	*Défaut de preuve*

TAB. 4.3 – Caractérisation des cas de défaut de preuve des conditions

De plus, en cas de défaut de preuve, il devient possible d'atteindre un état vide. Or, nos obligations de preuve (1) et (4) permettent à tout événement de se déclencher depuis un état vide et d'atteindre tous les états du système. Ainsi, de nombreuses transitions inexistantes sont constructibles, si un état vide est atteint par défaut de preuve. Afin d'éviter cette surcharge, nous proposons de favoriser la non introduction des transitions. En effet, les obligations de preuve (2) et (5) de non déclenchabilité et de non atteignabilité sont également vraies pour tout événement et tout état d'arrivée depuis un état vide. L'heuristique que nous utilisons dans l'algorithme 2 consiste alors à vérifier les cas (2) et (5) d'absence de transition avant les autres cas. Ainsi, lorsqu'un état vide est atteint, on limite le nombre de transitions qui en partent.

Définition 15 (Minimalité d'un STES) *Un **STES** est dit **minimal** si et seulement si aucune de ses conditions de franchissement n'est en défaut de preuve (cas* $(3')$ *et* $(6')$ *du tableau 4.3).*

Notons que, contrairement à la terminologie classique, comme l'espace d'états est figé, nous définissons la minimalité par rapport au nombre de défauts de preuve au lieu du nombre d'états.

Algorithme 2 (*Trouver D et A*) :

Paramètres en entrée : l'événement ev et deux états E et F.

Déterminer la condition de déclenchabilité D d'un événement ev depuis un état E :

1. si la formule (2) est prouvée alors $D := \mathsf{bfalse}$; /* *FIN. Transition non franchissable* */
2. sinon, si la formule (1) est prouvée alors $D := \mathsf{btrue}$;
3. sinon, si la formule (3) est prouvée alors $D := \mathsf{Garde}(ev)$;

3'. sinon, c'est un défaut de preuve. $D := \mathsf{Garde}(ev)$ par défaut.

Déterminer la condition d'atteignabilité A d'un état F par ev depuis E :

4. si la formule (5) est prouvée alors $A := \mathsf{bfalse}$; /* *FIN. Transition non franchissable* */
5. sinon, si la formule (4) est prouvée alors $A := \mathsf{btrue}$;
6. sinon, si la formule (6) est prouvée alors $A := \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$;

6'. sinon, c'est un défaut de preuve. $A := \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$ par défaut.

Résultat : les conditions D et A.

Pour finir, le tableau 4.4 résume le nombre d'obligations de preuve nécessaires pour construire un système de transition représentant l'ensemble des comportements d'un système B événementiel.

Initialisation	*3 OP par état*
Déclenchabilité d'une transition	*3 OP par événement et par couple d'états*
Atteignabilité d'une transition	*3 OP par événement et par couple d'états*

TAB. 4.4 – Coût maximum, en nombre d'obligations de preuve (OP), de la génération d'un système de transitions associé à système B événementiel

Dans la section suivante, nous nous intéressons à établir la sémantique d'un système de transitions étiquetées symbolique en termes de ses chemins et nous établissons son équivalence sémantique avec les traces d'exécution du système B associé.

4.3 Lien entre un STES et le système B dont il est issu

Pour conclure sur la construction d'un système de transitions, nous rappelons la sémantique d'un système B événementiel, qui est définie en termes de ses traces d'exécution, et nous définissons la sémantique d'un **STES** en termes de ses chemins (Définition 5, section 2.2.2). Enfin, nous prouvons l'égalité entre ces deux ensembles, concluant ainsi que la méthode proposée permet de générer un **STES** représentant **exactement** les comportements du système B.

4.3.1 Sémantique d'un système B événementiel

Le B événementiel est initialement défini par J-R. Abrial dans [Abr96a] comme étant un système encapsulant ses variables et dans lequel chaque événement est atomique et visible par un observateur extérieur. En accord avec cette définition, la notion de raffinement B événementiel, ainsi que les utilisations qui en sont faites [BF03, BJK00], nous choisissons ici une vue *event-based* qui permet d'observer un système sous la forme de ses événements déclenchables.

Définition 16 (Trace d'un système B événementiel) *Une suite finie d'occurrences d'événements* $Init \ ; \ oc_1 \ ; \ oc_2 \ ; \ \ldots \ ; \ oc_n$ *est une trace d'un système* S *si et seulement si :*
- $Init$ *est l'initialisation de* S *;*
- $oc_1 \ ; \ oc_2 \ ; \ \ldots \ ; \ oc_n$ *sont des occurrences d'événements de* S *;*
- $\mathsf{fis}(Init \ ; \ oc_1 \ ; \ oc_2 \ ; \ \ldots \ ; \ oc_n) \Leftrightarrow \mathsf{btrue}$.

Terminologie. *On désigne par* $\mathsf{Traces}(S)$ *l'ensemble des traces d'un système* S.

En utilisant les formes normalisées et la définition de la faisabilité, on obtient le lemme suivant, dont la démonstration est décrite en annexe B.2 :

Lemme 1 (Caractérisation d'une trace) *Soit* S *un système B événementiel,* $Init$ *l'initialisation de* S *et* oc_1 *à* oc_n *des occurrences d'événements de* S*, alors :*

$$Init \ ; \ oc_1 \ ; \ \ldots \ ; \ oc_n \in \mathsf{Traces}(S) \Leftrightarrow$$
$$\exists x_1, \ldots, x_{n+1} \cdot ([x' := x_1]\mathsf{prd}_x(Init) \wedge \bigwedge_{i=1}^{n}([x := x_i]\mathsf{Garde}(oc_i) \wedge [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(\mathsf{Action}(oc_i))))$$

avec x_i *une valuation des variables du système* S.

4.3.2 Sémantique d'un STES (Système de transitions étiquetées symbolique)

Pour définir les chemins d'un système de transitions, nous avons besoin de pouvoir caractériser les valuations avant et après le franchissement d'une transition. Intuitivement, une transition t est valide si et seulement si il existe deux valuations x et x' des variables du système, telles que le franchissement de t depuis x puisse mener dans x'. Conformément à la définition de la validité d'une transition (Définition 14), nous définissons :

Définition 17 (Franchissement d'une transition) *Une transition* $(E, (D, A, ev), F)$ *est* ***franchissable*** *depuis une valuation* x *de* E *jusqu'à une valuation* x' *de* F *si et seulement si :*

$$\mathcal{D}ef(E) \ \wedge \ \mathsf{Garde}(ev) \ \wedge \ \mathsf{prd}_x(\mathsf{Action}(ev)) \ \wedge \ [x := x']\mathcal{D}ef(F))$$

On note $((E, x) \rightsquigarrow^{(D,A,ev)}\rightsquigarrow (F, x'))$ le franchissement depuis la valeur x de E vers la valeur x' de F. On a donc : $Valide(E, (D, A, ev), F) \Leftrightarrow (\exists (x, x') \cdot ((E, x) \rightsquigarrow^{(D,A,ev)}\rightsquigarrow (F, x')))$.

Informellement, un chemin d'un **STES** est une séquence d'occurrences d'événements commençant par l'initialisation et correspondant à des franchissements de transitions valides, tels que la valeur d'arrivée des variables après un franchissement soit la valuation de départ du franchissement

suivant. On note $\mathsf{Chemins}(T)$ l'ensemble des chemins de T. Plus formellement on a :

Définition 18 (Chemins d'un système de transitions étiquetées) *Toute suite d'occurrences d'événements $Init; oc_1; \ldots; oc_n$ d'un système de transitions étiquetées T est un chemin si et seulement si :*

$$(Init; oc_1; \ldots; oc_n) \in \mathsf{Chemins}(T) \;\Leftrightarrow\; \exists \begin{pmatrix} x_0, \ldots, x_{n+1}, \\ E_1, \ldots, E_{n+1} \end{pmatrix} \cdot \begin{pmatrix} \{E_1, \ldots, E_{n+1}\} \subseteq \mathbb{Q} - \{\mathsf{q}_{Init}\} \wedge \\ ((\mathsf{q}_{Init}, x_0) \overset{(\mathsf{btrue}, A_0, Init)}{\rightsquigarrow} (E_1, x_1)) \wedge \\ \bigwedge_{i=1}^{n} ((E_i, x_i) \overset{(D_i, A_i, oc_i)}{\rightsquigarrow} (E_{i+1}, x_{i+1})) \end{pmatrix}$$

avec x_i des valuations des variables du système et E_i des noms d'états.

Terminologie. *On désigne par $\mathsf{Chemins}(T)$ l'ensemble des chemins d'un système de transitions T.*

4.3.3 Égalité des traces

Pour finir, établir l'équivalence sémantique entre un système de transitions T, produit avec les algorithmes 1 et 2, et le système B événementiel S dont il est issu, revient à montrer l'égalité de l'ensemble des chemins de T et de l'ensemble des traces de S. Nous voulons donc établir le théorème suivant :

Théorème 1 (Égalité des traces) *Si S est un système B événementiel pour lequel l'invariant I a été établi et si T est un système de transitions étiquetées généré à partir de S en utilisant les algorithmes 1 et 2, alors :*

$$\mathsf{Traces}(S) = \mathsf{Chemins}(T)$$

Démonstration 1 (Égalité des traces) *Montrons que pour toute séquence d'événements t, on a : $t \in \mathsf{Chemins}(T) \Leftrightarrow t \in \mathsf{Traces}(S)$. Soit $t \mathrel{\hat=} Init; oc_1; \ldots; oc_n$ un chemin de T. En utilisant la définition des chemins d'un **STES** (définition 18), on a alors :*

$$\exists \begin{pmatrix} x_0, \ldots, x_{n+1}, \\ E_1, \ldots, E_{n+1} \end{pmatrix} \cdot \begin{pmatrix} \{E_1, \ldots, E_{n+1}\} \subseteq \mathbb{Q} - \{\mathsf{q}_{Init}\} \wedge \\ ((\mathsf{q}_{Init}, x_0) \overset{(\mathsf{btrue}, A_0, Init)}{\rightsquigarrow} (E_1, x_1)) \wedge \bigwedge_{i=1}^{n} ((E_i, x_i) \overset{(D_i, A_i, oc_i)}{\rightsquigarrow} (E_{i+1}, x_{i+1})) \end{pmatrix}$$

Par application de la définition 17, caractérisant la condition franchissement d'une transition, nous avons :

$$\equiv \exists \begin{pmatrix} x_1, \ldots, x_{n+1}, \\ E_1, \ldots, E_{n+1} \end{pmatrix} \cdot \begin{pmatrix} \{E_1 \ldots E_{n+1}\} \subseteq \mathbb{Q} - \{\mathsf{q}_{Init}\} \;\wedge\; [x' := x_1]\mathsf{prd}_x(Init) \\ \wedge\; [x := x_1]\mathcal{D}ef(E_1) \;\wedge \\ \bigwedge_{i=1}^{n} \begin{pmatrix} [x := x_i](\mathcal{D}ef(E_i) \;\wedge\; \mathsf{Garde}(oc_i)) \wedge \\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(\mathsf{Action}(oc_i)) \wedge \\ [x := x_{i+1}]\mathcal{D}ef(E_{i+1}) \end{pmatrix} \end{pmatrix} \quad (1)$$

Nous allons montrer que cette formule (1) *est équivalente à la formule* (2) *suivante, issue du lemme 1, et qui définit les traces d'un système* B *événementiel, pour lequel l'invariant I a été établi. C'est-à-dire que nous ajoutons l'hypothèse supplémentaire que chaque valuation x_i établit I :*

$$\exists(x_1, \ldots, x_{n+1}) \cdot \left(\begin{array}{c} [x' := x_1]\mathsf{prd}_x(\mathit{Init}) \ \wedge \ [x := x_1]I \ \wedge \\ \bigwedge_{i=1}^{n}([x := x_i]\mathsf{Garde}(oc_i) \ \wedge \\ [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(\mathsf{Action}(oc_i)) \ \wedge \ [x := x_{i+1}]I) \end{array} \right) \quad (2)$$

L'implication **(1)** $\Rightarrow$ **(2)** *est vérifiée, car chacun des états E_i de $\mathbb{Q}-\{\mathsf{q}_{Init}\}$ est tel que $\mathcal{D}ef(E_i) \Rightarrow I$ (Condition 2, page 74). Par monotonie du $\mathcal{WP}$ on a donc $[x := y]\mathcal{D}ef(E_i) \Rightarrow [x := y]I$.*

Pour montrer l'implication **(2)** $\Rightarrow$ **(1)**, *nous devons exhiber une liste d'états $\{E_1, \ldots, E_{n+1}\}$ de $\mathbb{Q}-\{\mathsf{q}_{Init}\}$ telle que ces états vérifient* (1)*. Comme l'initialisation établit l'invariant et comme la condition 1 du choix de l'espace d'états impose que celui-ci couvre l'invariant ($I \Rightarrow \bigvee_{j=1}^{m} \mathcal{D}ef(E_j)$), alors il existe nécessairement un état E_1 de $\mathbb{Q}-\{\mathsf{q}_{Init}\}$ vérifié par $[x := x_1]$. Pour les mêmes raisons, si I est établi pour chacune des valeurs x_i de x, alors il existe un état E_i de $\mathbb{Q}-\{\mathsf{q}_{Init}\}$ pour chaque x_i, tel que $[x := x_i]\mathcal{D}ef(E_i)$ est vrai.*

□

4.4 Critères de choix des états

Dans les sections précédentes, nous avons détaillé une méthode permettant de construire un système de transitions étiquetées symbolique représentant l'ensemble des comportements d'un modèle B événementiel. Cependant, nous avons fait l'hypothèse que l'espace d'états est fourni par l'utilisateur. En effet, comme une propriété est définie par un ensemble de traces d'exécution, il s'ensuit que le choix des états permet de mettre en évidence certaines propriétés particulières d'un modèle. De plus, le choix des états permet de maîtriser la complexité de la vue que l'on a d'un système. Par exemple, utiliser des états qui se chevauchent, c'est-à-dire qui partagent des valeurs communes, entraîne nécessairement des duplications de transitions, ce qui a pour effet d'alourdir la description. Dans cette section, nous illustrons l'importance du choix de l'espace d'états, puis nous proposons différentes méthodes de choix.

4.4.1 Exemple de mise en évidence d'une propriété

Les figures 4.5 et 4.6 sont deux représentations minimales de l'exemple du canal de communication, pour lesquelles l'espace d'états choisi est différent. Ces deux figures sont sémantiquement équivalentes et possèdent le même nombre d'états. Nous proposons de valider, sur chacune de ces figures, que le système vérifie la propriété : « *l'événement Envoyer ne peut être suivi que par Traiter ou Reset* ».

Sur la figure 4.5 l'événement *Envoyer* mène nécessairement dans l'état **Envoi en cours**, où seuls les événements *Traiter* et *Reset* sont déclenchables. La propriété est donc trivialement vraie.

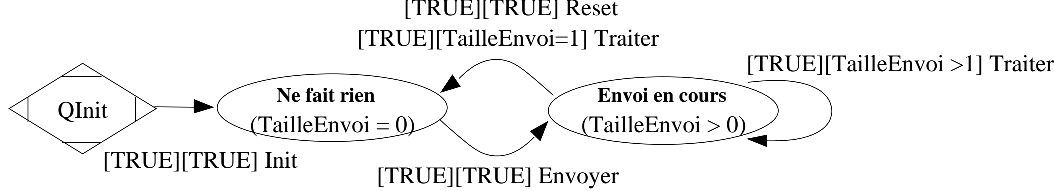


FIG. 4.5 – Première vue du canal de communication

En revanche, sur la figure 4.6, on ne peut avoir qu'une intuition de la validité de la propriété. En effet, l'événement *Envoyer* mène soit dans **E2**, où seuls les événements *Traiter* et *Reset* sont déclenchables, soit dans **E1**, où lui-même est déclenchable sous la condition $TailleEnvoi = 0$. Pour établir la propriété il faudrait donc pouvoir garantir que *TailleEnvoi* est nécessairement supérieur à 0 après exécution de *Envoyer*, ce qui ne peut être fait que par analyse de la spécification de l'événement.

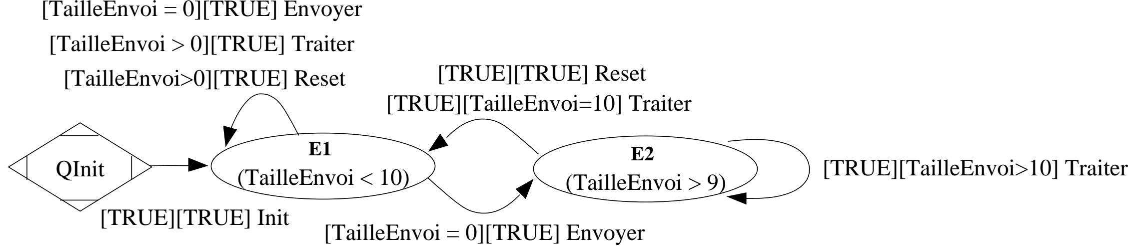


FIG. 4.6 – Seconde vue du canal de communication

Ainsi, bien que ces deux figures soient sémantiquement équivalentes, la figure 4.5 semble être un meilleur choix pour vérifier que l'événement *Envoyer* ne se succède pas à lui-même.

Pour mettre en avant une propriété particulière, il est nécessaire de connaître et d'analyser la propriété. Nous étudierons ce point plus en détail au chapitre 7. Pour l'instant, nous nous intéressons à fournir au concepteur des techniques de choix d'espace d'états. Intuitivement nous nous intéressons aux critères suivants :
- Plus petit nombre d'états tel que la propriété voulue soit mise en évidence
- Pas de chevauchement d'états
- Maximum de conditions réduites à btrue ou bfalse

4.4.2 Techniques de choix d'états

Choisir un espace d'états consiste à caractériser des valeurs particulières d'une ou de plusieurs variables. Nous nous inspirons de l'approche proposée par J-C Voisinet [Voi04], en ne nous intéressant qu'à une seule variable. Pour considérer plusieurs variables, il sera ensuite possible de composer les ensembles d'états construits.

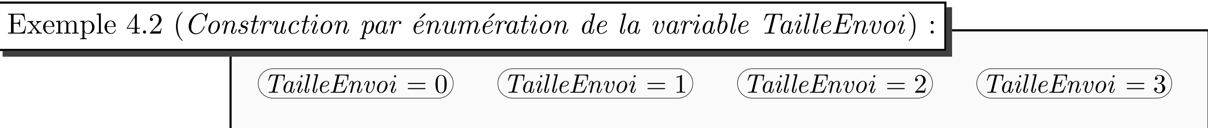
Dans cette section, nous proposons quatre techniques de choix d'un ensemble d'états, basée sur des découpages particuliers de l'invariant. Les méthodes proposées garantissent, par construction, la

complétude des états par rapport à l'invariant. La correction, quant à elle, n'est pas nécessairement établie. Nous terminons cette section en proposant deux techniques permettant de rendre un espace d'état correct ou complet par rapport à un invariant donné.

4.4.2.1 Construction par énumération

Cette technique élémentaire consiste à énumérer toutes les valeurs que peut prendre une variable. Elle s'applique très bien au cas des variables booléennes ou de type ensemble énuméré. Dans le cas des variables entières, le domaine de définition doit être suffisamment petit. Enfin, pour les variables de type ensemble (non énuméré), on peut s'intéresser à la cardinalité[1] de la variable, si celle-ci est bornée par un entier suffisamment petit.

Dans l'exemple précédent, la variable *TailleEnvoi* est définie sur un ensemble infini. Il n'est donc pas possible de construire d'ensemble d'états par énumération. Si l'on considérait que son domaine était borné à $0 \ldots 3$, alors on obtiendrait l'ensemble d'états :

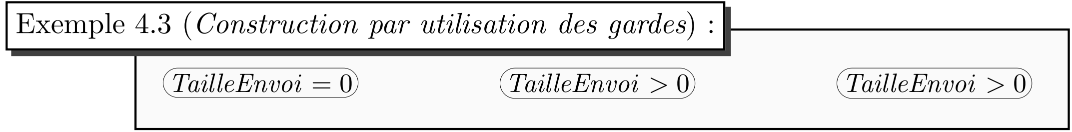


Le principal intérêt de cette méthode est de mettre en évidence chacun des comportements associés à une valuation particulière d'une variable. Par contre, cette méthode peut aboutir à la construction d'un système de transitions étiquetées contenant plus d'états qu'il n'est nécessaire pour observer les propriétés voulues.

4.4.2.2 Utilisation des gardes

Cette technique consiste à considérer la garde de chacun des événements comme un état du système. L'intérêt est alors de caractériser précisément l'état d'origine de chaque événement et de réduire toutes les conditions de déclenchabilité à btrue. Notons cependant que cette méthode peut amener à construire un espace d'états dont plusieurs états se chevauchent ou sont équivalents.

Par exemple, dans le cas du canal de communication, l'espaces d'états construit serait le suivant :



Afin d'éliminer les chevauchements d'états, il est possible de caractériser les intersections de ces états. Ainsi, pour chaque couple de prédicats P_1 et P_2, on construit trois états correspondants aux

[1] Notons toutefois que, dans la pratique, les prouveurs de l'*AtelierB* ou de *B4free*, les deux principaux prouveurs utilisés dans le domaine du B, sont peu efficaces pour les formules contenant des cardinalités. Leur utilisation est donc peu recommandée.

éléments communs de P_1 et P_2, aux éléments de P_1 n'étant pas dans P_2 et aux éléments de P_2 n'étant pas dans P_1 :

$$P_1 \wedge \neg P_2 \qquad P_2 \wedge \neg P_1 \qquad P_1 \wedge P_2$$

Cependant, cette méthode peut introduire des états vides dans l'espace d'états. Il peut alors être intéressant d'effectuer un traitement *a posteriori* pour les supprimer.

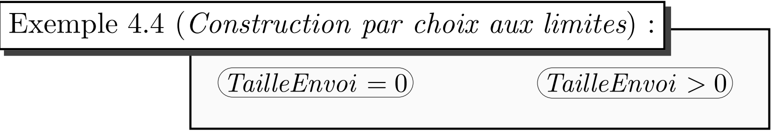
Enfin, notons qu'il est possible de considérer soit la garde logique, soit la garde syntaxique des événements. L'avantage de la première approche est que tout événement aura nécessairement sa condition à btrue depuis l'état correspondant à sa garde. Cependant, la garde logique d'un événement est le résultat d'un calcul de faisabilité, et peut nécessiter l'utilisation d'un simplificateur de prédicat. À l'inverse, l'avantage d'utiliser les gardes syntaxiques est que les prédicats écrits par le spécifieur sont souvent simplifiés, clarifiant ainsi la lecture. Cependant, il est possible que certains événements aient une condition de déclenchabilité différente de btrue depuis l'état correspondant à leur garde.

4.4.2.3 Caractérisation des états aux limites

Cette heuristique de choix des états se base sur la technique de test aux limites (ou d'interprétation abstraite). Celle-ci consiste à partitionner le domaine de définition d'une variable pour s'intéresser aux comportements des événements aux limites de ce domaine. En effet, pour éviter la violation d'invariant, les événements doivent parfois avoir un comportement particulier aux bornes du domaine de définition. C'est cela, que l'on s'intéresse à mettre en évidence avec cette approche.

Dans le cas des variables entières, elle consiste à caractériser 3 états pour chaque variable : sa borne maximum, sa borne minimum et les autres valeurs. Dans le cas où la variable n'a qu'une borne, par exemple définie sur les entiers naturels, il n'y a que 2 états générés. Si par contre son domaine de définition n'a pas de borne, les entiers relatifs par exemple, alors cette technique n'est pas intéressante, car un seul état serait construit. Enfin, dans le cas des variables de type ensemble, nous proposons de considérer deux états : vide et non-vide.

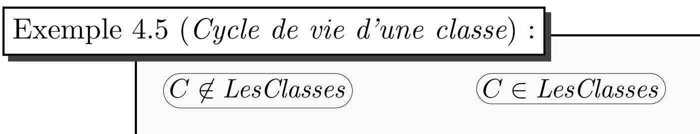
Par exemple, dans le cas de la variable *TailleEnvoi*, il n'y a que deux états, car elle n'a pas de borne maximum. Les états choisis dans cet exemple mettent en évidence que l'événement *Traiter* peut mener dans $TailleEnvoi = 0$, caractérisant la borne minimum de *TailleEnvoi*, mais qu'il n'y est pas déclenchable. Une représentation des comportements du canal de communication exploitant ces états peut être trouvée en figure 4.5, section 4.4.1.



4.4.2.4 Exhibition d'un témoin

Cette dernière méthode permet de mettre en évidence les évolutions d'un système en fonction d'un élément particulier de celui-ci : un témoin. La valeur de ce témoin est constante durant toute l'exécution du système, mais peut être contrainte à certaines propriétés. Chaque état est alors choisi pour caractériser la configuration du système par rapport au témoin.

Cette méthode permet notamment de mettre en évidence le cycle de vie d'un élément du système. Par exemple, dans le cas du modèle *GestionObjets*, décrite dans le chapitre 3 (en spécification 3.1), il est possible de s'intéresser à l'évolution d'une classe. Pour ce faire, il suffit d'introduire une classe quelconque C ($C \in \mathit{IdfClasses} - \{\mathit{Object}\}$) et de caractériser ses deux états : *pas encore chargée en mémoire* ou *déjà chargée en mémoire*.



Il est également possible d'augmenter le distinguo entre le témoin observé et les autres éléments en décrivant deux copies de chaque événement. La première permet de considérer les actions agissant sur le témoin et la seconde décrit les actions agissant sur tout élément autre que témoin. Typiquement, si la constante C est le témoin, alors un événement de la forme :

$ev \mathrel{\hat{=}} \textbf{any } p \textbf{ where } p \in \mathit{LesClasses} \textbf{ then } S \textbf{ end}$

est réécrit en :

$ev_p \mathrel{\hat{=}} \textbf{any } p \textbf{ where } p \in \mathit{LesClasses} \ \wedge \ p = C \textbf{ then } S \textbf{ end}$
$ev_{autres} \mathrel{\hat{=}} \textbf{any } p \textbf{ where } p \in \mathit{LesClasses} \ \wedge \ p \neq C \textbf{ then } S \textbf{ end}$

Avec cette mise en évidence, il est possible de distinguer directement sur les traces d'événements les comportements associés au témoin et ceux associés aux autres éléments.

De manière générale, la méthode d'exhibition d'un témoin n'est pas automatisable. Elle fait appel à l'expertise du développeur pour la définition du témoin, ainsi que pour le choix des propriétés à caractériser dans les différents états.

4.4.2.5 Correction et complétude automatique de l'espace d'états

Une fois l'espace d'états choisi, il est possible de le restreindre ou de le compléter pour le rendre correct et complet par rapport à l'invariant du modèle.

Pour rendre un espace d'état correct vis-à-vis d'un invariant, il suffit de renforcer le prédicat de définition P associé à un état E par $P \ \wedge \ I$. De cette manière, toute valeur vérifiant le prédicat de définition d'un état vérifie également l'invariant. Nous proposons donc de renforcer la fonction $\mathcal{D}\mathit{ef}$ comme suit :

$$\mathcal{D}\mathit{ef}(E) = P \ \wedge \ I$$

De la même manière, il est possible de compléter un espace d'états $\{E_1, \ldots, E_n\}$ en rajoutant un état E_{n+1} contenant exactement l'ensemble des valuations vérifiant l'invariant mais couverte par aucun état. On définit :

$$\mathcal{D}\mathit{ef}(E_{n+1}) = I \ \wedge \ \neg \bigvee_{i=1}^{n} \mathcal{D}\mathit{ef}(E_i)$$

4.4.3 Bilan

Nous proposons de laisser l'utilisateur fournir manuellement l'espace d'états, ce qui lui permet d'obtenir une vue aussi proche que possible de la propriété voulue. Cependant, nous lui proposons un ensemble de techniques lui permettant de structurer sa réflexion. De manière générale, l'approche par choix des gardes permet de donner une intuition générale des comportements d'un système. Pour affiner cette vue, on peut alors diviser ces états selon l'une des autres techniques. Pour mettre en évidence les évolutions d'une unique variable ayant un domaine de définition suffisamment petit, alors la technique par énumération semble la mieux adaptée. Au contraire, si le domaine de définition est trop grand ou si l'on veut généraliser cette approche sur plusieurs variables en évitant l'explosion combinatoire, alors la technique de choix des états aux limites pourra convenir. Cette dernière technique peut également être utilisée pour focaliser l'attention sur les cas particuliers que sont les comportements des événements aux limites. Enfin, la technique d'exhibition d'un témoin semble parfaitement adaptée pour exhiber un cycle de vie ou bien les comportements du système pour une valuation quelconque des données.

Différents travaux ont été menés dans l'équipe VASCO pour permettre d'automatiser la construction d'un espace d'états à partir d'une spécification B. Par exemple, l'outil *GénéEtat* [Ham02, Ham03], décrit plus en détail dans la section 6.1.1, implante les techniques d'énumération et de choix basées sur les gardes. Une seconde approche, a été proposée par A. Idani [Ida06b, IL06, Ida06a]. Il se base sur des travaux de R. Laleau et F. Polack [LP01] et introduit des schémas de transformation permettant d'effectuer la conversion des modèles UML en modèles B. En plus des différentes techniques de choix d'états que nous proposons, il présente deux autres patrons.

Le premier patron est une approche permettant de mettre en évidence le cycle de vie d'un élément par rapport à une décomposition de son domaine de définition. Par exemple, un passager p vérifie soit $p \in PASSAGERS - Enregistrés$, soit $p \in PASSAGERS \cap Enregistrés$. Ce patron est un cas particulier d'exhibition de témoin, pour lequel l'auteur décrit une description détaillée de l'approche à suivre. Le second patron, quant à lui, est une utilisation des autres patrons pour décomposer soit l'image, soit l'ensemble des antécédents, associés à un élément par une relation.

Enfin, l'auteur propose également d'exploiter l'approche concurrente des diagrammes d'états-transitions. Cette technique consiste à énumérer chacune des variables et à construire un système de transitions par variables. Ces systèmes sont ensuite parallélisés et donne chacun une vue particulière du système global. Cette approche permet de représenter plusieurs propriétés sur un seul diagramme, mais également de limiter l'explosion combinatoire associée à l'énumération de toutes les variables d'un système.

4.5 Application au B classique

Nous voulons adapter l'approche proposée dans ce chapitre pour construire l'ensemble des séquences d'appel d'opérations autorisées par une spécification B classique. Pour ce faire, nous proposons de convertir les machines B en systèmes B événementiel en associant un événement à chaque opération. Intuitivement, cela revient à considérer chaque occurrence d'un événement comme l'ob-

servation d'un appel de son opération associée. Il suffit alors de changer les pré-conditions en gardes. Dans un premier temps, nous proposons des règles de traduction d'opérations en événements, avant de discuter sur la sémantique d'une telle transformation.

4.5.1 Traduction d'une machine B en système B événementiel

Pour traduire une opération en événement il est nécessaire de considérer les paramètres entrants et sortants, car ceux-ci n'ont pas de sens dans le cas d'événements se déclenchant spontanément. Nous proposons ci-après deux approches pour prendre en compte les paramètres : paramètres internalisés (approche classique) ou externalisés (états définissables en termes des paramètres). Cette seconde approche est décrite en section 4.5.3.

En B, Une opération peut avoir l'une des formes suivantes :

$$\begin{array}{ll} (1) & op \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \\ (2) & Res \longleftarrow op \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \\ (3) & op(params) \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \\ (4) & Res \longleftarrow op(params) \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \end{array}$$

Dans le cas (1), l'opération n'a aucun paramètre. Il suffit donc de remplacer syntaxiquement **pre** par **select** pour transformer cette opération en un événement se déclenchant sous les mêmes conditions que l'opération.

$$(1) \quad op \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \quad \rightsquigarrow \quad op \hat{=} \ \textbf{select}\ P\ \textbf{then}\ S\ \textbf{end}$$

Dans le cas (2), les paramètres sortants n'ont pas d'impact sur le comportement de la machine. Cependant, il est possible que ces paramètres soient utilisés comme variables temporaires dans le corps de l'opération. Nous proposons donc de déclarer les paramètres en tant que variables locales.

$$(2) \quad Res \longleftarrow op \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} \quad \rightsquigarrow \quad op \hat{=} \ \begin{array}{l} \textbf{select}\ P\ \textbf{then} \\ \quad \textbf{var}\ Res\ \textbf{in}\ S\ \textbf{end} \\ \textbf{end} \end{array}$$

Dans les cas (3) et (4), les paramètres entrants peuvent être caractérisés par un choix non-déterministe. Celui-ci permet alors d'assigner n'importe quelles valeurs aux paramètres, parmi celles autorisées.

$$\begin{array}{llll} (3) & op(params) \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} & \rightsquigarrow & op \hat{=} \ \textbf{any}\ params\ \textbf{where}\ P\ \textbf{then}\ S\ \textbf{end} \\ (4) & Res \longleftarrow op(params) \hat{=} \ \textbf{pre}\ P\ \textbf{then}\ S\ \textbf{end} & \rightsquigarrow & op \hat{=} \ \begin{array}{l} \textbf{any}\ params\ \textbf{where}\ P\ \textbf{then} \\ \quad \textbf{var}\ Res\ \textbf{in}\ S\ \textbf{end} \\ \textbf{end} \end{array} \end{array}$$

La transformation des opérations en événements selon ces règles permet donc de conserver inchangé le corps des opérations. Dans la section suivante, nous étudions la sémantique de cette transformation.

4.5.2 Sémantique de la transformation proposée

En B événementiel, une séquence $e_1\ ;\ e_2$ de deux événements est réalisable s'il existe une valuation atteignable par e_1 et depuis laquelle e_2 est déclenchable. À l'inverse, en B classique, deux opérations $o_1\ ;\ o_2$ ne peuvent être appelées en séquence que si toutes les valuations atteignables par o_1 vérifient la pré-condition de o_2. Plus formellement, en B événementiel on s'intéresse à la faisabilité d'un enchaînement, tandis qu'en B classique on s'intéresse à sa terminaison.

$$\mathsf{trm}(o_1\ ;\ o_2) \hat{=} [o_1\ ;\ o_2]\mathsf{btrue}$$
$$\mathsf{fis}(e_1\ ;\ e_2) \hat{=} \neg[e_1\ ;\ e_2]\neg\mathsf{btrue}$$

L'exemple 4.6 met en évidence cette différence de sémantique. Intuitivement, cette différence porte principalement sur les structures non-déterministes, puisque ce sont les seules à être affectées par la double négation, en dehors des substitutions pré-conditionnées et gardées.

Exemple 4.6 (*Séquences d'opérations ou d'événements*) :

Considérons les trois opérations :
$op_1 \hat{=}$ **pre** $x = 0$ **then choice** $x := 1$ **or** $x := 2$ **end end** ;
$op_2 \hat{=}$ **pre** $x = 1$ **then** $x := x + 1$ **end** ;
$op_3 \hat{=}$ **pre** $x > 0$ **then** $x := 0$ **end**

Il est possible d'appeler op_3 après op_1, car toute valuation atteignable par la première opération vérifie la pré-condition de la seconde. Par contre, op_2 ne peut pas être appelée après op_1, car l'appelant ne connaît pas la valeur de x et ne peut donc pas garantir le respect de la pré-condition de op_2.

Conformément à la transformation décrite, notée $\mathcal{T}$, ces opérations peuvent être changées en les événements suivants :
$\mathcal{T}(op_1) \hat{=}$ **select** $x = 0$ **then choice** $x := 1$ **or** $x := 2$ **end end** ;
$\mathcal{T}(op_2) \hat{=}$ **select** $x = 1$ **then** $x := x + 1$ **end** ;
$\mathcal{T}(op_3) \hat{=}$ **select** $x > 0$ **then** $x := 0$ **end**

Dans ce cas, $\mathcal{T}(op_3)$ est déclenchable après $\mathcal{T}(op_1)$, mais $\mathcal{T}(op_2)$ l'est aussi, puisque 1 est une valeur possible après $\mathcal{T}(op_2)$. Sa condition de déclenchabilité est alors $x = 1$.

Les comportements d'un système B événementiel S, généré à partir d'une machine M, est donc une sur-approximation des séquences d'appels possibles de M. On en déduit le résultat suivant :

Proposition 1 (Correction de la construction présentée) *L'ensemble des séquences d'appels autorisées par une spécification B classique est inclus dans l'ensemble des comportements du modèle B événementiel construit selon les règles présentées en section 4.5.1.*

Il s'ensuit qu'une propriété de vivacité établie sur le **STES** produit n'est pas nécessairement vérifiée sur le modèle. Seules les propriétés de sûreté sont préservées par cette transformation.

L'exemple 4.7 est une traduction de la machine *GestionObjets*, décrite dans le chapitre 3 (Spécification 3.1).

Exemple 4.7 (*Traduction d'une machine en système*) :

```
machine GestionObjets
...
operations
  bb←−InstanceOf(C1, C2) ≙ pre
    C1 ∈ IdfClasses ∧ C2 ∈ IdfClasses
  then
    bb := bool(C1 ↦ C2 ∈ Super+)
  end

;

  rr←−Charger(C1, C2) ≙ pre
    C1 ∈ IdfClasses ∧ C2 ∈ IdfClasses
  then
    if C1 ∈ LesClasses ∨ C2 ∉ LesClasses
       ∨ card(LesClasses) = ClassesMax
    then rr := false
    else rr := true
       || LesClasses := LesClasses ∪ {C1}
       || Super := Super ∪ {C1 ↦ C2}
    end
  end
end
```

```
system GestionObjets
.../* Inchangé */
events
  InstanceOf ≙ any C1, C2 where
    C1 ∈ IdfClasses ∧ C2 ∈ IdfClasses
  then
    var bb in
      bb := bool(C1 ↦ C2 ∈ Super+)
    end
  end
;
  Charger ≙ any C1, C2 where
    C1 ∈ IdfClasses ∧ C2 ∈ IdfClasses
  then
    var rr in
      if C1 ∈ LesClasses ∨ C2 ∉ LesClasses
       ∨ card(LesClasses) = ClassesMax
      then rr := false
      else rr := true
       || LesClasses := LesClasses ∪ {C1}
       || Super := Super ∪ {C1 ↦ C2}
      end
    end
  end
end
```

Il est possible de construire le **STES** présenté en figure 4.7, et caractérisant l'ensemble des séquences d'opérations autorisées dans la machine *GestionObjets*. Nous avons voulu mettre en évidence ici le fait que seule l'opération *Charger* permet de modifier l'ensemble *LesClasses*. Nous avons donc choisi un espace d'états exhibant les cas aux limites $(\mathsf{card}(LesClasses) = ClassesMax)$ et $(LesClasses = \{Object\})$, ainsi que le cas général $(\mathsf{card}(LesClasses) \in 2..ClassesMax - 1)$.

4.5.3 Externalisation des paramètres

Dans la méthode présentée précédemment la valeur des paramètres entrants et sortants ne peut pas être utilisée pour caractériser les états. En effet, la portée des paramètres est limitée à l'exécution d'un événement. Dans cette section, nous proposons une seconde méthode, qui permet d'externaliser ces valeurs. Il est alors possible de choisir des états, et donc d'exprimer les comportements du système, par rapport aux valeurs des paramètres.

Pour ce faire, nous proposons de mettre en évidence l'acquisition des entrées et la consommation des résultats par des événements indépendants et nous caractérisons les paramètres entrants et sortants par les variables globales $Params_{op}$ et Res_{op}. Pour décrire une opération op, nous introduisons alors jusqu'à trois événements : $op_{Entrants}$ permettant d'instancier les paramètres entrants, $op_{Sortants}$

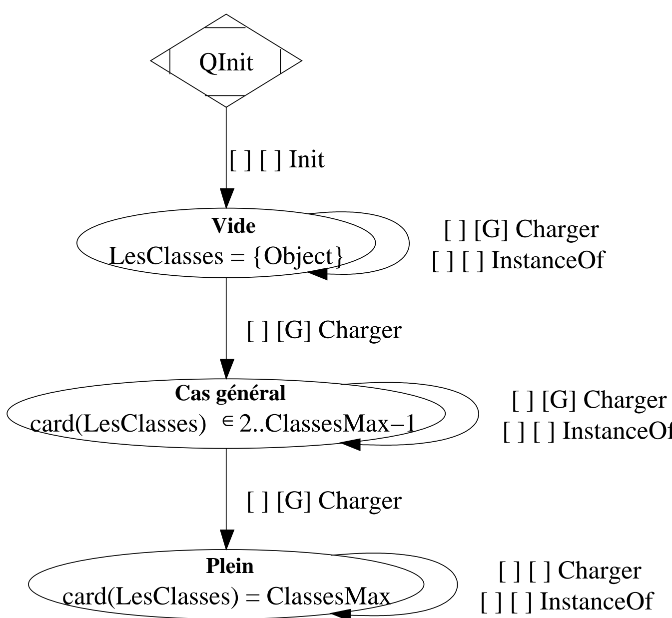


FIG. 4.7 – Représentation de l'ensemble des séquences d'opérations autorisées dans l'exemple 4.7

permettant de simuler la consommation des paramètres sortants et *op* qui est le traitement décrit dans le corps de l'opération associée.

Cependant, la valeur des variables caractérisant les paramètres d'une opération n'a de sens que juste avant ou juste après l'exécution de l'événement. Nous utilisons alors un sémaphore pour caractériser ces instants et séquentialiser strictement l'exécution des événements modélisant une unique opération. Ce sémaphore est défini par un ensemble pouvant avoir la valeur *Libre*, si aucune séquence d'événements n'est en cours, $Sem_{op_i/Entrants}$ pour chacune des opérations op_i dont on externalise les paramètres entrants et $Sem_{op_i/Sortants}$ pour chacune des opérations op_i dont on externalise les paramètres sortants.

$$
\begin{array}{l}
\textbf{sets } \textit{SÉMAPHORE} = \{ \textit{Libre}, \\
\qquad Sem_{op_1/Entrants}, Sem_{op_1/Sortants}, \\
\qquad Sem_{op_2/Entrants}, Sem_{op_2/Sortants}, \\
\qquad \ldots \}
\end{array}
$$

Déclaration du sémaphore

Dans le cas des paramètres entrants, il est également important de préciser que, lorsque des paramètres sont prêts pour un appel de l'opération *op*, alors sa pré-condition P_{op} est nécessairement vérifiée sur les variables globales $Params_{op}$. Enfin, d'un point de vue pratique, pour déclarer des paramètres entrants et sortants en variables globales, nous avons besoin de leur type $T_{Params_{op}}$ et $T_{Res_{op}}$. En effet, celui-ci est utilisé dans l'invariant, ainsi que dans l'initialisation, pour affecter une valeur initiale aux variables.

sets $\mathit{SÉMAPHORE} = \{Libre, Sem_{op/Entrants}, Sem_{op/Sortants}, \ldots$ /* Autres sémaphores */ }

variables $Params_{op}, Res_{op}, sémaphore, \ldots$ /* Variables du système */
invariant
/* Invariant du système */ ...
$\wedge\ sémaphore \in \mathit{SÉMAPHORE}$
$\wedge\ Params_{op} \in T_{Params_{op}}$
$\wedge\ Res_{op} \in T_{Res_{op}}$
$\wedge\ (sémaphore = Sem_{op/Entrants} \Rightarrow P_{op})$ /* où P_{op} est la pré-condition de op */

initialisation
$Params_{op} :\in T_{Params_{op}} \mid\mid Res_{op} :\in T_{Res_{op}} \mid\mid sémaphore := Libre \mid\mid \ldots$

events
/* Instanciation des paramètres entrants */
$op_{Entrants} \mathrel{\hat{=}}$ **any** pp **where** $sémaphore = Libre \wedge [Params_{op} := pp]P_{op}$ **then**
$sémaphore := Sem_{op/Entrants} \mid\mid Params_{op} := pp$
end ;

/* Exécution du corps de l'opération */
$op \mathrel{\hat{=}}$ **select** $sémaphore = Sem_{op/Entrants}$ **then** $S_{op} \mid\mid sémaphore := Sem_{op/Sortants}$ **end** ;

/* Ré-initialisation du sémaphore */
$op_{Sortants} \mathrel{\hat{=}}$ **select** $sémaphore = Sem_{op/Sortants}$ **then** $sémaphore := Libre$ **end** ;

...
end

Traduction d'une opération ayant des paramètres entrants et sortants – Forme (4)

Pour ne pas externaliser les paramètres entrants ou sortants d'une opération, formes (2) ou (3) d'opération, il suffit de ne pas introduire le sémaphore et l'événement associés. Enfin, en cas de conflit entre le nom d'un paramètre et le nom d'une variable globale, il suffit de renommer le paramètre.

La spécification 4.2 est une traduction de la machine *GestionObjets*, où les paramètres des deux opérations on été externalisés. Afin d'éviter le conflit des noms des paramètres entrants, nous avons renommé les paramètres C_1 et C_2 de *Charger* en C_3 et C_4.

Spécification 4.2 (*Externalisation des paramètres*) :

system *GestionObjets*
sets $SÉMAPHORE = \{\ Libre,$
$Sem_{InstanceOf\,/\,Entrants}, Sem_{InstanceOf\,/\,Sortants},$
$Sem_{Charger/Entrants}, Sem_{Charger/Sortants}\}$
variables $\ldots, C_1, C_2, C_3, C_4$, *sémaphore*, *rr*, *bb*
invariant
$\ldots$
$\wedge\ C_1 \in IdfClasses\ \wedge\ C_2 \in IdfClasses\ \wedge\ C_3 \in IdfClasses\ \wedge\ C_4 \in IdfClasses$
$\wedge\ sémaphore \in SÉMAPHORE$
$\wedge\ (sémaphore = Sem_{InstanceOf\,/\,Entrants}\ \Rightarrow\ C_1 \in IdfClasses\ \wedge\ C_2 \in IdfClasses)$
$\wedge\ (sémaphore = Sem_{Charger/Entrants}\ \Rightarrow\ C_3 \in IdfClasses\ \wedge\ C_4 \in IdfClasses)$
$\wedge\ rr \in \mathsf{BOOL}\ \wedge\ bb \in \mathsf{BOOL}$
initialisation
$C_1 :\in IdfClasses \mid\mid C_2 :\in IdfClasses \mid\mid C_3 :\in IdfClasses \mid\mid C_4 :\in IdfClasses$
$\mid\mid sémaphore := Libre \mid\mid rr :\in \mathsf{BOOL} \mid\mid bb :\in \mathsf{BOOL} \mid\mid\ \ldots$
events

$InstanceOf_{Entrants} \hat{=}$ **any** p_1, p_2
where $sémaphore = Libre\ \wedge\ p_1 \in IdfClasses\ \wedge\ p_2 \in IdfClasses$
then $sémaphore := Sem_{InstanceOf\,/\,Entrants} \mid\mid C_1 := p_1 \mid\mid C_2 := p_2$
end ;
— *Paramètres entrants d'InstanceOf*

$InstanceOf \hat{=}$ **select** $sémaphore = Sem_{InstanceOf\,/\,Entrants}$ **then**
$bb := \mathbf{bool}(C_1 \mapsto C_2 \in Super^+) \mid\mid$
$sémaphore := Sem_{InstanceOf\,/\,Sortants}$
end ;
— *InstanceOf*

$InstanceOf_{Sortants} \hat{=}$ **select** $sémaphore = Sem_{InstanceOf\,/\,Sortants}$ **then**
$sémaphore := Libre$
end ;
— *Paramètres sortants d'InstanceOf*

$Charger_{Entrants} \hat{=}$ **any** p_1, p_2
where $sémaphore = Libre\ \wedge\ p_1 \in IdfClasses\ \wedge\ p_2 \in IdfClasses$
then $sémaphore := Sem_{Charger/Entrants} \mid\mid C_3 := p_1 \mid\mid C_4 := p_2$
end ;
— *Paramètres entrants de Charger*

$Charger \hat{=}$ **select** $sémaphore = Sem_{Charger/Entrants}$ **then**
if $C_3 \in LesClasses\ \vee\ C_4 \notin LesClasses\ \vee\ \mathsf{card}(LesClasses) = ClassesMax$
then $rr := \mathsf{false}$
else $rr := \mathsf{true}$
$\mid\mid LesClasses := LesClasses \cup \{C_3\}$
$\mid\mid Super := Super \cup \{C_3 \mapsto C_4\}$
end
$\mid\mid sémaphore := Sem_{Charger/Sortants}$
end ;
— *Charger*

$Charger_{Sortants} \hat{=}$ **select** $sémaphore = Sem_{Charger/Sortants}$ **then**
$sémaphore := Libre$
end
— *Paramètres sortants de Charger*

end

La figure 4.8 est une représentation des comportements du système *GestionObjets*. Nous avons

choisi un espace d'états qui permet d'exhiber deux classes de traces d'exécution de *Charger* : la première aboutit nécessairement dans un statut d'erreur (**Pas de chargement**), tandis que la seconde aboutit nécessairement au chargement de C_3 en mémoire (**Chargement effectué**). Les états ($\mathbf{E}_1$) et ($\mathbf{E}_2$) caractérisent alors les conditions nécessaires et suffisantes pour que le chargement échoue ou réussisse. Ces conditions portent notamment sur les classes C_3 et C_4 qui sont les paramètres entrants de *Charger*. Pour observer que le chargement a été effectué avec succès, nous observons également la valeur du paramètre sortant rr. Celui-ci vaut vrai si et seulement si le chargement est effectué.

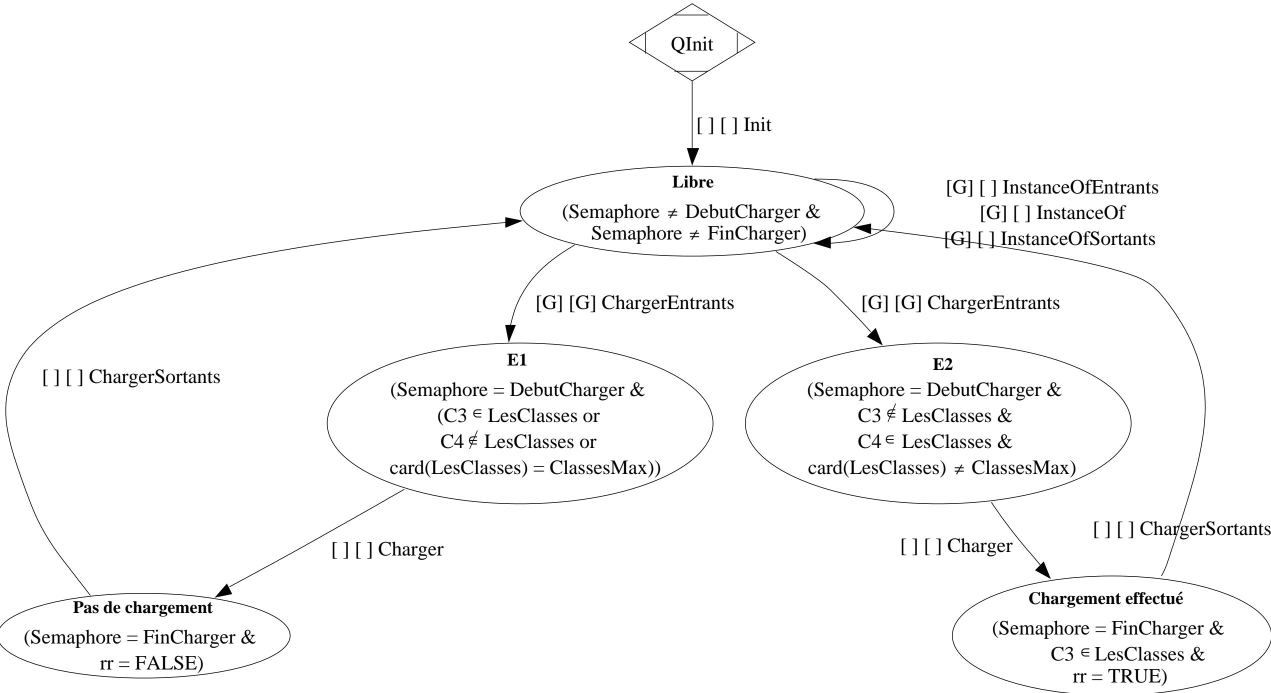


FIG. 4.8 – Mise en évidence de la condition permettant la réussite du chargement

4.6 Synthèse

Dans ce chapitre, nous avons défini les systèmes de transitions étiquetées symboliques basés sur des doubles conditions de franchissement des transitions : la condition de déclenchabilité et la condition d'atteignabilité.

Nous avons ensuite introduit une méthode permettant de construire, à partir d'un ensemble d'états, une représentation graphique de l'ensemble des comportements d'un système B événementiel. Le calcul de la relation de transition est basé sur la vérification d'obligations de preuve. Lors du calcul de chaque transition, nous nous intéressons à trois états caractéristiques des conditions : btrue, bfalse ou conditionné. Nous avons également introduit un quatrième cas, permettant de mettre en évidence les défauts de preuve.

La preuve de l'équivalence sémantique entre un système B événementiel et la représentation

générée par notre algorithme a été faite. Cette preuve se base sur une sémantique de traces d'occurrences d'événements définissant aussi bien les systèmes B que les systèmes de transitions étiquetées.

Ensuite, nous avons proposé des heuristiques permettant d'aider au choix des états. Ces heuristiques se basent sur le découpage de l'invariant en fonction de différents critères. L'idée est alors de laisser au spécifieur le choix des états qu'il veut mettre en évidence, en fonction de la, ou des, propriétés qu'il veut observer.

Enfin, nous avons adapté cette méthode de construction des comportements à l'approche B classique, pour construire une sur-approximation de l'ensemble des séquences d'appel d'opérations autorisées par une machine. Ce faisant, nous avons proposé deux traductions possibles des opérations permettant d'internaliser ou d'externaliser les paramètres entrants et sortants. La première méthode de traduction est plus simple que la seconde et permet de générer des systèmes de transitions en termes des variables d'état du système. La seconde méthode de traduction est, quant à elle, plus complexe, mais permet de caractériser les états du système en termes des paramètres entrants en sortants des opérations.

Dans le chapitre suivant, nous étendons notre approche pour prendre en compte le développement par raffinement.

Représentation des comportements de raffinements B événementiel

5

The refinement calculus is a logical framework for reasoning about programs. It is concerned with two main questions : is a program correct with respect to a given specification, and how can we improve, or refine, a program while preserving its correctness.

R-J. Back and J. von Wright

Sommaire

5.1 Choix d'un formalisme de description . . . 98

5.1.1 Définition des systèmes de transitions hiérarchiques . . . 99

5.1.2 Représentation des **STEH** . . . 102

5.2 Construction des comportements d'un raffinement B événementiel . . . 105

5.2.1 Définition de l'espace d'états . . . 107

5.2.2 Construction de la relation de transition sur les états-feuille . . . 108

5.2.3 Réduction du nombre de transitions externes . . . 110

5.3 Illustration de la construction d'un STEH . . . 117

5.4 Application au B classique . . . 119

5.5 Synthèse . . . 119

Le processus de développement par raffinement permet de décrire les modèles B par étapes successives, tout en maintenant une certaine traçabilité des données et des événements. Dans ce chapitre, notre objectif est de mettre en évidence cette notion de traçabilité des données et des événements, pour faciliter la compréhension et la validation du modèle.

Nous illustrons nos propos avec le raffinement présenté dans l'exemple 3.4 et rappelé ci-après. Pour mémoire, celui-ci raffine le canal de communication en introduisant un buffer, de telle sorte que le nombre d'éléments en attente d'être traité est borné. C'est le nouvel événement *EnvoyerSuite* qui émet, un par un, les éléments composant chaque message, ce qui autorise un entrelacement des traitements et des envois.

Spécification 5.1 (*Rappel de la spécification 3.4, section 3.6.5*) :

refinement *Canal_De_Communication_R*
refines *Canal_De_Communication*
constants *TailleBuff*
properties $TailleBuff \in \mathbb{N}_1$
variables *DansBuffer*, *AEnvoyer*
invariant $AEnvoyer \in \mathbb{N} \wedge DansBuffer \in 0..TailleBuff \wedge DansBuffer + AEnvoyer = TailleEnvoi$
initialisation $DansBuffer := 0 \mid\mid AEnvoyer := 0$
events
Envoyer $\hat{=}$ **select** $AEnvoyer = 0 \wedge DansBuffer = 0$ **then** $AEnvoyer :\in \mathbb{N}_1$ **end** ;
EnvoyerSuite $\hat{=}$ **select** $AEnvoyer > 0 \wedge DansBuffer < TailleBuff$ **then**
$AEnvoyer := AEnvoyer - 1 \mid\mid DansBuffer := DansBuffer + 1$
end ;
Traiter $\hat{=}$ **select** $DansBuffer > 0$ **then** $DansBuffer := DansBuffer - 1$ **end** ;
Reset $\hat{=}$ **select** $AEnvoyer > 0 \vee DansBuffer > 0$ **then** $AEnvoyer := 0 \mid\mid DansBuffer := 0$ **end**
end

Dans l'approche B événementiel, le raffinement prend en compte deux aspects orthogonaux :

1. **Le changement de représentation des données.** Les événements sont alors redéfinis pour prendre en compte les nouvelles représentations des données et peuvent préciser les calculs. En particulier le non-déterminisme peut être levé.
2. **L'affinage du comportement du système.** Le raffinement autorise le renforcement des gardes, ainsi que l'introduction de nouveaux événements.

Ces deux aspects du raffinement ont des impacts différents sur la visualisation. Le premier ne modifie pas intrinsèquement le comportement du système et donc sa visualisation. Néanmoins, la réduction du non-déterminisme peut modifier les conditions de franchissement des transitions existantes, mais n'ajoute pas de nouvelles transitions. Le deuxième aspect, quant à lui, influe directement sur les comportements du système, en affinant les comportements existants et en permettant l'ajout de transitions associées aux nouveaux événements.

Dans cette thèse, nous proposons de représenter les comportements d'un raffinement comme une instance plus précise des comportements de son abstraction. Pour ce faire, nous introduisons la notion de hiérarchie dans les systèmes de transitions. Cette construction hiérarchique permet de conserver la structure générale de la représentation des comportements du système abstrait, ainsi que les noms des états abstraits. Les efforts de compréhension faits sur le système de transitions abstrait peuvent ainsi être mis à profit pour aider à comprendre le raffinement, en ne s'intéressant qu'aux modifications ou aux ajouts effectués.

5.1 Choix d'un formalisme de description

Dans cette section, nous présentons les systèmes de transitions hiérarchiques en deux temps : d'abord nous les définissons avec chacun de leurs concepts sous-jacents (états hiérarchiques, facto-

risation, etc.), puis nous présentons le lien entre ces définitions et leur représentation graphique.

5.1.1 Définition des systèmes de transitions hiérarchiques

5.1.1.1 Définition générale

Classiquement [MLS97, Yan00], les systèmes de transitions hiérarchiques sont décrits par un ensemble de systèmes de transitions et une fonction de hiérarchisation. Cette dernière associe à chaque état le système de transitions qu'il contient. La principale limitation de cette approche est liée aux transitions externes[1]. En effet, les transitions sont nécessairement entre deux états d'un même système de transitions. Cette construction interdit donc les transitions entre deux états quelconques du système.

Dans notre approche, nous avons choisi de définir un système de transitions étiquetées hiérarchique comme un système de transitions étiquetées symbolique où seuls les états sont structurés par la fonction de hiérarchisation $\mathcal{S}up$. Celle-ci associe un super-état à chaque état. Ce choix permet de décrire aisément des transitions entre des états n'ayant pas le même super-état. Chaque état hiérarchique est donc un arbre d'états et la fonction $\mathcal{S}up$ forme une forêt.

Terminologie. *Une **forêt** est un ensemble d'**arbres**. Pour qu'une relation décrive une forêt, il suffit que ce soit une fonction associant un père à chacun de ses fils et qui ne contienne pas de cycle.*

De plus, chaque état hiérarchique peut contenir un sous-état initial et un sous-état final. Ce sont les ensembles $\mathbb{Q}_{Init}$ et $\mathbb{Q}_{Final}$ qui précisent ceux des sous-états qui sont initiaux ou finaux. Comme l'état q_{Init} est l'état initial du système, nous avons choisi de le traiter séparément. Il n'est donc pas dans l'ensemble $\mathbb{Q}_{Init}$.

Définition 19 (Système de transitions hiérarchique) *Un système de transitions hiérarchique est défini par un déca-uplet* $(\mathbb{V}, \mathbb{E}, \mathbb{Q}, \mathsf{q}_{Init}, \mathbb{Q}_{Init}, \mathbb{Q}_{Final}, \mathcal{D}ef, \mathbb{L}, \mathbb{R}, \mathcal{S}up)$ *tel que :*

- $(\mathbb{V}, \mathbb{E}, \mathbb{Q}, \mathsf{q}_{Init}, \mathcal{D}ef, \mathbb{L}, \mathbb{R})$ *est un système de transitions étiquetées où* $\mathbb{Q}$ *est l'ensemble de **tous** les noms d'état, **quelque soit leur niveau de profondeur** ;*
- $\mathcal{S}up$ *est une fonction sans cycle associant chaque état à son super-état et formant une forêt* $(\mathcal{S}up \in \mathbb{Q} \nrightarrow \mathbb{Q} \ \wedge \ \mathsf{id}(\mathbb{Q}) \cap \mathcal{S}up^+ = \emptyset)$
- $\mathbb{Q}_{Init}$ *est l'ensemble des noms des sous-états initiaux* $(\mathbb{Q}_{Init} \subseteq \mathsf{dom}(\mathcal{S}up))$ *;*
- $\mathbb{Q}_{Final}$ *est l'ensemble des noms des sous-états finaux* $(\mathbb{Q}_{Final} \subseteq \mathsf{dom}(\mathcal{S}up))$*.*

Un système de transitions symbolique est donc un système de transitions hiérarchique dont aucun état n'est hiérarchisé ($\mathcal{S}up = \emptyset \ \wedge \ \mathbb{Q}_{Final} = \emptyset \ \wedge \ \mathbb{Q}_{Init} = \emptyset$). Notons également que les prédicats présents dans la définition d'un **STEH** portent tous sur le même espace de variables $\mathbb{V}$. Ce point sera détaillé en section 5.2.1.

[1] Entre l'extérieur et l'intérieur d'un état hiérarchique.

Terminologie. *En abrégé, nous notons **STEH**, un Système de Transitions Étiquetées Hiérarchique.*

Dans le cadre de cette thèse, nous ne considérons pas tous les **STEH**. En effet, nous ne considérons que ceux ayant une certaine cohérence entre l'inclusion des états et celle de leurs prédicats de définition. Dans les sections suivantes, nous précisons cette contrainte avant de définir la sémantique que nous donnons à de tels systèmes de transitions.

Terminologie. *Par la suite nous appelons **état-feuille** un état n'ayant pas de sous-état et **racine** un état n'ayant pas de super-état.*

Plus formellement, on note $\mathsf{Feuilles}(\mathcal{S}up)$ et $\mathsf{Racines}(\mathcal{S}up)$ l'ensemble des états-feuille et des racines de la fonction de hiérarchie $\mathcal{S}up$. On définit alors :

$$\begin{cases} \mathsf{Feuilles}(\mathcal{S}up) \hat{=} \mathsf{dom}(\mathcal{S}up) - \mathsf{ran}(\mathcal{S}up) \\ \mathsf{Racines}(\mathcal{S}up) \hat{=} \mathsf{ran}(\mathcal{S}up) - \mathsf{dom}(\mathcal{S}up) \end{cases}$$

5.1.1.2 Cohérence entre l'inclusion des états et leurs prédicats de définition

Afin de conserver une cohérence entre la hiérarchie des états et leur définition, il est important que deux états ne puissent être inclus que si leurs prédicats de définition sont également inclus. De plus, nous imposons que toute valuation présente dans un état composé soit présente dans au moins un de ses sous-états. Comme tous les prédicats de définition des états portent sur le même ensemble de variables $\mathbb{V}$, alors on définit la condition suivante :

Condition 4 (Cohérence de la hiérarchie)
Pour tout état hiérarchique E, tel que $\mathcal{S}up^{-1}[\{E\}] = \{E_1, \ldots, E_n\}$:

$$\mathcal{D}ef(E) \Leftrightarrow \bigvee_{i=1}^{n} \mathcal{D}ef(E_i)$$

5.1.1.3 Définition des transitions factorisées

Une transition peut être factorisée soit sur son état de départ, soit son état d'arrivée, soit sur les deux à la fois. Nous définissons les transitions factorisées comme suit :

Définition 20 (Transition factorisée) *Une transition $(E, (D, A, ev), F)$ est factorisée sur E si et seulement si E n'est pas un état-feuille. De même, elle est factorisée sur F si et seulement si F n'est pas un état-feuille.*

La sémantique d'une transition factorisée peut être définie de différentes manières. Dans les StateCharts, par exemple, une transition ayant pour origine un état hiérarchique E peut être déclenchée depuis l'un au moins des sous-états de E. Ce choix introduit une sur-approximation, puisque la lecture du système de transitions ne permet pas de savoir depuis quels sous-états cette transition est franchissable. Une alternative consiste à préciser cette information dans les étiquettes des transitions ou par le biais de signaux (dans le cas de systèmes synchrones). Par exemple, E. Mich, Y. Lakhnech et M. Siegel [MLS97] proposent de factoriser les transitions en étendant les

étiquettes des transitions de telle sorte que les listes des sous-états d'arrivée et de départ y soient précisées. Enfin, l'approche prise par F. Maraninchi dans l'outil *Argos* [Mar92, Mar91, MR01] est de contourner le problème en utilisant des signaux. En effet, il est possible de remplacer une transition sortant d'un état composé par l'émission d'un signal de sortie, qui est capté par une transition partant de l'état composé. Cette approche purement synchrone permet donc de factoriser des transitions externes sortantes, qui ne sont déclenchables que depuis une partie des sous-états.

Pour notre part, nous définissons qu'une transition $(E, (D, A, ev), F)$ factorisée sur son état de départ E implique que la même transition recopiée sur les sous-états soit valide avec les mêmes conditions de franchissement D et A (et réciproquement pour les transitions factorisées sur leur état d'arrivée). L'intérêt de cette sémantique, est qu'elle préserve les conditions de franchissement. Cela signifie notamment que si une transition factorisée atteint toujours un état F, alors elle atteint toujours chacun des sous-états de F. Typiquement, les transitions de la figure 5.1 ne sont pas factorisables, car la condition d'atteignabilité de l'état F par l'événement ev depuis l'état E est btrue, tandis que ev ne peut atteindre F_1 que si $x > 0$ et atteindre F_2 que si $x \leq 0$. La sémantique que nous proposons permet de préserver ce type d'information.

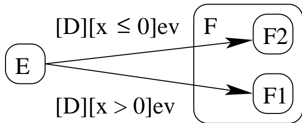


FIG. 5.1 – Transitions ne pouvant pas être factorisées

Nous définissons alors la sémantique d'une transition factorisée en termes de son aplatissement sur les sous-états de la manière suivante :

Définition 21 (Sémantique d'une transition factorisée)
Si $(E, (D, A, ev), F)$ *est une transition factorisée sur l'état* E*, tel que* $\mathcal{S}up^{-1}[\{E\}] = \{E_1, \ldots, E_n\}$*, alors elle peut être remplacée par l'ensemble de transitions valides* $\{(E_i, (D, A, ev), F) \mid i \in 1..n\}$*, sans que cela ne modifie l'ensemble des chemins du* ***STEH****.*

Si $(E, (D, A, ev), F)$ *est une transition factorisée sur l'état* F*, tel que* $\mathcal{S}up^{-1}[\{F\}] = \{F_1, \ldots, F_n\}$*, alors elle peut être remplacée par l'ensemble de transitions valides* $\{(E, (D, A, ev), F_i) \mid i \in 1..n\}$*, sans que cela ne modifie l'ensemble des chemins du* ***STEH****.*

Par transitivité, toute transition factorisée est définie en terme des états-feuille du système.

5.1.1.4 Sous-état initiaux ou finaux

Les sous-états initiaux et finaux permettent de préciser l'entrée et la sortie d'un sous-système de transitions. Cependant, nous verrons en section 5.1.2 que cette notion n'est utilisée que pour alléger la représentation des **STEH**.

5.1.1.5 Sémantique des STEH

Finalement, nous définissons la sémantique des systèmes de transitions hiérarchiques en fonction du système de transitions symbolique en lequel ils s'aplatissent. Aplatir un **STEH** consiste à le

projeter sur ses états-feuille. Plus formellement on définit :

Définition 22 (Sémantique d'un STEH)
La sémantique d'un ***STEH*** $(\mathbb{V},\ \mathbb{E},\ \mathbb{Q}_H,\ \mathsf{q}_{Init},\ \mathbb{Q}_{Init},\ \mathbb{Q}_{Final},\ \mathcal{D}ef_H,\ \mathbb{L}_H,\ \mathbb{R}_H,\ \mathcal{S}up)$ *est définie, par aplatissement, par le* ***STES*** $(\mathbb{V},\ \mathbb{E},\ \mathbb{Q}_S,\ \mathsf{q}_{Init},\ \mathcal{D}ef_S,\ \mathbb{L}_S,\ \mathbb{R}_S)$ *tel que :*

$\mathbb{Q}_S \mathrel{\hat{=}} \mathsf{Feuilles}(\mathcal{S}up)$ /* On ne conserve que les états-feuille */

$\mathcal{D}ef_S \mathrel{\hat{=}} \mathbb{Q}_S \lhd \mathcal{D}ef_H$ /* Prédicats de définition des états-feuille conservés */

/* Aplatissement de la relation de transition */

$$\mathbb{R}_S \mathrel{\hat{=}} \left\{ (E_i,(D,A,ev),F_i) \;\middle|\; E_i \in \mathbb{Q}_S \ \wedge\ F_i \in \mathbb{Q}_S \ \wedge\ \exists (E,ev,F)\cdot \begin{pmatrix} (E,(D,A,ev),F) \in \mathbb{R}_H \\ \wedge\ (E_i \in \mathcal{S}up^{-1})^*[\{E\}] \\ \wedge\ (F_i \in \mathcal{S}up^{-1})^*[\{F\}] \end{pmatrix} \right\}$$

$\mathbb{L}_S \mathrel{\hat{=}} \{(D,A,ev) \mid \exists (E,F)\cdot ((E,(D,A,ev),F) \in \mathbb{R}_S)\}$ /* Étiquettes présentes dans $\mathbb{R}_S$ */

Rappels de notation :
$A \lhd R \mathrel{\hat{=}} \{(x,y) \mid (x,y) \in R \ \wedge\ x \in A\}$ *(Restriction sur le domaine).*
$R^* \mathrel{\hat{=}} \mathsf{id}(A) \cup R \cup (R\ ;\ R) \cup (R\ ;\ R\ ;\ R) \cup \ldots$, *où* $R \in A \leftrightarrow A$ *(Fermeture réflexive et transitive).*

5.1.2 Représentation des STEH

Dans cette section nous faisons le lien entre la définition des systèmes de transitions hiérarchiques et leur représentation. Ce faisant, nous caractérisons des conditions de représentation relatives aux sous-états initiaux et finaux, nécessaires pour donner un sens à toutes les représentations.

5.1.2.1 Transitions factorisées

La figure 5.2 est un exemple de représentation d'une transition $(E,(D,A,ev),F)$ factorisée sur son état d'arrivée ($F \notin \mathsf{Feuilles}(\mathcal{S}up)$). L'extrémité factorisée d'une transition est indiquée par un demi-cercle noir. Dans cet exemple, chacun des sous-états de F est atteignable par ev depuis E sous la condition $D \ \wedge\ A$.

$(E,(D,A,ev),F) \in \mathbb{R} \ \wedge\ F \notin \mathsf{Feuilles}(\mathcal{S}up)$

A. *Transition définie*

[D][A]ev F F2 F1

B. *Transition affichée*

FIG. 5.2 – Exemple de transition factorisée

5.1.2.2 Transitions en relation avec des sous-états initiaux ou finaux

Comme nous l'avons vu, les transitions qui ne sont pas factorisées sont des transitions reliant des états-feuille et dont la sémantique est celle des transitions des **STES**. Parmi ces transitions, nous caractérisons le cas particulier des transitions externes en relation avec un état-feuille initial ou final. Intuitivement, un état hiérarchique est un état qui a été décomposé par raffinement pour décrire un processus qui, dans un cas idéal, commence à son état initial et finit à son état final. Les transitions externes partant d'un sous-état final ou atteignant un sous-état initial sont donc intéressantes, car elles correspondent au début ou à la fin du processus décrit par le sous-système et n'ont pas d'incidence sur les comportements de ce dernier. Nous mettons donc en évidence ces

transitions. Celles atteignant un état-feuille initial F_1 sont représentées en relation avec le super-état de F_1 (figure 5.3) et celles partant d'un état-feuille final E_1 sont représentées en relation avec le super-état de E_1.

$$(E,(D,A,ev),F_1) \in \mathbb{R} \ \wedge \ F_1 \in \mathsf{Feuilles}(\mathcal{S}up)$$
$$\wedge \ F_1 \in \mathbb{Q}_{Init} \ \wedge \ E \notin \mathcal{S}up^{-1}[\{\mathcal{S}up(F_1)\}]$$

A. *Transition définie*

B. *Transition affichée*

FIG. 5.3 – Exemple de transition en relation avec un sous-état feuille initial

Par transitivité, une transition peut être représentée en relation avec le $n^{\text{ème}}$ super-état $\mathcal{S}up^n(F)$ de F si tous les super-états intermédiaires sont initiaux et si sa représentation sur le super-état de niveau $n-1$ est externe.).

$$(E,(D,A,ev),F_1) \in \mathbb{R} \ \wedge \ F_1 \in \mathsf{Feuilles}(\mathcal{S}up)$$
$$\wedge \ F_1 \in \mathbb{Q}_{Init} \ \wedge \ \mathcal{S}up(F_1) \in \mathbb{Q}_{Init} \ \wedge \ E \notin (\mathcal{S}up^{-1})^*[\{\mathcal{S}up^2(F_1)\}]$$

A. *Transition définie*

B. *Transition affichée*

FIG. 5.4 – Exemple de transition en relation avec un sous-sous-état initial

Condition 5 (Représentation d'une transition atteignant un sous-état initial)
Soit $(E,(D,A,ev),F)$ une transition où F est un état-feuille. Elle est représentée comme atteignant le $n^{\text{ème}}$ super-état de F si :

1. *F est un sous-état feuille initial ($F \in \mathsf{Feuilles}(\mathcal{S}up) \cap \mathbb{Q}_{Init}$)*
2. *tous les super-états intermédiaires de F sont initiaux ($\bigwedge_{i=1}^{n-1} \mathcal{S}up^i(F) \in \mathbb{Q}_{Init}$)*
3. *la transition $(E,(D,A,ev),\mathcal{S}up^{n-1}(F))$ est externe ($E \notin \mathcal{S}up^{-1}[\{\mathcal{S}up^n(F)\}]$)*
4. *le $(n+1)^{\text{ème}}$ super-état de F ne vérifie pas ces conditions. C'est-à-dire :*

$\mathcal{S}up^n(F) \notin \mathsf{dom}(\mathcal{S}up)$	*$\mathcal{S}up^n(F)$ est une racine*
$\vee \ \mathcal{S}up^n(F) \notin \mathbb{Q}_{Init}$	*ou $\mathcal{S}up^n(F)$ n'est pas un sous-état initial*
$\vee \ E \in \mathcal{S}up^{-1}[\{\mathcal{S}up^{n+1}(F)\}]$	*ou la transition $(E,(D,A,ev),\mathcal{S}up^n(F))$ n'est pas externe*

Terminologie. *Une transition est* ***remontée*** *sur un super-état E si elle est représentée comme atteignant E, bien qu'elle soit définie comme atteignant le sous-état initial de E ou qu'elle prenne son origine dans le sous-état final de E.*

De manière similaire, nous avons les conditions suivantes sur la représentation des transitions partant d'un sous-état final :

Condition 6 (Représentation d'une transition partant d'un sous-état final)
Soit $(E, (D, A, ev), F)$ *une transition où* E *est un état-feuille. Elle est représentée comme partant du* $n^{\text{ème}}$ *super-état de* E *si :*

1. E *est un sous-état final* ($E \in \mathsf{Feuilles}(\mathcal{S}up) \cap \mathbb{Q}_{Final}$)
2. *tous les super-états intermédiaires de* E *sont finaux* ($\bigwedge_{i=1}^{n-1} \mathcal{S}up^i(E) \in \mathbb{Q}_{Final}$)
3. *la transition* $(\mathcal{S}up^{n-1}(E), (D, A, ev), F)$ *est externe* ($F \notin \mathcal{S}up^{-1}[\{\mathcal{S}up^n(E)\}]$).
4. *le* $(n+1)^{\text{ème}}$ *super-état de* E *ne vérifie pas ces conditions. C'est-à-dire :*

$$\begin{array}{rl} \mathcal{S}up^n(E) \notin \mathsf{dom}(\mathcal{S}up) & \textit{Sup}^n(E) \textit{ est une racine} \\ \vee\ \mathcal{S}up^n(E) \notin \mathbb{Q}_{Final} & \textit{ou Sup}^n(E) \textit{ n'est pas un sous-état final} \\ \vee\ F \in \mathcal{S}up^{-1}[\{\mathcal{S}up^{n+1}(E)\}] & \textit{ou la transition } (\mathcal{S}up^n(E), (D, A, ev), F) \textit{ n'est pas externe} \end{array}$$

5.1.2.3 Bilan sur la représentation des transitions

La figure 5.5 résume la manière dont est représenté chacun des types de transitions. Si une transition t est en relation avec un état hiérarchique F (Fig. 5.5.A), alors elle est factorisée sur cet état. Si une transition externe atteint un état-feuille initial F_1, alors elle est représentée comme atteignant le super-état F (Fig. 5.5.B). Sinon, si une transition n'est pas externe, ou si elle n'atteint pas un état-feuille initial F_1, alors est représentée comme atteignant directement F_1 (Fig. 5.5.C).

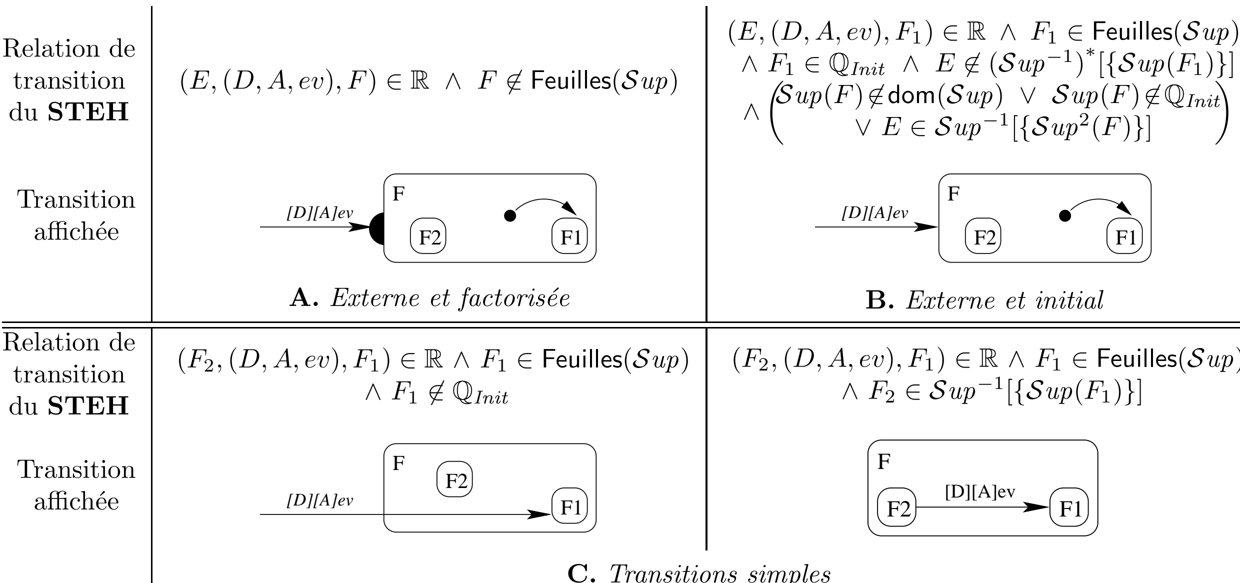


FIG. 5.5 – Exemple de représentation des différents types de transitions

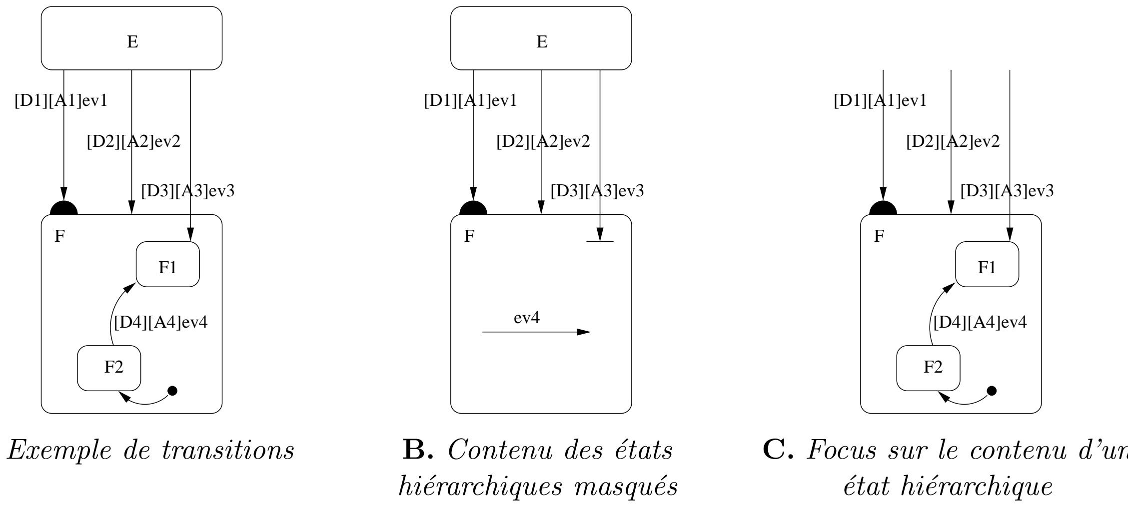
5.1.2.4 Différentes granularités de représentation

Afin de minimiser l'espace nécessaire à la représentation d'un système de transitions, il est possible de masquer ou d'afficher le contenu des états hiérarchiques, suivant le niveau de granularité d'observation choisi.

Masquer le contenu d'un état hiérarchique (figure 5.6.B) permet de supprimer les détails pour se concentrer sur les interactions entre les différents sous-systèmes. Lors du masquage, nous pro-

posons de conserver une vision simplifiée des transitions internes, en ne conservant que le nom des événements pouvant se déclencher (cas de la transition ev_4). De plus, pour distinguer les transitions externes étant ni factorisées, ni en relation avec un sous-état initial ou final, nous utilisons la notation introduite dans les StateCharts : la transition atteint l'intérieur de l'état hiérarchique et est terminée par un segment (cas de la transition ev_3).

À l'inverse, zoomer sur le contenu d'un état hiérarchique permet de faire abstraction des interactions du sous-**STES** avec l'extérieur (figure 5.6.C). Lors de ce type de représentation, nous proposons de conserver les transitions externes en relation avec l'état hiérarchique affiché.



A. *Exemple de transitions* **B.** *Contenu des états hiérarchiques masqués* **C.** *Focus sur le contenu d'un état hiérarchique*

FIG. 5.6 – Différentes granularités de représentation des états hiérarchiques.

5.2 Construction des comportements d'un raffinement B événementiel

De même que la sémantique d'un **STEH** est définie par le **STES** en lequel il s'aplatit, l'ensemble des comportements d'un modèle construit par raffinement est défini par les événements décrits dans son raffinement le plus bas. Nous associons donc les états-feuille d'un **STEH** aux états du dernier raffinement. Par extension, nous associons exactement un niveau de hiérarchie à chaque niveau de description du modèle, mettant ainsi en évidence le lien entre les données des différents raffinements du modèle B.

Terminologie. *Dans un* ***STEH****, un* ***niveau de hiérarchie*** *est l'ensemble des états ayant la même* ***profondeur****; c'est-à-dire le même nombre de super-états. Le niveau 0 est l'ensemble des états racine.*

Par exemple, considérons la figure 5.7 représentant les comportements du raffinement du canal de communication. Celle-ci a été construite à partir de la figure 4.5 (section 4.4.1) décrivant les comportements de la spécification du canal de communication à laquelle a été ajouté un niveau de hiérarchie des états. Les transitions, quant à elles, ont été affinées. Ainsi, les états **Ne fait rien** et

(Envoi en cours) correspondent à des états de l'abstraction, tandis que les quatre états-feuille sont une subdivision introduite dans le raffinement, pour visualiser les comportements des événements en termes des nouvelles variables.

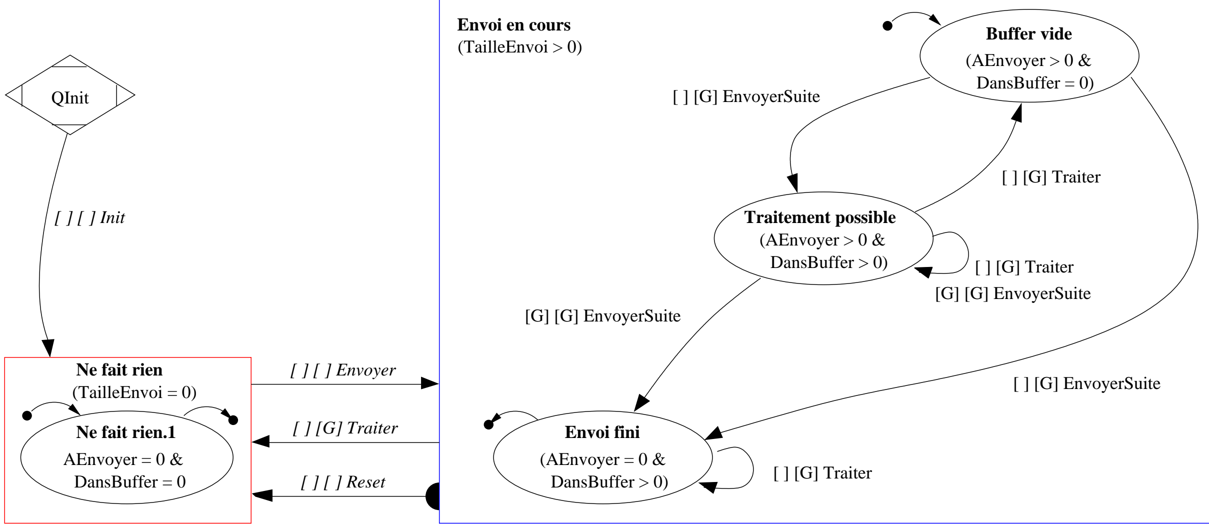


FIG. 5.7 – Représentation hiérarchique du canal de communication.

Ainsi, pour construire un **STEH** associé à un raffinement R d'un système abstrait S, nous procédons en modifiant un système de transitions hiérarchique T_S représentant les comportements de S. La méthode de construction que nous proposons se décompose en trois phases :

- **Définition de l'espace d'états** par projection de l'espace d'états de T_S sur les variables du raffinement et ajout d'un niveau de hiérarchie ;
- **Construction de la relation de transition entre les états-feuille** en utilisant la connaissance des comportements du système abstrait pour ne calculer qu'une partie des transitions ;
- **Réduction du nombre de transitions externes** par factorisation et par introduction des sous-états initiaux et finaux.

Dans cette construction, les transitions de T_S ne sont pas conservées, mais sont utilisées pour diminuer la complexité et le coût de construction de la relation de transition. La hiérarchie des états, quant à elle, permet de préserver la structure du système de transitions. Ainsi, comme l'utilisateur obtient une vue plus précise des comportements, mais ayant la même structure, il peut, par comparaison avec le **STEH** du niveau précédent, comprendre plus facilement le raffinement en ne s'intéressant qu'à leurs différences. Les trois étapes de la construction d'un **STEH** sont successivement décrites dans cette section.

5.2.1 Définition de l'espace d'états

Les différents prédicats définissant un **STEH** (définition des états et conditions de franchissement) portent sur un même ensemble de variables $\mathbb{V}$. Cependant, chacun des raffinements qui compose un modèle B est décrit par un ensemble distinct de variables. Comme les comportements du modèle sont définis par les événements du dernier niveau de raffinement, c'est l'espace de variables de celui-ci que nous utilisons pour construire le **STEH** associé.

Chacun des états, quelque soit son niveau de hiérarchie, est donc défini en terme des variables du raffinement. Or, les états qui ne sont pas des feuilles ont été décrits en fonction des variables plus abstraites. Nous devons donc introduire la notion de projection, pour définir les prédicats abstraits en termes des variables du raffinement. Intuitivement, projeter un état E défini par un prédicat $\mathcal{D}ef_S(E)$ conformément à un invariant de liaison L consiste à caractériser l'ensemble des valuations des variables du raffinement liées par L à des valuations vérifiant $\mathcal{D}ef_S(E)$.

Définition 23 (Projection d'un état) *Soit E un état défini par le prédicat $\mathcal{D}ef_S(E)$ portant sur les variables x et L un invariant de liaison entre les espaces de variables x et y. La projection $\mathsf{Proj}_L(E)$ de E selon L est définie par :*

$$\mathsf{Proj}_L(E) \hat{=} \{y \mid \exists x \cdot (L \ \wedge \ \mathcal{D}ef_S(E))\}$$

Chaque état peut donc être défini en termes des variables du raffinement. Pour ce qui est du nom des états, nous proposons de les conserver, même après leur projection. Ainsi, les noms des états sont composés d'un identifiant permettant de faire facilement le lien entre le texte et la figure, et d'une description textuelle de son prédicat de définition. Le prédicat présent dans le nom est donc défini en termes des variables du raffinement dans lequel il a été initialement défini. Cette information n'est pas utilisée pour le processus de construction, mais elle peut être utile à l'utilisateur pour faciliter le lien avec les **STEH** des niveaux plus abstraits.

Terminologie. *Dans le cas des états hiérarchiques, nous choisissons de* ***nommer*** *chaque état d'un système de transitions avec un identifiant permettant de clarifier les renvois entre le texte et le diagramme, ainsi que le prédicat utilisé pour décrire cet état dans son raffinement associé.*

La figure 5.7 utilise cette nomenclature. Par exemple, l'état **Envoi en cours** contient dans son étiquette le prédicat $TailleEnvoi > 0$ qui est sa définition dans l'abstraction. Sa définition dans le raffinement n'est pas précisée explicitement sur la représentation graphique, mais elle est définie par $\exists TailleEnvoi \cdot (L \ \wedge \ TailleEnvoi > 0)$.

Dans notre approche, l'utilisateur fournit l'ensemble des nouveaux sous-états ; c'est-à-dire qu'il fixe les prédicats de définition des états ajoutés en états-feuille ($\mathbb{Q}_U$), en précisant le super-état associé à chacun d'entre eux ($\mathcal{S}up_U$). Cependant, cela nécessite de connaître les noms de ces états. C'est pourquoi, nous considérons, dans ce chapitre, que l'ensemble des noms des nouveaux états $\mathbb{Q}_U$ est fourni par l'utilisateur. Nous faisons également l'hypothèse que cet espace est disjoint de $\mathbb{Q}_S$.

Si T_S est le **STEH** associé à l'abstraction S, alors l'espace d'états de T_R, représentant les comportements du raffinement R, est construit de la manière suivante :

Algorithme 3 (*Construction incrémentale de l'espace d'états de T_R*) :

Paramètres en entrée : $\mathbb{Q}_S$, $\mathcal{D}ef_S$, $\mathcal{S}up_S$, $\mathbb{Q}_U$, $\mathcal{D}ef_U$ et $\mathcal{S}up_U$

$\mathbb{Q}_R := \mathbb{Q}_S \cup \mathbb{Q}_U \;||$ /* *Construction de l'espace de noms d'états* */

$\mathcal{D}ef_R := \{(\mathsf{q}_{Init} \mapsto \mathsf{btrue})\}$ /* q_{Init} *est conservé* */
$\quad \cup \{E \mapsto \mathsf{Proj}_L(E) \mid E \in \mathbb{Q}_S - \{\mathsf{q}_{Init}\}\}$ /* *États abstraits projetés sur les variables de R* */
$\quad \cup \mathcal{D}ef_U \;||$ /* *Nouveaux états donnés par l'utilisateur* */

$\mathcal{S}up_R := \mathcal{S}up_S \cup \mathcal{S}up_U$ /* *Construction de la hiérarchie* */

Résultat : $\mathbb{Q}_R$, $\mathcal{D}ef_R$ et $\mathcal{S}up_R$

Comme nous l'avons vu, la sémantique d'un **STEH** est définie par le **STES** en lequel il s'aplatit. L'espace d'état de ce dernier ($\mathbb{Q}_U$) doit donc vérifier les conditions de correction et de complétude par rapport à l'invariant, qui sont définies dans le chapitre précédent (Section 4.2.1.1). Il est facile de vérifier ces conditions de manière incrémentale sur T_R sous les conditions suivantes :

1. Les états-feuille de T_S sont corrects et complets par rapport à l'invariant de S ;
2. Les nouveaux états vérifient la condition 4 (Section 5.1.1.2) de cohérence de la hiérarchie.

La première hypothèse est vérifiée, puisque T_S vérifie ces hypothèses. La seconde nécessite la vérification des obligations de preuve décrites dans la condition suivante :

Condition 7 (Cohérence incrémentale de la hiérarchie)
Tout état hiérarchique de $\mathcal{S}up_U$ est correct et complet par rapport à ses sous-états :

$$\forall E \cdot \left(E \in \mathsf{ran}(\mathcal{S}up_U) \Rightarrow \left(\mathcal{D}ef_R(E) \Leftrightarrow \left(\bigvee_{E_i \in \mathcal{S}up_U^{-1}[\{E\}]} \mathcal{D}ef_U(E_i) \right) \right) \right)$$

5.2.2 Construction de la relation de transition sur les états-feuille

Dans cette section, nous nous intéressons à construire la relation de transition $\mathbb{R}_R$ sur les états-feuille $\mathbb{Q}_U$. Dans la section suivante, nous utilisons ce résultat pour prendre en compte la notion de hiérarchie en proposant des méthodes permettant de réduire, *a posteriori*, le nombre de transitions externes.

Pour construire la relation de transition sur les états-feuille, il suffit d'utiliser les algorithmes 1 et 2 décrits dans le chapitre précédent. Cette construction est toutefois coûteuse en nombre d'obligations de preuve. Nous pouvons en effet exploiter les propriétés du raffinement pour limiter les transitions à considérer. Dans un premier temps, nous nous intéressons à simplifier la construction des transitions associées aux événements présents dans l'abstraction. Nous affinons ensuite la construction des événements introduits par raffinement.

5.2.2.1 Exploitation de la relation de transition abstraite

Par définition du raffinement, les comportements d'un raffinement R sont nécessairement autorisés par son système abstrait S. La propriété suivante permet d'établir qu'une transition ne peut

être valide dans le raffinement que si elle est valide dans l'abstraction pour le même événement et entre les super-états des états correspondants.

Propriété 1 (Implication des conditions des transitions) *Soit la transition* $(E, (D_R, A_R, ev), F)$ *du système raffiné* T_R *et la transition* $(\mathcal{S}up(E), (D_S, A_S, ev), \mathcal{S}up(F))$ *du système abstrait* T_S*. S'il a été prouvé que* R *raffine* S*, alors on a nécessairement :*

$$L \wedge \mathcal{D}ef_S(\mathcal{S}up(E)) \wedge D_R \Rightarrow D_S \quad (1.1)$$
$$L \wedge \mathcal{D}ef_S(\mathcal{S}up(E)) \wedge D_R \wedge A_R \Rightarrow A_S \quad (1.2)$$

La propriété 1.1 met en évidence le fait qu'une transition n'est déclenchable depuis un état E du raffinement que si elle l'est déjà, depuis le super état de E, dans l'abstraction. La propriété 1.2 concerne, quant à elle, l'atteignabilité des états. La preuve de ces propriétés est donnée en annexe B.3.

La relation de transition $\mathbb{R}_R$ peut alors être construite en utilisant les algorithmes 1 et 2 (Sections 4.2.2.2 et 4.2.2.4), où ce dernier est modifié, comme suit, pour ne s'intéresser qu'aux transitions déjà existantes dans l'abstraction.

Algorithme 4 (*Trouver* D *et* A *– Exploitation de* $\mathbb{R}_S$) :

Paramètres en entrée : le nom d'événement ev et les noms d'états E et F.

Si $ev \in \mathbb{E}_S$ et qu'il n'existe pas de transition depuis $\mathcal{S}up(E)$ vers $\mathcal{S}up(F)$ par ev dans $\mathbb{R}_S$ alors :
$D :=$ bfalse || $A :=$ bfalse /* *FIN. Transition non franchissable* */

Sinon :

Recherche de D et A selon l'algorithme 2

Résultat : les conditions D et A.

5.2.2.2 Transitions associées aux nouveaux événements

L'algorithme proposé dans la section précédente construit notamment les transitions associées aux nouveaux événements, puisque les événements non présents dans $\mathbb{E}_S$ ne sont pas ignorés. Cependant, certaines propriétés de ces derniers pourraient être prises en compte pour diminuer le coût de cette construction. En effet, en B événementiel, un nouvel événement raffine **skip** (Section 3.6.4). Il ne peut donc pas franchir de transitions que **skip** ne peut pas franchir dans l'abstraction. À partir de l'obligation de preuve du raffinement, nous pouvons déduire pour tout état E de l'abstraction :

$$L \wedge \mathcal{D}ef_R(E) \Rightarrow [ev_N]\mathcal{D}ef_R(E)$$

Ainsi, un nouvel événement ne peut pas atteindre un état F depuis un état E si leurs super-états sont disjoints, c'est-à-dire si l'obligation de preuve suivante est vérifiée :

$$\forall y \cdot (\mathcal{D}ef_R(\mathcal{S}up(E)) \Rightarrow \neg\mathcal{D}ef_R(\mathcal{S}up(F)))$$

Propriété 2 (Cas de réflexivité des nouveaux événements) *Soit E et F deux états. Si leur super-états sont disjoints, alors un événement ev_N introduit par raffinement ne peut pas franchir de transition entre E et F :*

$$\forall y \cdot (\mathcal{D}ef_R(\mathcal{S}up(E)) \Rightarrow \neg \mathcal{D}ef_R(\mathcal{S}up(F))) \;\wedge\; \mathcal{D}ef_R(E) \Rightarrow [ev_N]\neg \mathcal{D}ef_R(F)$$

L'algorithme proposé dans la section précédente peut donc être modifié comme suit. Notons qu'il est possible de commencer par vérifier, pour chaque couple d'états hiérarchiques, s'ils sont disjoints, avant de construire la relation de transition.

Algorithme 5 (*Trouver D et A – Gestion des nouveaux événements*) :

Paramètres en entrée : le nom d'événement ev et les noms d'états E et F.

Si $ev \in \mathbb{E}_S$ et qu'il n'existe pas de transition depuis $\mathcal{S}up(E)$ vers $\mathcal{S}up(F)$ par ev dans $\mathbb{R}_S$ alors :
- $D :=$ bfalse || $A :=$ bfalse /* *FIN. Transition non franchissable* */

Sinon :
- Si $ev \notin \mathbb{E}_S$ et si $\mathcal{S}up(E)$ et $\mathcal{S}up(F)$ sont disjoints, alors :
  - $D :=$ bfalse || $A :=$ bfalse /* *FIN. Transition non franchissable* */
- Sinon :
  - Recherche de D et A selon l'algorithme 2

Résultat : les conditions D et A.

5.2.2.3 Bilan sur la construction de la relation de transition

Dans cette section, nous avons présenté une méthode permettant de construire la relation de transition d'un **STEH** en prenant en compte les propriétés du raffinement pour diminuer le nombre d'obligations de preuve à vérifier et donc diminuer le risque de défaut de preuve.

Le tableau 5.1 résume le nombre d'obligations de preuve nécessaires pour calculer le système de transition d'un raffinement, dans le cas où l'on connaît déjà celui de la spécification abstraite. À titre de comparaison, nous rappelons dans ce tableau le coût de construction d'un **STES** en utilisant les algorithmes 1 et 2.

Dans la section suivante, nous exploitons la hiérarchie pour réduire le nombre de transitions externes présentes dans la relation de transition entre états-feuille produite dans cette section.

5.2.3 Réduction du nombre de transitions externes

Dans cette section, l'enjeu est de réduire au maximum le nombre de transitions externes, afin de faciliter la compréhension du système de transitions. Pour ce faire, nous introduisons une méthode de factorisation des transitions et de choix des sous-états initiaux et finaux.

5.2.3.1 Factorisation des transitions

Nous avons défini (Définition 21, section 5.1.1.3) qu'une transition factorisée sur un état E est équivalente à la même transition recopiée sur chacun des sous-états. De plus, chacune de ces transitions recopiées doit être valide. D'où la condition de factorisation suivante :

Construction de $\mathbb{R}_R$ entre les états-feuille (Algo. 1 et 2)	
Initialisation	*2 OP par état*
Transitions	*4 OP par événement et par couple d'états*
Nouvelles transitions	*4 OP par nouvel événement et par couple d'états*
Construction de $\mathbb{R}_R$ en connaissant $\mathbb{R}_S$	
Initialisation	*2 OP par état* ***déjà initial dans l'abstraction***
Transitions	*4 OP par événement et par couple d'états* ***associés à une transition abstraite valide***
Nouvelles transitions	*4 OP par nouvel événement et par couple d'états*
Construction de $\mathbb{R}_R$ en connaissant $\mathbb{R}_S$ et en prenant en compte les états disjoints	
Disjointure des états	***1 OP par couple d'états hiérarchiques de noms différents***
Initialisation	*2 OP par état déjà initial dans l'abstraction*
Transitions	*4 OP par événement et par couple d'états associés à une transition abstraite valide*
Nouvelles transitions	*4 OP par nouvel événement et par couple d'***états non disjoints***

TAB. 5.1 – Coût maximum, en nombre d'obligations de preuve (OP), de la génération d'un système de transitions associé à un raffinement, en fonction de la méthode de calcul utilisée.

Condition 8 (Cas de factorisation)
Factorisation sur l'origine des transitions *S'il existe une transition $(E_i, (D_i, A_i, ev), F)$ valide pour chaque sous-état E_i de E, alors nous pouvons remplacer ces transitions par $(E, (D', A', ev), F)$ s'il existe D' et A' tels que, pour chaque E_i :*

$$\begin{array}{r}\mathcal{D}ef(E_i) \Rightarrow (D' \Leftrightarrow D_i) \\ \mathcal{D}ef(E_i) \ \wedge \ D_i \Rightarrow (A' \Leftrightarrow A_i)\end{array} \tag{8.1}$$

Factorisation sur l'arrivée des transitions *S'il existe une transition $(E, (D, A_i, ev), F_i)$ valide pour chaque sous-état F_i de F, alors nous pouvons remplacer ces transitions par $(E, (D, A', ev), F)$ s'il existe A' tel que, pour chaque F_i :*

$$\mathcal{D}ef(E) \ \wedge \ D \Rightarrow (A' \Leftrightarrow A_i) \tag{8.2}$$

Rappelons que, même si l'état F ou ses sous-états F_i n'apparaissent pas explicitement dans la condition précédente, ils sont implicitement présents, car les conditions d'atteignabilité A_i sont définies en fonction de l'état d'arrivée des transitions (condition 13, section 4.2.2.1).

Dans le cas de factorisation sur l'état de départ (8.1), D' et A' peuvent être syntaxiquement construits comme une conjonction d'implications qui, à l'état d'origine, associent la condition originale, de la manière suivante :

- $D' \mathrel{\hat{=}} \bigvee_i (\mathcal{D}ef(E_i) \Rightarrow D_i)$
- $A' \mathrel{\hat{=}} \bigvee_i (\mathcal{D}ef(E_i) \ \wedge \ D_i \Rightarrow A_i)$

Notons que si D_i vaut btrue alors le terme $(E_i \Rightarrow D_i)$ aussi vaut btrue. Il n'est donc pas nécessaire de l'ajouter dans la condition construite. De la même manière, si A_i vaut btrue alors le terme $(E_i \ \wedge \ D_i \Rightarrow A_i)$ aussi vaut btrue. Par transitivité, si toutes les transitions composant une factorisation ont leur condition d'atteignabilité à btrue, alors la condition de la transition factorisée vaut aussi btrue.

Dans le cas de factorisation sur l'état d'arrivée (8.2), A' doit être équivalent à chaque A_i sous les

mêmes hypothèses ($\mathcal{D}ef(E) \land D$). Par transitivité, pour pouvoir factoriser des transitions sur leur état d'arrivée, leurs conditions d'atteignabilité A_i doivent être équivalentes deux à deux. Ce second cas de factorisation nécessite donc la vérification d'obligations de preuve, pour établir l'équivalence des conditions d'atteignabilité. La condition A' est alors n'importe laquelle des conditions A_i :

– $A' \hat{=} A_1$ /* *Par exemple* A_1 */

Dans le cas d'une factorisation sur l'état d'arrivée, une alternative à la génération d'obligation de preuve consiste à trouver un critère syntaxique d'équivalence. Par exemple, il suffit que toutes les conditions d'atteignabilité soient réduites à btrue pour qu'elles soient équivalentes. C'est cette solution que nous proposons d'utiliser.

Cette condition est très restrictive, mais elle permet de factoriser une transition en préservant les conditions d'atteignabilité vers chacun des sous-états. Il existe deux autres alternatives consistant soit à préciser, dans la transition factorisée, la condition d'atteignabilité pour chaque sous-état, soit à affaiblir la sémantique des transitions factorisées (Def. 21) en disant qu'il n'est pas possible de retrouver les A_i à partir de A. Nous n'avons pas retenu ces solutions, car, dans le premier cas il faudrait associer de multiples conditions d'atteignabilité à une transition factorisée, tandis que dans le second cas, l'utilisateur ne pourrait pas retrouver les conditions d'atteignabilité des sous-états.

La figure 5.8 résume les cas de factorisation en mettant en évidence la valeur des conditions de franchissement de la transition factorisée construite.

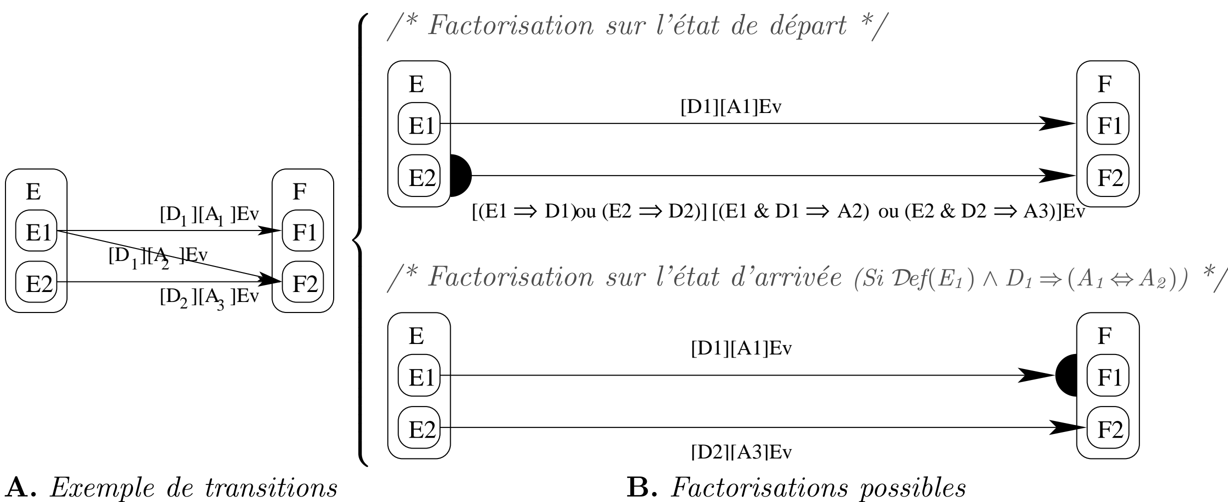


FIG. 5.8 – Calcul des conditions de franchissement d'une factorisation.

L'algorithme suivant permet de factoriser la relation de transition d'un **STEH**. Celui-ci effectue un parcours par niveau de hiérarchie, en commençant par le plus bas. La variable $\mathbb{Q}_{Cour}$ permet de caractériser l'ensemble des états de la hiérarchie courante. Les transitions sont progressivement factorisées et le parcours s'arrête lorsque les racines ont été traitées.

Algorithme 6 (*Factorisation des transitions*) :

Paramètres en entrée : La relation de transition $\mathbb{R}$ et la fonction de hiérarchie $\mathcal{S}up$.

$\mathbb{Q}_{Cour} := \mathcal{S}up[\mathsf{Feuilles}(\mathcal{S}up)]$; /* *Initialement, les états hiérarchiques de plus bas niveau* */
Tant que $\mathbb{Q}_{Cour} \neq \emptyset$:
  Pour chaque état hiérarchique $E \in \mathbb{Q}_{Cour}$ tel que $\mathcal{S}up^{-1}[\{E\}] = \{E_1, .., E_n\}$,
    chaque événement $ev \in \mathbb{E}$ et chaque état $F \in \mathbb{Q}$: /* *Pas de contrainte sur F* */
    /* *Factorisation sur les états de départ* */
    S'il existe une transition valide $(E_i, (D_i, A_i, ev), F)$ depuis chaque sous-état E_i de E, alors :
      $\mathbb{R} := \mathbb{R} \cup \{(E, (\bigvee_{i=1}^{n}(E_i \Rightarrow D_i), \bigvee_{i=1}^{n}(E_i \;\wedge\; D_i \Rightarrow A_i), ev), F)\}$;
      $\mathbb{R} := \mathbb{R} - \{(E_i, (D_i, A_i, ev), F) \mid i \in 1..n\}$
    /* *Factorisation sur les états d'arrivée (Solution syntaxique)* */
    S'il existe une transition valide $(F, (D, \mathsf{btrue}, ev), E_i)$ vers chaque sous-état E_i de E, alors :
      $\mathbb{R} := \mathbb{R} \cup \{(F, D, \mathsf{btrue}, ev), E)\}$;
      $\mathbb{R} := \mathbb{R} - \{(F, (D, \mathsf{btrue}, ev), E_i) \mid i \in 1..n\}$
  $\mathbb{Q}_{Cour} := \mathcal{S}up[\mathbb{Q}_{Cour}]$

Résultat : la relation de transition factorisée $\mathbb{R}$.

L'absence de vérification d'obligations de preuve permet d'éviter les défauts de preuve induits par l'indécidabilité de la logique du premier ordre. Cependant, la factorisation est indirectement dépendante de ces imprécisions, car, si le système de transitions initial n'est pas minimal (Définition 15, section 4.2.2.4), alors les factorisations effectuées conserveront cette imprécision.

5.2.3.2 Choix des sous-états initiaux et finaux

Dans cette section, nous proposons une heuristique de choix des sous-états initiaux et finaux. Nous considérons ici qu'aucune transition n'a été factorisée. Nous verrons dans la section suivante qu'il n'existe quasiment aucune interaction entre ces deux approches.

Étant donné les contraintes de représentation des transitions en relation avec un sous-état initial ou final (Section 5.1.2.2), si une transition atteint un état initial hiérarchique F, alors c'est qu'elle atteint le sous-état initial de F. Tout état hiérarchique initial doit donc contenir un sous-état initial. De la même manière, si un état hiérarchique F contient plusieurs sous-états initiaux, alors il n'est pas possible de savoir lequel est atteint par une transition représentée comme atteignant F. Afin de limiter l'introduction de non-déterminisme externe, nous restreignons donc le nombre de sous-états initiaux à, au plus, un par état hiérarchique. D'où les conditions suivantes sur les sous-états initiaux et finaux :

Condition 9 (Contraintes sur le choix des sous-état initiaux ou finaux)
Chaque état hiérarchique contient au plus un sous-état initial et un sous-état final :

$$\forall(E_1, E_2) \cdot (E_1 \in \mathbb{Q}_{Init} \ \wedge \ E_2 \in \mathbb{Q}_{Init} \ \wedge \ E_1 \neq E_2 \Rightarrow \mathcal{S}up(E_1) \neq \mathcal{S}up(E_2))$$
$$\forall(E_1, E_2) \cdot (E_1 \in \mathbb{Q}_{Final} \ \wedge \ E_2 \in \mathbb{Q}_{Final} \ \wedge \ E_1 \neq E_2 \Rightarrow \mathcal{S}up(E_1) \neq \mathcal{S}up(E_2))$$

Tout état hiérarchique initial contient un sous-état initial :

$$\forall E.(E \in \mathbb{Q}_{Init} \cap \mathsf{ran}(\mathcal{S}up) \Rightarrow \mathcal{S}up^{-1}[\{E\}] \cap \mathbb{Q}_{Init} \neq \emptyset)$$

Tout état hiérarchique final contient un sous-état final :

$$\forall E \cdot (E \in \mathbb{Q}_{Final} \cap \mathsf{ran}(\mathcal{S}up) \Rightarrow \mathcal{S}up^{-1}[\{E\}] \cap \mathbb{Q}_{Final} \neq \emptyset)$$

Un état hiérarchique n'a pas nécessairement qu'un unique sous-état atteint par des transitions externes ou à l'origine de transitions externes. C'est pourquoi, nous avons besoin d'une méthode pour caractériser de manière unique un sous-état initial et un sous-état final dans un état composé.

Nous proposons de choisir comme sous-état initial d'un état hiérarchique celui des sous-états qui est atteint par le plus grand nombre de transitions externes. En cas d'égalité, nous proposons de choisir le candidat à l'origine du plus petit nombre de transitions externes. Ce second critère correspond à l'idée qu'un état initial n'est souvent pas un état final, comme c'est le cas, par exemple, dans les structures de contrôle classiques, telles que les boucles (while, for, exit on, etc). D'autres critères de départage peuvent être choisis ou successivement appliqués. Par exemple, nous pourrions choisir le candidat qui est atteint par le moins de transitions venant de l'intérieur de l'état. Enfin, si aucune règle ne permet de départager des sous-états candidats, alors le choix final peut être non-déterministe. Notons que si un état hiérarchique n'est atteint par aucune transition externe, alors celui-ci n'a pas de sous-état initial.

Des choix similaires peuvent être faits dans le cas des sous-états finaux. Nous ne détaillons donc pas ce deuxième cas. Nous présentons l'algorithme du choix des sous-états initiaux en deux parties : La première introduit quelques macros et décrit le parcours de l'ensemble des états en partant des états-feuille et en remontant progressivement dans la hiérarchie, tandis que la seconde partie décrit le choix du sous-état initial d'un état E donné.

Algorithme 7 (*Construction de* $\mathbb{Q}_{Init}$) :

Paramètres en entrée : $\mathcal{S}up$ et $\mathbb{R}$

Construction de $\mathbb{Q}_{Init}$ **:**

$\mathbb{Q}_{Init} := \emptyset$;
$\mathbb{Q}_{Cour} := \mathcal{S}up[\mathsf{Feuilles}(\mathcal{S}up)]$;
Tant que $\mathbb{Q}_{Cour} \neq \emptyset$ *:*
  Pour chaque $E \in \mathbb{Q}_{Cour}$ *:*
    $\mathbb{Q}_{Init} := \mathbb{Q}_{Init} \cup ChoisitSousEtatInit(E)$
  $\mathbb{Q}_{Cour} := \mathcal{S}up[\mathbb{Q}_{Cour}]$

Résultat : $\mathbb{Q}_{Init}$

Définitions de quelques macros :

/* *États ayant le même super-état que E* */
$Frères(E) \mathrel{\hat=} (\mathcal{S}up^{-1}[\{\mathcal{S}up(E)\}]) - \{E\}$

/* *États extérieurs par rapport à E (Ni parent, ni fils, ni frère)* */
$Ext(E) \mathrel{\hat=} \mathbb{Q} - \mathcal{S}up^{*}[\{E\}] - (\mathcal{S}up^{-1^{*}}[\{\mathcal{S}up(E)\}])$

/* *Ensemble des transitions extérieures atteignant E* */
$DepuisExt(E) \mathrel{\hat=} \{(F, l, E) \mid F \in Ext(E) \ \land \ (F, l, E) \in \mathbb{R}\}$

/* *Ensemble des transitions extérieures partant de E* */
$VersExt(E) \mathrel{\hat=} \{(E, l, F) \mid F \in Ext(E) \ \land \ (E, l, F) \in \mathbb{R}\}$

Sous-fonction principale : Choix du sous-état initial d'un état hiérarchique

Paramètres en entrée : un état hiérarchique E

$ChoisitSousEtatInit(E) \mathrel{\hat=}$ **begin**

/* *Les candidats sont les sous-états feuille de E ou ceux ayant un sous-états initial* */
$C := \mathcal{S}up^{-1}[\{E\}]$;
$C := (C \cap \mathsf{Feuilles}(\mathcal{S}up)) \cup (\mathcal{S}up[\mathcal{S}up^{-1}[C] \cap \mathbb{Q}_{Init}])$;

/* *Critères 1 : Candidats atteints depuis l'extérieur* */
$C := \{E \mid E \in C \ \land \ \mathsf{card}(DepuisExt(E)) > 0\}$;

/* *Critères 2 : Candidats les plus atteints depuis l'extérieur* */
$C := \{E \mid E \in C \ \land \ \forall F \cdot (F \in Frères(E) \ \land \ \mathsf{card}(DepuisExt(E)) \geq \mathsf{card}(DepuisExt(F)))\}$;

/* *Critères 3 : Candidats les moins à l'origine de transitions extérieures* */
$C := \{E \mid E \in C \ \land \ \forall F \cdot (F \in Frères(E) \ \land \ \mathsf{card}(VersExt(E)) \leq \mathsf{card}(VersExt(F)))\}$;

/* *Le candidat est l'un des candidats restants* */
if $C = \emptyset$ **then** $Result := \emptyset$ **else** $Result :\in C$ **end**

end

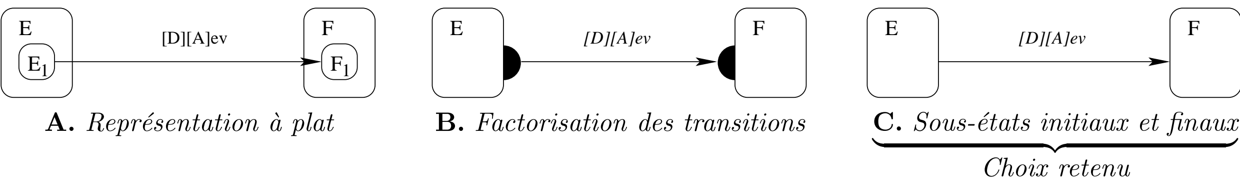
Résultat : un ensemble *Result* vide ou ne contenant qu'un unique sous-état de E

5.2.3.3 Bilan sur la réduction du nombre de transitions externes

Dans cette section, nous avons donné des algorithmes permettant de factoriser les transitions et de choisir des sous-états initiaux et finaux. Les constructions proposées sont purement syntaxiques et ne nécessitent donc pas de preuve.

Cependant, il est possible soit de factoriser les transitions avant de choisir les sous-états initiaux et finaux, soit l'inverse. Les **STEH** construits dans les deux cas sont égaux, à une exception près. Si on commence par factoriser les transitions, alors les états dont toutes les transitions externes ont été factorisées n'ont pas de sous-état initial ou final, puisque toutes les transitions externes ont été factorisées. Si les transitions avaient été factorisées après, alors il y aurait eu des transitions externes et donc des sous-états initiaux et finaux. Nous proposons de factoriser les transitions avant de choisir les sous-états initiaux et finaux.

Cependant, nous proposons de faire un cas particulier pour les états hiérarchiques n'ayant qu'un sous-état (Figure 5.9.A). Au lieu de factoriser toutes les transitions externes d'un tel état (Figure 5.9.B), nous proposons plutôt de les mettre en relation avec le sous-état qui est alors initial et final (Figure 5.9.C). La justification de ce choix vient du fait que, si les sous-systèmes de transitions sont masqués, alors l'utilisation de la factorisation fait penser à l'utilisateur qu'il y a plusieurs sous-états masqués. À l'inverse, les notions de sous-état initial ou final sont associées à l'idée du début et de la fin d'un processus interne.



A. *Représentation à plat* **B.** *Factorisation des transitions* **C.** *Sous-états initiaux et finaux*

Choix retenu

Fig. 5.9 – Cas particulier d'un unique sous-état : plusieurs représentations possibles

Les choix des sous-états initiaux et des sous-états finaux n'ont en revanche aucun effet de bord l'un sur l'autre. L'ordre de leur réalisation est donc sans conséquence. Nous proposons donc l'algorithme suivant :

Algorithme 8 (*Réduction du nombre de transitions externes*) :

Paramètres en entrée : $\mathbb{R}$ sur les états-feuille et $\mathcal{S}up$

1. Factorisation des transitions (Algorithme 6)
2. Aplatissement des transitions factorisées associées à un unique état-feuille :

   Pour toute transition $(E, l, F) \in \mathbb{R}$ *:*

   Si E *est hiérarchique et n'est associé qu'à un seul état-feuille :*
   $\mathbb{R} := \mathbb{R} - \{(E,l,F)\} \cup \{(E_f,l,F) \mid \{E_f\} = \mathsf{Feuilles}(\mathcal{S}up) \cap (\mathcal{S}up^{-1^*}[\{E\}])\}$

   Si F *est hiérarchique et n'est associé qu'à un seul état-feuille :*
   $\mathbb{R} := \mathbb{R} - \{(E,l,F)\} \cup \{(E,l,F_f) \mid \{F_f\} = \mathsf{Feuilles}(\mathcal{S}up) \cap (\mathcal{S}up^{-1^*}[\{F\}])\}$
3. Choix des sous-états initiaux et finaux (Algorithme 7)

Résultat : $\mathbb{R}$ avec d'éventuelles factorisations, $\mathbb{Q}_{Init}$ et $\mathbb{Q}_{Final}$

5.3 Illustration de la construction d'un STEH

Dans cette section nous illustrons l'approche décrite dans la section précédente en introduisant un second niveau de raffinement dans le modèle du canal de communication. Ce raffinement est présenté dans la spécification 5.2, où le texte en gras met en évidence les différences avec le niveau de raffinement précédent. Cette nouvelle description introduit une signature numérique au début de chaque communication. Comme nous ne nous intéressons pas aux aspects cryptographiques du modèle, nous définissons simplement un ensemble de *Signatures* (1), dont un sous-ensemble *SignAutorisees* correspond à celles qui sont autorisées (2). L'événement *Envoyer* transmet donc la taille du message à émettre (*AEnvoyer*), ainsi que sa signature *SignMessage* (3). Si celle-ci n'est pas autorisée, alors la communication est annulée (5). Sinon un acquittement *Ack* est effectué par le nouvel événement *AcquittementSign* (6) et la transmission commence (4).

Spécification 5.2 (*Données du second raffinement*) :

```
    refinement Canal_Communication_V3_R2
    refines Canal_Communication_V3_R
(1) sets Signatures
    constants SignAutorisees
(2) properties SignAutorisees ⊆ Signatures
    variables DansBuffer, AEnvoyer, Ack, SignMessage
    invariant SignMessage ∈ Signatures
              ∧ Ack ∈ BOOL
              ∧ (AEnvoyer = 0 ∧ DansBuffer = 0 ⇒ Ack = false)
              ∧ (SignMessage ∉ SignAutorisees ⇒ Ack = false)
              ∧ (Ack = false ⇒ DansBuffer = 0)
    initialisation
      DansBuffer := 0 || AEnvoyer := 0 || SignMessage :∈ Signatures || Ack := false
    events
      Envoyer ≙ select AEnvoyer = 0 ∧ DansBuffer = 0 then
(3)      AEnvoyer :∈ ℕ1 || SignMessage :∈ Signatures
      end ;
(4)   EnvoyerSuite ≙ select Ack = true ∧ AEnvoyer > 0 ∧ DansBuffer < TailleBuff then
         AEnvoyer := AEnvoyer − 1 || DansBuffer := DansBuffer + 1
      end ;
      Traiter ≙ select Ack = true ∧ DansBuffer > 0 then
         DansBuffer := DansBuffer − 1 || Ack := bool(DansBuffer > 1 ∨ AEnvoyer > 0)
      end ;
(5)   Reset ≙ select Ack = true ∨ (AEnvoyer > 0 ∧ SignMessage ∉ SignAutorisees) then
         DansBuffer := 0 || AEnvoyer := 0 || Ack := false
      end ;
(6)   AcquittementSign ≙ select Ack = false ∧ AEnvoyer > 0 ∧ DansBuffer = 0
                                ∧ SignMessage ∈ SignAutorisees then
         Ack := true
      end
    end
```

Pour construire l'ensemble des comportements de ce raffinement, nous nous basons sur le **STEH** de la figure 5.7 (Section 5.2). Pour mettre en évidence le traitement de la signature, nous divisons l'état **Buffer vide** en énumérant les valuations possibles de la nouvelle variable *Ack*. Les autres états ne sont pas subdivisés, car l'ajout de la signature ne modifie pas le comportement du traitement des messages. L'utilisation des méthodes décrites dans ce chapitre permet de construire le **STEH** de ce second raffinement (Figure 5.10). Afin d'alléger la représentation, nous avons masqué le sous-**STES** des états n'ayant qu'un sous-état.

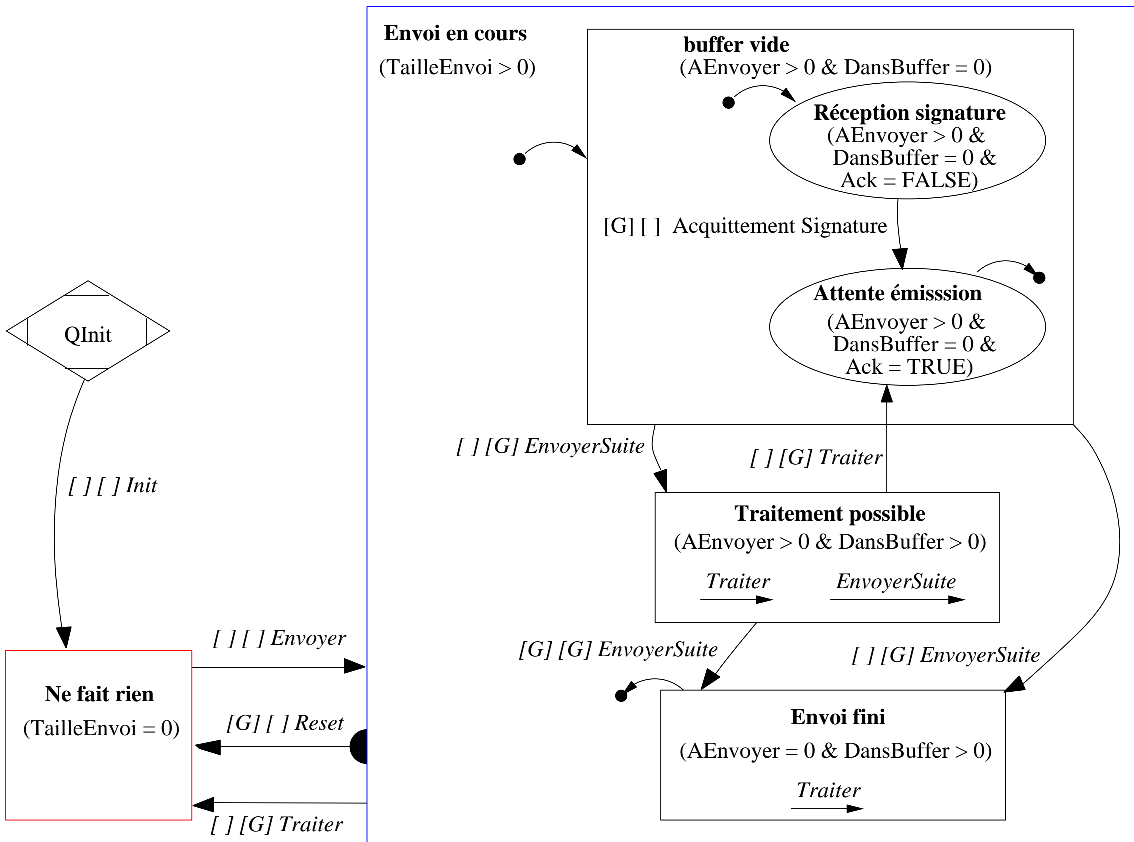


FIG. 5.10 – Représentation des comportements du second raffinement du canal de communication

Contrairement à l'abstraction, *Reset* n'est pas toujours déclenchable depuis **Envoi en cours**. En effet, dans le sous-état **Réception signature**, il est nécessaire que la signature soit incorrecte pour pouvoir annuler la transmission. Comme dans les autres sous-états de **Envoi en cours** *Reset* est déclenchable, ces transitions ont été factorisées.

Enfin, ce **STEH** met en évidence qu'il n'est pas toujours possible de supprimer toutes les transitions externes, puisqu'il en reste une, étiquetée par *Traiter*, et allant de **Traitement possible** vers **Attente émission**. Cependant, l'absence de transitions externes permet souvent de comprendre plus facilement un **STEH**, car il est possible de l'étudier par parties. L'absence ou la présence de transitions externes est liée en même temps au choix de l'espace d'états et à la manière dont a été décrit le système.

5.4 Application au B classique

Dans cette section, nous faisons un bilan des propriétés particulières liées au raffinement d'un modèle B classique et qui permettent de simplifier la construction d'un **STEH** représentant ses comportements.

En B classique, il n'est pas possible de restreindre les pré-conditions par raffinement. En effet, la correction du raffinement d'une opération s'effectue sous l'hypothèse de la pré-condition abstraite, permettant ainsi de définir une pré-condition raffinée plus large. Cependant, on ne considère que les séquences d'appels admises par l'abstraction. Pour caractériser les gardes des événements raffinés, nous devons donc considérer les pré-conditions des opérations de la machine associée au raffinement, comme ceci est défini dans [Abr96b, page 524]. Étant donné que la représentation des données peut être modifiée par raffinement, nous devons donc projeter la pré-condition abstraite de la manière suivante :

$$\begin{array}{ccc} op_S \mathrel{\hat{=}} \textbf{pre } P_S \textbf{ then } T_S \textbf{ end} & & ev_{op_S} \mathrel{\hat{=}} \textbf{select } P_S \textbf{ then } T_S \textbf{ end} \\ \sqsubseteq_L & \leadsto & \sqsubseteq_L \\ op_R \mathrel{\hat{=}} \textbf{pre } P_R \textbf{ then } T_R \textbf{ end} & & ev_{op_R} \mathrel{\hat{=}} \textbf{select } P_R \wedge \exists x \cdot (P_S \wedge L) \textbf{ then } T_R \textbf{ end} \end{array}$$

Ainsi, tout événement déclenchable depuis un état donné est également déclenchable dans le raffinement, depuis la projection de cet état. De plus, si l'on considère qu'aucune substitution n'est miraculeuse[2], alors une condition de déclenchabilité à btrue conserve cette valeur par raffinement. Les transitions associées pourront donc être factorisées sur le super-état correspondant à la projection de E selon l'invariant de liaison L.

Pour ce qui est des conditions d'atteignabilité, leur évolution n'est pas contrainte. En effet, la réduction du non-déterminisme peut introduire des conditions supplémentaires, tandis que le renforcement de l'invariant par raffinement permet d'affaiblir les conditions d'atteignabilité. Toutefois, si un événement est déclenchable depuis un état alors il mène nécessairement quelque part, dans l'un des états atteints par cet événement dans l'abstraction.

En choisissant un espace d'états abstrait basé sur l'intersection des gardes des événements, toutes les conditions de déclenchabilité sont à btrue et le restent par raffinement. La décomposition des états par hiérarchie permet alors de déporter les conditions d'atteignabilité des transitions dans les états.

5.5 Synthèse

Dans ce chapitre, nous avons proposé une méthode de représentation des comportements d'un modèle B construit par raffinement. Nous avons exploité la notion de hiérarchie pour structurer cette représentation. Chacun des niveaux de hiérarchie est associé à un niveau de raffinement et la relation de transition est définie par rapport aux événements raffinés. Nous avons mis en évidence l'aspect changement de représentation des données en hiérarchisant les états. Cela a également per-

[2] En B classique, une substitution est dite miraculeuse si elle n'est pas faisable.

mis de prendre en compte l'aspect affinage du comportement du système, puisque la décomposition des états permet de représenter les comportements du système par une vision plus fine des comportements abstraits. Enfin, la hiérarchie donne un cadre rigoureux pour décomposer un système de transitions et le comprendre par partie. Il est par exemple possible de ne s'intéresser qu'aux comportements par rapport aux états les plus abstraits, ou au contraire se focaliser sur les comportements internes d'un état hiérarchique.

Pour construire ces systèmes de transitions hiérarchiques, nous avons proposé une méthode basée sur les algorithmes présentés dans le chapitre précédent. Elle consiste à construire un système de transitions hiérarchique en modifiant un **STEH** représentant les comportements du niveau de description plus abstrait. La méthode s'organise en trois étapes : construction de l'espace d'états, construction de la relation de transition entre les états-feuille et réduction du nombre de transitions externes. La première étape consiste à projeter la définition des états abstraits sur les variables du raffinement et à ajouter un niveau de hiérarchie supplémentaire. La seconde étape se base sur les comportements autorisés dans l'abstraction et les propriétés du raffinement pour réduire le nombre d'obligations de preuve à vérifier. Enfin, la dernière étape consiste à factoriser les transitions qui peuvent l'être et à choisir les sous-états initiaux et finaux.

Nous avons conclu ce chapitre en mettant en évidence les spécificités du B classique, auquel nous avons étendu l'approche.

Dans la partie suivante nous présentons l'outil *GénéSyst*, mettant en œuvre l'approche présentée ici, et nous décrivons certaines applications qui ont été faites de cette méthode durant cette thèse.

Troisième partie

Outils et exemples d'application

L'outil *GénéSyst* 6

Comme la Hongrie, le monde informatique a une langue qui lui est propre. Mais il y a une différence. Si vous restez assez longtemps avec des Hongrois, vous finirez bien par comprendre de quoi ils parlent.

Dave Barry, Extrait des Chroniques déjantées d'internet

Sommaire

6.1 Comportements d'une spécification abstraite . . . 123
6.1.1 Choix de l'espace d'états . . . 124
6.1.2 Construction de la relation de transition . . . 125
6.1.3 Exemple d'utilisation . . . 129
6.2 Prise en compte du raffinement . . . 130
6.2.1 Choix de l'espace d'états . . . 131
6.2.2 Construction de la relation de transition . . . 131
6.3 Expérimentations . . . 132
6.3.1 Batterie de tests de *GénéSyst* . . . 132
6.3.2 Études de cas . . . 138
6.4 Généralités . . . 140
6.5 Synthèse . . . 141

L'outil *GénéSyst* [MPS04, PS04, Mor04, Moh04, Sto06a] est un programme Java permettant de générer un système de transitions étiquetées à partir d'un modèle B événementiel. Dans ce chapitre, nous présentons cet outil et les techniques qu'il met en œuvre.

6.1 Comportements d'une spécification abstraite

Dans cette section, nous commençons par décrire la méthode de saisie de l'espace d'états qui est implantée dans *GénéSyst*. Nous introduisons ensuite les algorithmes mis en œuvre pour construire la relation de transition, avant de terminer avec un exemple mettant en évidence le gain, en nombre d'obligations de preuve, associé à l'utilisation des optimisations introduites dans *GénéSyst*.

6.1.1 Choix de l'espace d'états

Nous proposons d'utiliser la clause **assertions** du langage B pour décrire l'espace d'états choisi par l'utilisateur. Cet espace est décrit par une disjonction de prédicats représentant chacun le prédicat de définition d'un état. Les assertions sont des propriétés qui doivent se déduire de l'invariant. Si I est l'invariant du système et $\mathcal{B}$ les propriétés sur les constantes, alors l'obligation de preuve associée à la vérification de l'assertion A est la suivante [Abr96b] :

$$I \wedge \mathcal{B} \Rightarrow A$$

Si l'assertion est une disjonction de prédicats $\mathcal{D}ef(E_0) \vee \ldots \vee \mathcal{D}ef(E_n)$, caractérisant les états $E_0, \ldots, E_n$, alors la vérification des obligations de preuve de cette clause permet de garantir la complétude de l'espace d'états par rapport à l'invariant (Condition 1, section 4.2.1.1), puisque l'on montrera $I \wedge \mathcal{B} \Rightarrow \mathcal{D}ef(E_0) \vee \ldots \vee \mathcal{D}ef(E_n)$.

L'exemple 6.1 suivant déclare les deux états $TailleEnvoi > 0$ et $TailleEnvoi = 0$, qui sont utilisés pour construire le système de transitions de la section 6.1.3.

Exemple 6.1 (*Déclaration des états de la figure 6.1*) :

assertions $(TailleEnvoi > 0) \vee (TailleEnvoi = 0)$

Dans le cas de cette assertion, l'obligation de preuve à vérifier est la suivante :

$$\underbrace{TailleEnvoi \in \mathbb{N}}_{\textit{Invariant du modèle}} \quad \Rightarrow \quad \underbrace{(TailleEnvoi > 0) \vee (TailleEnvoi = 0)}_{\textit{Disjonction des états}}$$

Elle est trivialement vraie et permet donc de garantir que toute valuation vérifiant l'invariant est représentée dans l'espace d'états choisi. Les conditions de correction de l'espace d'états et d'absence d'état vide (Conditions 2 et 3, section 4.2.1.1), quant à elles, sont garanties par construction, lors du calcul de la relation de transition. Ces points sont détaillés en section 6.1.2.4.

Enfin, comme nous l'avons vu en section 4.4.3, S. Hamdane et D. Bert [Ham03] ont réalisé un outil permettant de générer, à partir d'une spécification B événementiel, une assertion décrivant un espace d'états. Cet outil, nommé *GénéEtat*, propose cinq méthodes :

1. **Énumération** : technique présentée en section 4.4.2.1
2. **Gardes** : technique présentée en section 4.4.2.2 et basée sur les gardes logiques des événements ;
3. **Gardes et abstraction** : comme *Gardes*, avec la possibilité d'ignorer certaines variables ;
4. **Combinaison des gardes** : utilise l'ensemble des gardes comme espace d'états, en considérant que l'intersection de deux gardes est un état à part entière ;
5. **Combinaison des gardes et abstraction** : comme *Combinaison des Gardes*, avec la possibilité d'ignorer certaines variables.

6.1.2 Construction de la relation de transition

Dans cette section, nous décrivons les optimisations introduites dans l'outil *GénéSyst* et qui simplifient les algorithmes décrits dans les chapitres précédents.

6.1.2.1 Simplifications liées aux conditions de déclenchabilité

Dans les algorithmes décrits précédemment, nous calculons les conditions de déclenchabilité pour chaque événement ev et chaque couple d'états (E, F). Or cette condition ne dépend que de l'état de départ de la transition. Nous proposons donc de ne calculer qu'une fois chaque condition de déclenchabilité. La version simplifiée du calcul de la relation de transition est décrite ci-dessous :

Algorithme 9 (*Factorisation du calcul des conditions de déclenchabilité*) :

Paramètres en entrée : $\mathbb{Q}$ et $\mathsf{Interface}(S)$

Avant initialisation du système : $\mathbb{Q}_{Trait\acute{e}} := \emptyset \; || \; \mathbb{Q}_{Nouv} := \emptyset \; || \; \mathbb{R} := \emptyset$.

Transitions depuis q_{Init} (Recherche des états atteignables par l'initialisation) :
.../* *Inchangé* */

Construction de $\mathbb{R}$ par induction sur $\mathbb{Q}_{Trait\acute{e}}$:
Tant que $\mathbb{Q}_{Nouv} \neq \emptyset$

- $E :\in \mathbb{Q}_{Nouv}$; /* *On choisi un état E atteint mais non traité* */
- Pour chaque ev tel que $ev \in \mathsf{Interface}(S)$
  - **Trouver la condition D de ev depuis E** ;
  - **Si D n'est pas bfalse, alors pour chaque F tel que** $F \in \mathbb{Q} - \{\mathsf{q}_{Init}\}$
    - **Trouver la condition A de la transition** $(E, (D, A, ev), F)$;
    - Si A n'est pas bfalse alors :
      - $\mathbb{Q}_{Nouv} := (\mathbb{Q}_{Nouv} \cup \{F\}) - \mathbb{Q}_{Trait\acute{e}} \; ||$ /* *F est ajouté s'il n'a pas déjà été traité* */
      - $\mathbb{R} := \mathbb{R} \cup \{(E, (D, A, ev), F)\}$
    - /* *Sinon A est* bfalse *et il n'y a pas de transition de* E *vers* F *par ev* */
  - /* *Sinon D est* bfalse *et il n'y a pas de transition depuis* E *par* ev*/
- $\mathbb{Q}_{Trait\acute{e}} := \mathbb{Q}_{Trait\acute{e}} \cup \{E\}$;
- $\mathbb{Q}_{Nouv} := \mathbb{Q}_{Nouv} - \{E\}$

Résultat : $\mathbb{R}$

6.1.2.2 Simplifications liées aux conditions d'atteignabilité

Dans certains cas, il est également possible de simplifier le calcul des conditions d'atteignabilité. En effet, nous avons établi (corollaire 1 suivant) que si un événement est déclenchable depuis un état E et qu'il ne peut atteindre qu'un unique état F, alors la condition d'atteignabilité de la transition associée est nécessairement réductible à btrue. Nous avons donc réorganisé les étapes de l'algorithme implanté dans *GénéSyst* en commençant par établir, quels états ne peuvent pas être atteints par un événement ev, depuis un état E. Si aucun état ne peut être atteint, sauf un,

alors son atteignabilité est définie à btrue, sans vérifier d'obligation de preuve. Sinon on calcule les conditions d'atteignabilité de ces transitions. La démonstration de ce corollaire est donnée en annexe B.4.

Corollaire 1 (Simplification des calculs d'atteignabilité) *Si un événement ev est déclenchable dans un état E et qu'il ne peut mener dans aucun autre état que F, alors la condition d'atteignabilité A de la transition $(E, (D, A, ev), F)$ est réductible à* btrue.

6.1.2.3 Simplifications des obligations de preuve (3) et (6)

Dans la section 4.2.2.3, nous avons introduit 3 obligations de preuve pour chacune des deux conditions de franchissement. Nous avons donc caractérisé 4 possibilités pour chacune de ces deux conditions : btrue, bfalse, *conditionné* et *défaut de preuve*. Ces deux derniers cas sont distingués par les obligations de preuve (3) et (6) des tableaux 4.1 et 4.2 (page 78). Cependant, ces deux formules sont quantifiées existentiellement et sont donc difficilement prouvables par des outils automatiques. Il s'ensuit que la quasi-totalité des conditions de franchissement *conditionnées* aboutissent dans le cas *défaut de preuve*. C'est pourquoi nous avons choisi de laisser le choix à l'utilisateur de générer ou non ces obligations de preuve, dans l'objectif de faire la distinction entre les cas *Conditionné* et *Défaut de preuve*. L'algorithme de caractérisation des conditions est alors le suivant :

Algorithme 10 (*Trouver D et A – sans défaut de preuve*) :

Déterminer la condition de déclenchabilité D d'un événement ev depuis un état E :

1. si la formule (2) est établie alors D vaut bfalse. ***FIN. Pas de transition***;
2. sinon, si la formule (1) est établie alors D vaut btrue ;
3. sinon D vaut Garde(ev)

Déterminer la condition d'atteignabilité A d'un état F par ev depuis E :

4. si la formule (5) est établie alors A vaut bfalse. ***FIN. Pas de transition***;
5. sinon, si la formule (4) est établie alors A vaut btrue ;
6. sinon A vaut $\langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$.

Dans ce cas, les défauts de preuve ne sont plus mis en évidence et les transitions conditionnées sont potentiellement des défauts de preuve. Les systèmes générés ainsi sont donc potentiellement non-minimaux si le franchissement de l'une des transitions au moins est conditionné. C'est à l'utilisateur de vérifier manuellement si certaines conditions de franchissement sont réductibles à btrue ou bfalse.

Enfin, si l'utilisateur choisit de générer les obligations de preuve (3) et (6), alors *GénéSyst* ne vérifie qu'une partie de ces formules. En effet, chacune d'entre elles est une conjonction de deux formules quantifiées existentiellement. Comme il est difficile d'en vérifier une, il est d'autant plus dur de réussir à vérifier les deux. Ces formules sont composées des deux parties suivantes : « il existe une valuation vérifiant la condition », mais « toutes ne la vérifient pas ». La première moitié est la plus intéressante, car elle établit la validité de la transition. C'est pourquoi nous proposons de n'implanter que les obligations de preuve ($3''$) et ($6''$) suivantes à la place des formules (3) et (6).

Nous verrons en section 6.1.2.4 que les obligations preuve réellement implantées sont légèrement différentes.

$(3'')$ $\exists x \cdot (\mathcal{D}ef(E) \wedge \mathsf{Garde}(ev))$	$D \hat{=} \mathsf{Garde}(ev)$
$(6'')$ $\exists x \cdot (\mathcal{D}ef(E) \wedge D \wedge \langle \mathsf{Action}(ev)\rangle \mathcal{D}ef(F))$	$A \hat{=} \langle \mathsf{Action}(ev)\rangle \mathcal{D}ef(F)$

Enfin, la suppression de la moitié des formules (3) et (6) n'a pas d'incidence négative sur le résultat. En effet, nous avons vérifié dans la pratique que, seuls les cas de défaut de preuve signalés par la vérification des obligations $(3'')$ et $(6'')$ sont des défauts de preuve. C'est-à-dire que chaque fois qu'un défaut de preuve est survenu dans les cas (2) ou (5), alors les preuves $(3'')$ et $(6'')$ ont également échoué. Ce résultat s'explique par le fait qu'il est plus simple de prouver une formule non quantifiée (i.e. quantifiée universellement) qu'une formule quantifiée existentiellement et par le fait que les formules (2) et $(3'')$, respectivement (5) et $(6'')$, sont très similaires.

Dans la section suivante, nous décrivons la méthode que nous utilisons pour générer et vérifier les obligations de preuve de *GénéSyst*.

6.1.2.4 Utilisation de prouveurs automatiques

Pour vérifier les obligations de preuve générées par *GénéSyst*, nous utilisons le prouveur automatique de l'*AtelierB* ou de *B4free*. Cependant, ceux-ci sont intégrés à des environnements de développement B. Il faut donc leur fournir un composant B dont la preuve de cohérence permettra d'établir les obligations de preuve voulues. Les formules que nous sommes amenés à vérifier sont définies en termes des données du système. C'est pourquoi, nous construisons une machine en y recopiant l'ensemble des clauses de définition de la structure de données[1]. Il suffit alors de rajouter la formule dans la clause **assertions**, pour que la vérification des obligations de preuve produites permette d'établir la formule. Par la suite nous commençons par mettre en évidence les hypothèses que cette méthode apporte sur les états atteints, avant de simplifier le coût associé à la vérification de telles obligations de preuve.

Traitement des états vides ou incorrects par rapport à l'invariant

Dans les chapitres précédents, nous avons fait l'hypothèse que les états sont corrects vis-à-vis de l'invariant. Nous avons alors vu, en section 4.2.2.4, qu'en l'absence de défaut de preuve, les états vides ou incorrects par rapport à l'invariant ne sont pas atteignables depuis un état non vide. De plus, nous avons mis en évidence que si un état vide est atteint, alors l'ordre de vérification des différentes obligations de preuve (algorithme 1) permet de limiter le nombre de transitions (inatteignables) partant de cet état. Il suffit donc que les états de départ considérés pour les transitions soient corrects.

Dans le cas des formules (1), (2), (4) et (5) de la section 4.2.2.3, l'utilisation de la clause **assertions** permet d'ajouter l'invariant en hypothèse. Il s'ensuit que l'on ne considère que les

[1] **sets**, **constants**, **properties**, **variables** et **invariant**

valuations de l'état de départ qui vérifient l'invariant. Les formules ($3''$) et ($6''$), quant à elles, on été modifiées comme suit, afin de ne considérer que les valuations de l'état de départ qui vérifient I.

Formules implantées	Valeur de D si la formule associée est établie
($3''$) $\exists x \cdot (I \wedge \mathcal{D}ef(E) \wedge \mathsf{Garde}(ev))$	$D \mathrel{\hat{=}} \mathsf{Garde}(ev)$
($6''$) $\exists x \cdot (I \wedge \mathcal{D}ef(E) \wedge D \wedge \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F))$	$A \mathrel{\hat{=}} \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(F)$

Ainsi, les conditions de franchissement sont construites à partir d'états de départ nécessairement corrects. Si l'état de départ est vide, alors l'ordre d'évaluation des obligations de preuve permet de limiter l'introduction de transitions non atteignables (Section 4.2.2.4).

Diminution du coût de vérification de la machine construite

Afin de diminuer le temps nécessaire aux outils pour prouver les obligations de preuve, nous changeons, dans *GénéSyst*, les variables en constantes. En effet, la présence de variables nécessite d'inclure une clause d'initialisation. Les obligations de preuve associées à cette clause sont alors nécessairement générées et vérifiées par les outils utilisés, ralentissant d'autant la construction du système de transitions. Le fait de considérer les variables comme des constantes consiste à transformer l'invariant en une clause **properties** et à supprimer la clause d'initialisation. Enfin, les obligations de preuve attendues sont inchangées par cette transformation, puisqu'elles sont faites sous hypothèse de l'invariant et des propriétés des constantes.

L'exemple 6.2 est la machine B générée pour vérifier que, depuis l'état $TailleEnvoi = 0$, l'événement $Envoyer \mathrel{\hat{=}}$ **select** $TailleEnvoi = 0$ **then** $TailleEnvoi :\in \mathbb{N}_1$ **end** est toujours déclenchable.

Exemple 6.2 (*Déclenchabilité de Envoyer depuis l'état* $TailleEnvoi = 0$) :

machine *Canal_De_Communication*
constants *TailleEnvoi*
properties $TailleEnvoi \in \mathbb{N}$
assertions

$$\underbrace{(TailleEnvoi = 0)}_{\text{/* État de départ */}} \Rightarrow \underbrace{\begin{pmatrix} (TailleEnvoi = 0) \wedge \\ \neg(\forall TailleEnvoi_{apres} \cdot (TailleEnvoi_{apres} \in \mathbb{N}_1 \Rightarrow \mathsf{bfalse})) \end{pmatrix}}_{\text{/* Garde logique de l'événement Envoyer */}}$$

end

L'obligation de preuve vérifiée par l'outil est donc :

$$\underbrace{(TailleEnvoi \in \mathbb{N})}_{\text{/* Invariant */}} \wedge \underbrace{(TailleEnvoi = 0)}_{\text{/* État de départ */}} \Rightarrow \underbrace{\begin{pmatrix} (TailleEnvoi = 0) \wedge \\ \neg \begin{pmatrix} \forall TailleEnvoi_{apres} \cdot \\ (TailleEnvoi_{apres} \in \mathbb{N}_1 \Rightarrow \mathsf{bfalse}) \end{pmatrix} \end{pmatrix}}_{\text{/* Garde(Envoyer) */}}$$

ce qui correspond, comme attendu, à l'obligation de preuve permettant d'établir qu'un événement est toujours déclenchable depuis un état (formule (1) du tableau 4.1).

Enfin, d'un point de vue purement syntaxique, les constantes btrue et bfalse doivent être remplacées par des prédicats ayant ces valeurs, tel que $0 = 0$ et $0 = 1$, afin que la machine produite soit syntaxiquement correcte vis-à-vis du langage B.

6.1.3 Exemple d'utilisation

La figure 6.1 est le résultat produit par *GénéSyst* pour la spécification du canal de communication à partir de l'espace d'états de l'exemple 6.1. Pour la génération de cette figure, nous avons utilisé les choix par défaut de l'outil, qui consistent à ne pas générer les obligations preuve $(3'')$ et $(6'')$ et à utiliser le niveau de force 1 (sur 3) du prouveur automatique de *B4free*.

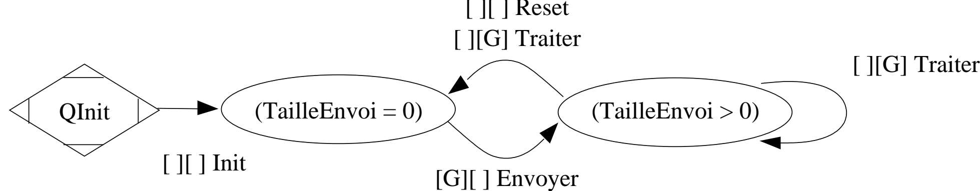


Fig. 6.1 – Système de transitions généré par *GénéSyst* pour la spécification 3.3.

Sur cette figure, la condition de déclenchabilité de l'événement *Envoyer* n'a pas été réduite à btrue, alors qu'elle y est réductible (Cf. exemple 6.2). C'est donc un cas de défaut de preuve. Après diverses comparaisons, il semble que les obligations de preuve de la forme $\neg\forall x \cdot (x \in E \Rightarrow P)$ soient particulièrement difficiles à résoudre pour *B4free* si E est non vide et que $\forall x \cdot (x \in E \Rightarrow \neg P)$ est établie. Nous pensons que le prouveur cherche à instancier la variable quantifiée existentiellement en choisissant un témoin parmi les différentes variables compatibles dans les hypothèses. Or, dans le cas présent, il n'existe pas de variables pouvant être utilisée comme témoin, puisque, pour toute valeur de E, $\neg P$ est vrai.

D'un point de vue représentation graphique, la première version de *GénéSyst* utilisait un format de sortie des **STES** imposant que les états soient étiquetés par des numéros. L'utilisation plus récente de l'outil *GraphViz* et de son format DOT nous a permis d'utiliser des noms d'états plus élaborés. Comme ces noms sont construits de manière automatique, nous avons choisi d'utiliser la description textuelle des prédicats fournis par l'utilisateur. Cela nécessite toutefois que tous les prédicats choisis soient syntaxiquement différents deux à deux. Dans le cas contraire, l'outil *GraphViz*, utilisé pour construire la représentation graphique du système, ne saurait pas distinguer les états. Cette restriction semble néanmoins raisonnablement peu contraignante.

De plus, les représentations produites par *GénéSyst* sont interactives, ce qui permet d'alléger leur encombrement graphique. L'obtention des prédicats caractérisant les conditions de franchissement se fait alors d'un clic de souris sur la transition. Par exemple, la figure 6.2 montre la description fournie à l'utilisateur s'il clique sur la transition de $TailleEnvoi > 0$ vers $TailleEnvoi = 0$. Dans cet exemple, toutes les conditions valent btrue, à l'exception de la condition d'atteignabilité de la

transition associée à *Traiter*, qui est $(TailleEnvoi - 1) = 0$.

L'état de départ : *(TailleEnvoi > 0)*

L'état d'arrivée : *(TailleEnvoi = 0)*

Événement numero 1

Nom de l'événement : *Traiter*

Condition de déclenchabilité : *True*

Condition d'atteignabilité : *Gardée*

(TailleEnvoi - 1) = 0

Événement numero 2

Nom de l'événement : *Reset*

Condition de déclenchabilité : *True*

Condition d'atteignabilité : *True*

FIG. 6.2 – Description produite pour la transition de $TailleEnvoi > 0$ vers $TailleEnvoi = 0$.

Pour finir cette section, nous proposons de faire le bilan du coût, en nombre d'obligations de preuve, de la construction de la figure 6.1. Celle-ci nécessite la vérification de 20 obligations de preuve et prend en moyenne quatre secondes[2], sachant que l'utilisation des algorithmes tels que présentés dans les chapitres précédents aurait nécessité la vérification de 32 obligations de preuve. Nous considérons ici une construction sans caractérisation du cas de défaut de preuve (sans les obligations de preuve ($3''$) et ($6''$)). Le détail de cette comparaison est présenté en tableau 6.1.

Méthode	Initialisation	Transitions de *Envoyer*	Transitions de *Traiter*	Transitions de *Reset*	Total
Algorithmes 1 et 2	*3 OP*	*9 OP*	*11 OP*	*9 OP*	*32 OP*
GénéSyst	*3 OP*	*5 OP*	*7 OP*	*5 OP*	*20 OP*

TAB. 6.1 – Nombre d'obligations de preuve (OP) vérifiées pour générer la figure 6.1.

6.2 Prise en compte du raffinement

L'outil *GénéSyst* permet aussi de construire un système de transitions étiquetées associé à un composant de raffinement. Comme vu au chapitre 5, la construction nécessite de disposer du système de transitions du composant abstrait. Celui-ci peut être soit calculé comme précédemment par *GénéSyst*, soit fourni en paramètre à l'outil. Le **STES** doit alors être décrit dans un format textuel propre à l'outil (appelé format intermédiaire), où chaque ligne correspond à l'état d'une condition d'une transition. Dans cette section, nous commençons par décrire la méthode de saisie de l'espace d'états d'un composant de raffinement. Nous décrivons ensuite les algorithmes mis en œuvre pour construire la relation de transition.

[2] En utilisant le prouveur automatique de l'outil *B4free* sur une station Linux équipée d'un Pentium IV cadencé à 2,8GHz et de 512Mo de mémoire vive.

6.2.1 Choix de l'espace d'états

Dans le cas d'un raffinement, il est nécessaire que chaque projection d'état abstrait soit équivalente à l'union de ses sous-états (Condition 4, section 5.1.1.2). L'utilisateur décrit alors son espace d'états à l'aide d'une conjonction d'équivalences dont le membre de gauche est un état abstrait et le membre de droite est la disjonction des sous-états associés à cet état abstrait :

$$\begin{aligned} & \big(\mathcal{D}ef\,(E_{1_{abstrait}}) \Leftrightarrow \big(\mathcal{D}ef\,\big(E_{(1,1)}\big) \ \vee \ \mathcal{D}ef\,\big(E_{(1,2)}\big) \ \vee \ \ldots \ \vee \ \mathcal{D}ef\,\big(E_{(1,n)}\big)\big)\big) \\ \wedge \ & \big(\mathcal{D}ef\,(E_{2_{abstrait}}) \Leftrightarrow (\ldots)\big) \\ \wedge \ & \ldots \end{aligned}$$

Les obligations de preuve liées à cette assertion permettent là encore de garantir la complétude de l'espace d'états par rapport à l'invariant du raffinement (Condition 7, section 5.2.1). De plus, nous avons choisi d'imposer que chaque état abstrait soit associé à au moins un sous-état dans le raffinement. Il n'est alors pas possible de faire disparaître des états par raffinement. Toutefois, un état peut ne plus être atteignable à travers ce processus, et ne plus apparaître dans la représentation construite.

Dans l'exemple du canal de communication, l'espace d'états décrit par l'assertion suivante permet de construire le système de transition donné en figure 6.3.

Exemple 6.3 (*Définition des états de la figure 6.3*) :

assertions
$((TailleEnvoi = 0) \Leftrightarrow (AEnvoyer = 0 \ \wedge \ DansBuffer = 0)) \ \wedge$
$((TailleEnvoi > 0) \Leftrightarrow ((AEnvoyer > 0) \ \vee \ (AEnvoyer = 0 \ \wedge \ DansBuffer > 0)))$

6.2.2 Construction de la relation de transition

D'un point de vue algorithmique, *GénéSyst* intègre les optimisations suivantes, permettant de réduire le nombre ou la complexité des obligations de preuve générées :

- projection du système de transitions étiquetées abstrait (Section 5.2.2.1) ;
- simplification des conditions de déclenchabilité (Section 6.1.2.1) ;
- simplification des conditions d'atteignabilité (Section 6.1.2.2) ;
- simplification des obligations de preuve d'identification des défauts de preuve (Section 6.1.2.3).

Cependant, dans son état actuel, l'outil ne prend pas en compte l'exclusion des états et ne permet donc pas de diminuer le nombre d'obligations de preuve nécessaires à la construction des transitions associées aux nouveaux événements (Section 5.2.2.2). De plus, bien que l'outil permette de factoriser les transitions et de choisir des sous-états initiaux ou finaux, ces constructions ne sont pas supportées par les sorties graphiques actuelles. Les transitions des systèmes produits sont uniquement entre les états-feuille. De même que pour les composants abstraits, les systèmes de transitions hiérarchiques produits par *GénéSyst* peuvent être interactifs. Il est ainsi possible d'avoir accès, d'un clic de souris, à une description complète de chaque transition et de ses conditions de franchissement.

La figure 6.3 est le **STEH** construit par *GénéSyst* pour le raffinement du canal de communication et l'espace d'états décrit dans la section précédente. De même que pour le cas de l'abstraction, le seul défaut de preuve de cette construction est celui lié à la condition de déclenchabilité de l'événement *Envoyer*. Le nombre d'obligations de preuve nécessaires à *GénéSyst* pour construire

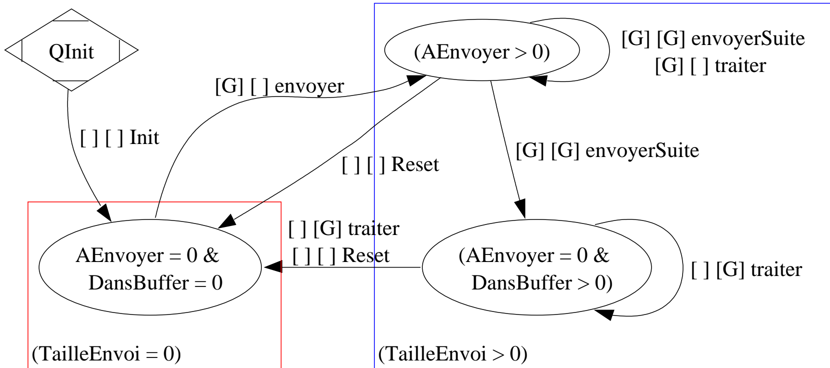


FIG. 6.3 – Représentation du canal de communication générée par *GénéSyst*.

cette figure est résumé dans le tableau 6.2. À titre de comparaison, les deux premières lignes sont les résultats attendus par les approches de projection proposées en section 5.2.2 et ne prenant pas en compte les optimisations présentées en section 6.1.2.

Méthode de projection	**Exclusion des états**	**Initialisation**	**Transitions**	**Nouvelles transitions**	**Total**
Algorithmes 1 et 2	*0 OP*	*2 OP*	*39 OP*	*17 OP*	*58 OP*
Prise en compte de l'exclusion des états	*1 OP*	*2 OP*	*39 OP*	*11 OP*	*53 OP*
Construction implantée dans *GénéSyst*	*0 OP*	*2 OP*	*24 OP*	*9 OP*	*35 OP*

TAB. 6.2 – Nombre d'obligations de preuve pour calculer la figure 6.3

6.3 Expérimentations

Évalué lors de son processus de développement et de validation sur une collection d'exemples jouets, l'outil *GénéSyst* a ensuite été utilisé sur quatre études de cas. Dans cette section nous tirons le bilan de ces différentes utilisations.

6.3.1 Batterie de tests de *GénéSyst*

Dans cette section, nous nous intéressons à un extrait de 6 modèles (présentés à la fin de cette section) faisant partie de la batterie de tests de *GénéSyst*. Ceux-ci peuvent avoir jusqu'à deux niveaux de raffinement. Ces exemples ont été choisis parce qu'ils mettent en œuvre les substitutions et les expressions les plus fréquemment utilisées en B événementiel. De plus, ils permettent d'utiliser toutes les techniques de choix de l'espace d'états proposées (Tableau 6.3) et ils illustrent

les principaux types de spécifications B événementiel (Paramétrées ou pas, avec un nombre fini ou infini d'états, avec ou sans raffinement, déterministes ou pas, avec ou sans introduction de nouveaux événements par raffinement, etc.). Notons que l'écluse a un nombre fini d'états. Sur tous les autres, lorsque la technique d'énumération est utilisée, seules certaines variables du modèle sont énumérées et les états restent donc symboliques.

	Technique de choix des états			
Modèle	***Énumération***	***Gardes***	***Limites***	***Témoin***
Centrale de réservation *(Tiré de [BP03])*		×		
Écluse	×			
Machine à chocolat *(Tiré de [Mil89])*	×		×	
Parking *(Tiré de [PS04])*	×			
Ordonnanceur *(Tiré de [LPU02])*				×
Canal de communication	×	×	×	

TAB. 6.3 – Utilisation des techniques de choix d'espaces d'états dans les tests

Pour chacun de ces modèles, le tableau 6.4 résume le nombre d'événements et d'états choisis. Il donne également le nombre de transitions construites, d'obligations de preuve vérifiées et de défauts de preuve, ainsi que le temps moyen de construction, suivant que les obligations de preuve (3″) et (6″) aient été vérifiées ou non (Distinction ou non des cas *Conditionné* et *Défaut de preuve*). Enfin, notons que la quantité de mémoire vive utilisée est assez constante et varie entre 250Mo et 320Mo. Cet espace mémoire inclus notamment plus de 200Mo de mémoire virtuelle utilisée par Java et environs 20Mo nécessaire au prouveur de *B4free*.

			Avec (3″) **et** (6″)				**Sans** (3″) **et** (6″)			
Nom	**Év**	**Ét**	**Tr**	**OP**	**DP**	**Tps**	**Tr**	**OP**	**DP**	**Tps**
Centrale de réservation	2	3	11	52	15	24s	11	40	2	19s
Écluse	6	4	6	59	0	12s	6	59	0	12s
Machine à chocolat	5	3	10	64	6	16s	10	57	0	15s
Premier raffinement	7	9	16	125	10	34s	16	114	0	26s
Parking	4	3	4	34	2	7s	4	32	0	6s
Premier raffinement	4	4	6	32	2	7s	6	29	0	6s
Ordonnanceur	6	4	16	121	15	35s	16	106	0	31s
Canal de communication	3	2	4	23	2	4s	4	20	1	4s
Premier raffinement	4	4	12	64	9	18s	12	54	1	15s
Second raffinement	5	5	14	74	13	26s	14	62	1	20s
Canal de communication avec énumération des états	3	102	304	21425	102	42,2min	304	21421	1	42,6min
Total				21941	166 (99,2%)			21875	6 (0.1%)	

Év=nombre d'événements / Ét=nombre d'états / Tr=Nombre de transitions / OP=Nombre d'obligations de preuve / DP=Nombre de défauts de preuve / Tps=temps de génération

TAB. 6.4 – Les tests de développement de *GénéSyst* en chiffres

Ces différents exemples confirment que la suppression des obligations de preuve (3″) et (6″) permet de diminuer le nombre de défauts de preuve en atteignant un taux de réussite proche de 100%. Les défauts de preuve restants sont principalement associés à des spécifications contenant une substitution **any**. En effet, le calcul des conditions de franchissement est basé sur le $\mathcal{WP}$ conjugué

de l'action de l'événement, ce qui, dans le cas du **any**, introduit nécessairement une quantification existentielle. Dans les exemples listés dans ce tableau, ce sont tous les défauts de preuve qui sont associés à ce cas particulier. Toutefois, toutes les substitutions **any** n'aboutissent pas nécessairement à un cas de défaut de preuve.

Pour finir cette section, nous décrivons chacun de ces modèles avant de présenter les études de cas sur lesquelles *GénéSyst* a été utilisé.

Modèle de la centrale de réservation

Ce modèle, tiré du polycopié d'enseignement du B à l'ENSIMAG [BP03] et présenté en figure 6.4, est composé de deux événements (réserver et libérer) et permet de gérer les réservations d'un ensemble de places numérotées. Il utilise de l'arithmétique simple (+ et -) et ensembliste (union et soustraction), et contient deux types d'éléments pouvant rendre les preuves plus difficiles : des substitutions **any** et la fonction de cardinalité d'un ensemble.

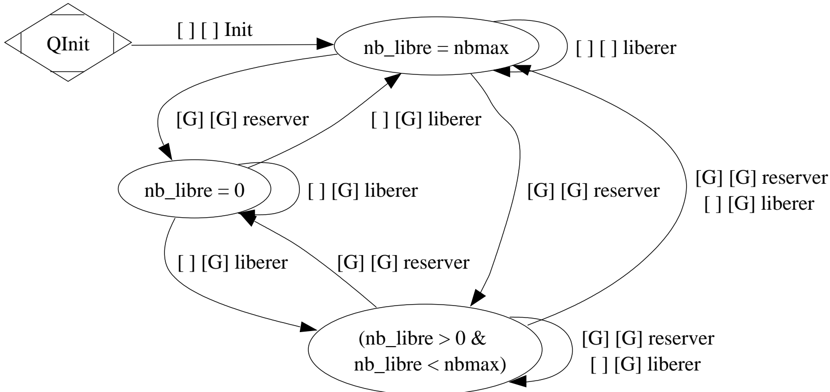


FIG. 6.4 – Comportements du modèle de la centrale de réservation

Modèle de l'écluse

Ce modèle décrit les enchaînements d'actions possibles sur une écluse (figure 6.5). Quatre événements permettent de gérer les portes hautes et basses de l'écluse (ouvrir et fermer) et deux événements permettent de monter ou baisser le niveau de l'eau dans l'écluse. La particularité de ce modèle est qu'il n'utilise que des ensembles énumérés. L'espace d'états choisi est donc une énumération complète des possibilités du modèle.

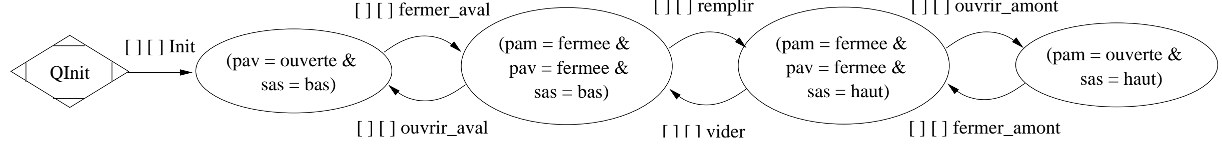


FIG. 6.5 – Comportements du modèle de l'écluse

Modèle de la machine à chocolat

Ce modèle a été initialement introduit dans [Mil89]. Cependant, nous avons paramétré cette machine pour que le crédit maximum accepté par cette machine soit un entier quelconque. Ce modèle

est présenté en figure 6.6. Il décrit les enchaînements autorisés par un automate de distribution de chocolats et proposant deux tailles de boissons : grande ou petite. Il est composé d'une spécification abstraite et d'un raffinement, et contient cinq événements abstraits, permettant de recevoir une ou deux unités d'argent, de servir un grand ou un petit chocolat et de collecter la boisson. Le raffinement introduit le choix de la boisson en fournissant deux événements supplémentaires, pour les grands et les petits chocolats.

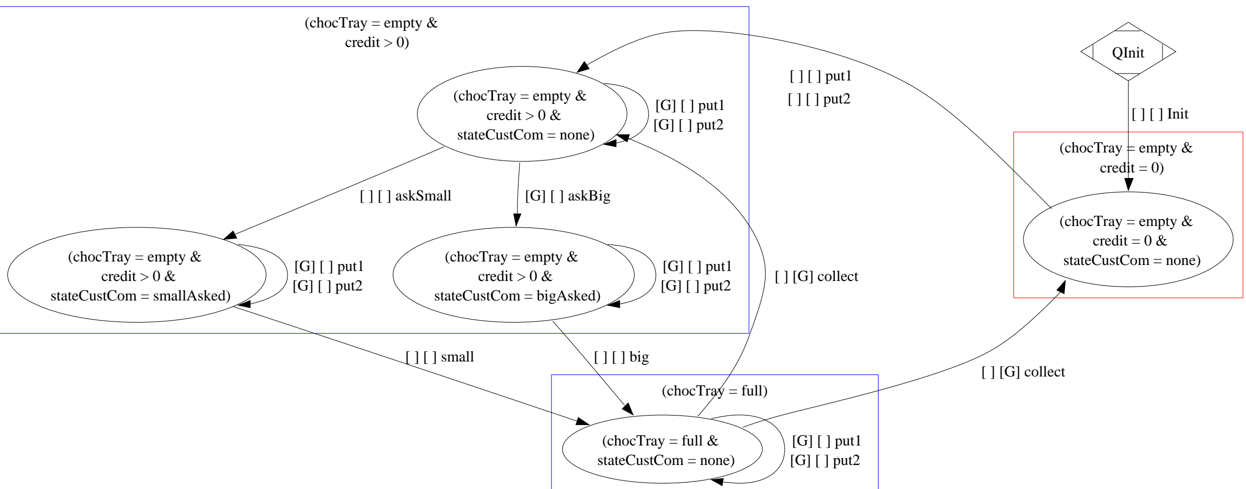


FIG. 6.6 – Comportements du modèle de la machine à chocolat

Modèle du parking

Ce modèle est tiré de [PS04] et décrit les entrées et sorties d'un parking en fonction du nombre de places libres (figure 6.7). Deux événements correspondent à une demande d'entrée ou de sortie, tandis que deux autres événements permettent de valider la demande. La spécification abstraite permet de gérer l'ensemble des places disponibles et de décrire les enchaînements possibles, tandis que le raffinement introduit la notion de feu tricolore à l'entrée du parking. Celui-ci permet de signaler s'il reste des places libres.

Modèle de l'ordonnanceur

Ce modèle est tiré de [LPU02] et décrit la gestion d'un ensemble quelconque de processus. Ceux chargés en mémoire peuvent avoir trois états : actif, prêt ou en attente. Les trois événements présents permettent soit de traiter un processus supplémentaire, soit de préparer un processus qui était en attente, soit d'activer l'un des processus qui est prêt. Le choix de l'espace d'état est basé sur l'exhibition d'un témoin (figure 6.8). Pour mettre plus en évidence son cycle de vie ainsi que les interactions entre les processus, les trois événements ont été décrits deux fois (section 4.4.2.4) : une fois pour ne prendre en compte que le processus témoin et une fois pour ne prendre en compte que les autres. La description produite met alors en évidence le cycle de vie d'un processus P_1 qui initialement n'est pas dans l'ensemble des processus traités. Après son ajout, ce processus peut ensuite avoir tour à tour les trois états : en attente, prêt et actif.

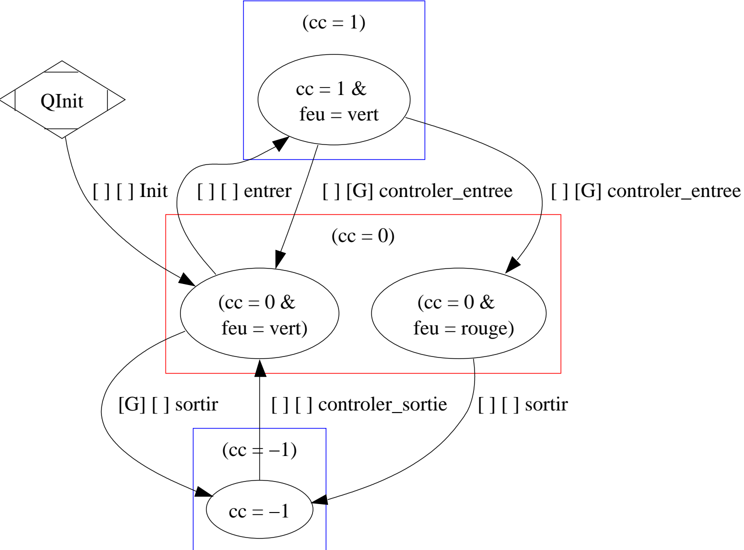


FIG. 6.7 – Comportements du modèle du parking

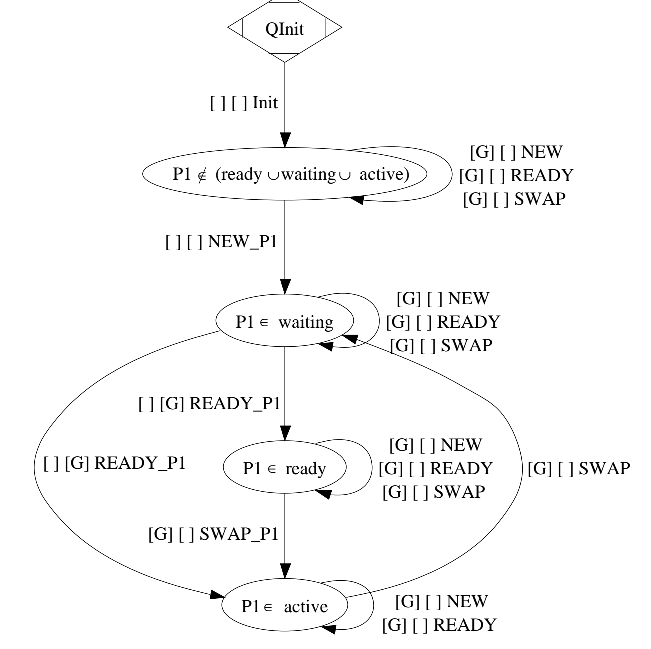


FIG. 6.8 – Comportements du modèle de l'ordonnanceur

Modèle du canal de communication

Ce modèle est celui utilisé tout au long du manuscrit. Cet exemple permet de manipuler des variables à valeur entière et d'introduire de nouveaux événements par raffinement.

Ce modèle est également utilisé pour tester les limites de l'outil. Pour vérifier la résistance du passage à l'échelle de l'outil, nous avons alors utilisé avec un espace d'états plus grand (figure 6.9). Les états sont alors une énumération du nombre des messages restants à envoyer de 0 à

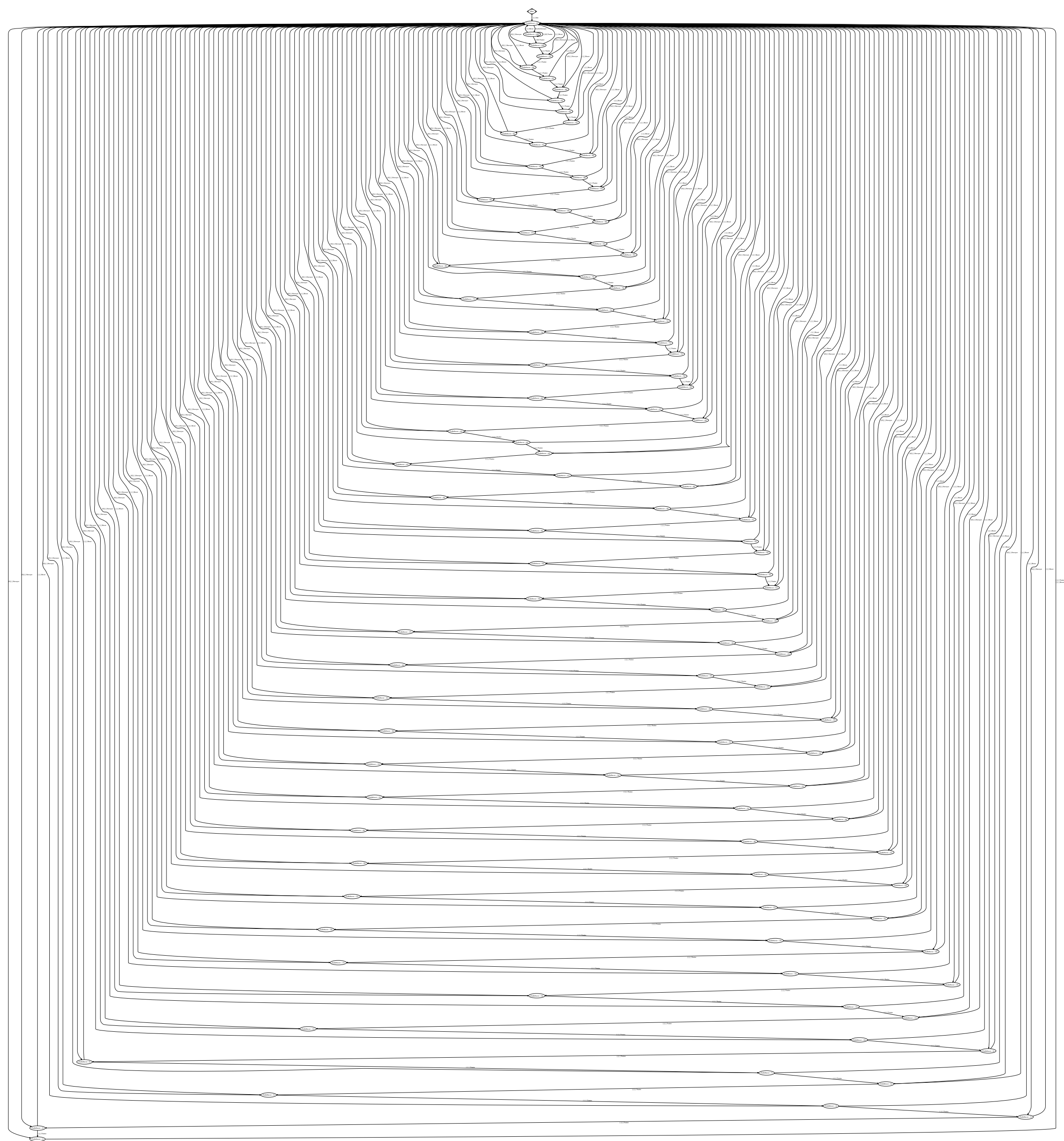


FIG. 6.9 – Comportements du modèle de canal de communication

100 (*TailleEnvoi* $= xx$), plus un état couvrant les autres cas (*TailleEnvoi* > 100). De ce test, nous avons conclu que les premières limites atteintes lors de l'utilisation de *GénéSyst* sont soit celles des outils qu'il utilise, soit celles de la lisibilité des systèmes de transitions produits. Par exemple, lors du calcul de l'ensemble des transitions associées à cet exemple (3 événements et 102 états), l'outil et son interface graphique sont restés stables, mais nous avons rencontré des difficultés pour visualiser le résultat. En effet, l'outil *GraphViz* demande 36 minutes pour générer l'image PNG intégrée à la page HTML produite. La visualisation de cette page avec un navigateur internet[3] nécessite, quant à elle, qu'un segment de 530Mo de mémoire vive soit disponible[4]. La visualisation de la page prend alors plusieurs minutes.

6.3.2 Études de cas

En plus des différents exemples présentés dans la section précédente, l'outil *GénéSyst* a été utilisé sur quatre études de cas : DEMONEY, la spécification du contrôle d'accès au téléchargement de *B4free*, un moniteur réseau et une modélisation d'un aéroport. Ces quatre études de cas sont détaillées ci-dessous.

DEMONEY

Développée par Trusted-Logic, DEMONEY [MM02] est une applette de porte-monnaie électronique pour carte à puce. Sa spécification informelle a servi d'étude de cas notamment dans le projet de l'ACI Sécurité GECCOO, dans lequel nous en avons développé un modèle formel en B. Cette applette étant abondamment détaillée dans le chapitre suivant, nous n'en parlons pas plus ici.

Spécification du contrôle d'accès au téléchargement de *B4free*

Le site web www.b4free.com permet de télécharger l'outil *B4free*. Le contrôle d'accès au téléchargement de l'outil a été modélisé en B événementiel par ClearSy. En effet, cet outil était, initialement, disponible au téléchargement uniquement pour les universitaires et pour les industriels ayant des licences de l'*AtelierB*. Ce modèle est composé de 22 événements et est construit à travers cinq raffinements. L'objectif était d'évaluer la capacité de *GénéSyst* à construire un graphe minimal et son utilité pour l'aide à la documentation et à la compréhension. Nous nous sommes restreints à n'étudier que les trois premiers niveaux de description (la spécification et les deux premiers raffinements), qui correspondent à l'implantation du contrôle d'accès sur le téléchargement. Les niveaux suivants introduisent les droits particuliers associés au webmaster. Huit représentations donnant des points de vue différents sur le système ont été générées, afin de mettre en évidence les différents aspects de ce modèle. Le choix des états a été systématiquement basé sur une exhibition d'un témoin. Les droits de chaque type d'individu et leurs évolutions possibles sont ainsi mis en évidence.

[3] Expérience réalisée sous linux avec le navigateur Firefox 1.5.
[4] Si la machine n'est équipée que de 512Mo de RAM, alors une erreur de dépassement de capacités survient.

Moniteur réseau

L'étude de cas du moniteur réseau [SD05, SD06, SP07] a été développée et publiée dans le cadre d'un partenariat avec des participants au projet POTESTAT [DFG+05]. Dans ce modèle, nous nous sommes intéressés à modéliser une politique de sécurité, puis à la tracer à travers les différents niveaux de raffinement. *GénéSyst* a alors été utilisé comme support de discussion avec des partenaires n'étant pas du domaine des méthodes formelles et a permis de mettre en évidence que des données distinctes dans l'abstraction peuvent être confondues à travers le processus de raffinement. En effet, l'émetteur et le récepteur de chaque message peuvent être précisément décrits et la politique peut être vérifiée exactement. À l'inverse, chaque niveau de raffinement est restreint aux connaissances données par les différentes couches TCP/IP du réseau. Ils sont donc de moins en moins précis, introduisant ainsi des cas conflictuels, où plusieurs valuations abstraites sont associées à une unique description concrète. Dans ce cas, la procédure de décision permettant de statuer sur le respect de la politique possède trois cas de sortie : correct, incorrect ou ne peut pas répondre. Les espaces d'états utilisés ont donc été choisis en fonction de cette propriété. Pour ce faire, nous avons exhibé le cycle de vie d'un témoin représentant un message lambda et nous avons caractérisé, dans les états, son appartenance possible aux différents ensembles en faisant apparaître deux fois le sous-ensemble des valeurs devenues communes.

Modélisation de la sécurité d'un aéroport

Enfin, *GénéSyst* a également été utilisé dans le cadre du projet EDEMOI, traitant de la sécurité dans les aéroports, pour illustrer la documentation du modèle développé et portant sur la réglementation de la sécurité des aéroports. L'objectif de sécurité de ce projet consistait à éviter de faire rentrer des objets dangereux dans les avions. *GénéSyst* a alors permis de générer automatiquement des diagrammes de flots de passagers ou d'avions, exhibant ainsi l'ensemble de leurs comportements possibles. Ici encore, les comportements sont mis en évidence par des témoins ayant tour à tour les différents attributs possibles.

Bilan de l'utilisation de *GénéSyst* sur des études de cas

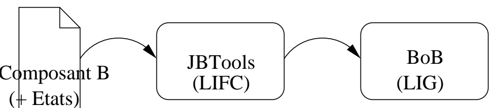
Dans ces différentes études de cas, *GénéSyst* a été utilisé dans le but de faciliter la compréhension et la documentation des différents modèles. Parmi les résultats obtenus, nous pouvons citer la détection de deux erreurs de spécification dans le modèle de DEMONEY. Ces erreurs étaient des problèmes de mauvaise interprétation du cahier des charges et n'ont donc pas pu être détectées par la preuve. L'une d'entre elles était une mauvaise gestion du blocage du porte-monnaie électronique lorsque le nombre maximum d'échecs de saisie du code PIN a été atteint, tandis que l'autre était une mauvaise interprétation des comportements attendus en cas de non respect de la séquentialité de deux événements. Dans le cas du moniteur réseau, *GénéSyst* a été utilisé avec succès durant le développement du modèle et pour illustrer les problèmes de fusion des variables lors de discussions internes. Enfin, les membres du projet EDEMOI ont utilisé *GénéSyst* pour générer l'intégralité des cinq figures présentes dans la documentation du modèle qu'ils ont développé [Ber06], ainsi que pour illustrer ce même modèle dans [BBLV06].

Pour clore cette section sur les expérimentations, notons que les résultats sont, de manière générale, assez proches d'être minimaux. Néanmoins, il est parfois arrivé que certaines spécifications aient été modifiées pour faciliter la vérification automatique des obligations de preuve.

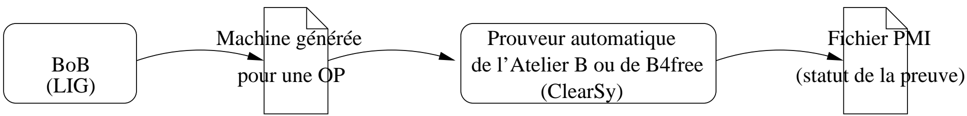
6.4 Généralités

L'outil *GénéSyst* utilise une boîte à outils pour B, développée par l'équipe VASCO (la BoB [Châ01, Sto02]), pour manipuler les composants B, et en particulier les substitutions généralisées et les prédicats. La BoB est notamment utilisée pour calculer les plus faibles pré-conditions d'une substitution pour un prédicat donné. Elle permet également de construire une machine B pour interagir avec les prouveurs utilisés. Cette boîte à outils est donc au cœur de la réalisation de *GénéSyst*. Toutefois, d'autres outils sont également nécessaires à *GénéSyst* :

– La BoB exploite l'analyseur syntaxique *jBTools* [VTH02], développé au LIFC[5], pour charger des composants B.



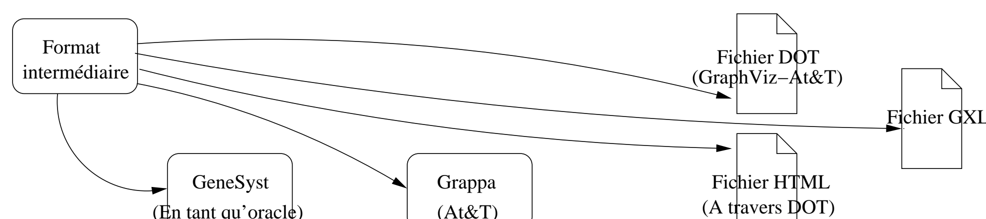
– Les obligations de preuve construites à l'aide de la BoB sont exportées dans des machines B. Celles-ci sont ensuite vérifiées à l'aide du prouveur automatique soit de l'*AtelierB*, soit de *B4free*, tous deux développés par la société ClearSy. Ceux-ci génèrent un fichier PMI donnant le statut des preuves de la machine passée en paramètre ;



– Les systèmes de transitions produits par *GénéSyst* sont exportés dans un format textuel intermédiaire. Celui-ci décrit le résultat de toutes les obligations de preuve générées. Cette sortie peut aussi être donnée en entrée à *GénéSyst* comme un oracle, pour ne pas avoir à calculer de nouveau les obligations de preuve déjà vérifiées ou pour imposer manuellement l'état d'une condition. Ce format est également utilisé pour la représentation graphique du **STEH** calculé. Le programme java Grappa [Moc06], développé dans les laboratoires AT&T, permet de visualiser, depuis l'interface de *GénéSyst*, le système de transitions. D'autres formats de sortie sont également supportés, tels que GXL [HWS00] (Graph eXchange Language) et DOT [GN99, EGK+01]. Ceux-ci sont ensuite utilisables par des outils, tel que *GraphViz*, qui traduisent ces descriptions en représentations graphiques. Ce dernier outil est particulièrement intéressant, car il permet de traduire une description DOT en de nombreux formats de fichiers. En particulier, nous l'utilisons pour générer des sorties dynamiques au format HTML, auto-

[5] Laboratoire d'Informatique de Franche-Comté

risant les interactions avec l'utilisateur par le biais d'hyper-liens sur l'image.



Enfin, au cours du développement de *GénéSyst*, quatre stagiaires ont successivement apporté leur contribution. Chacun d'entre eux a fait avancer ce projet à sa manière :

– Smaïne Hamdane [Ham02], stagiaire de Licence 3, a participé au développement de la première version de *GénéSyst* et a proposé la première structure d'interface avec la BoB ;
– Xavier Morselli [Mor04, MPS04], stagiaire de Master 1, a participé à la vérification du processus de génération des obligations de preuve implanté et à l'optimisation des appels systèmes permettant d'interagir avec le prouveur automatique ;
– Hounayda Mohamed [Moh04], stagiaire de MIAGE 2, a développé la première version de l'interface graphique ;
– Évelyne Altariba, stagiaire de seconde année d'ENSIMAG[6], a mené une étude prospective sur la prise en compte de la modularité dans *GénéSyst*, en se basant sur les travaux de Pierre Bontron et Marie-Laure Potet [BP00]. Ce dernier point n'est pour l'instant pas encore pris en compte dans *GénéSyst*, bien que la BoB intègre maintenant cette fonctionnalité.

6.5 Synthèse

Dans ce chapitre, nous avons présenté l'outil *GénéSyst* qui met en œuvre les algorithmes présentés dans les chapitres précédents. Pour ce faire, l'utilisateur doit fournir, dans la clause d'assertion, l'ensemble des états associés à chaque niveau de raffinement. Il suffit alors que les composants augmentés de ces assertions soient cohérents (prouvés par l'utilisateur) pour garantir que les ensembles d'états choisis sont complets par rapport à l'invariant.

Nous avons ensuite décrit les optimisations mises en œuvre par l'outil et permettant de réduire le nombre d'obligations de preuve à vérifier. Ce gain a été illustré en présentant un exemple de résultat d'exécution. Une optimisation prometteuse, qui est en cours d'intégration, est la possibilité de rajouter des tactiques utilisateur pour vérifier les obligations de preuve. En effet, nous avons constaté qu'une grande partie des obligations de preuve étant en défaut de preuve pourraient être résolues en utilisant le prouveur de prédicats qui est disponible dans le prouveur interactif de l'*AtelierB* ou de *B4free*.

Nous avons terminé en décrivant les différentes études de cas sur lesquelles *GénéSyst* a été utilisé. Il a permis d'aider à leur compréhension et à documenter les modèles. De plus, dans le cas de DEMONEY, il a servi à valider le respect de propriétés de sécurité. Ce dernier aspect est détaillé dans le chapitre suivant.

[6] École Nationale Supérieure d'Informatique et Mathématiques Appliquées de Grenoble, dépendant de l'INPG.

Enfin, *GénéSyst*est diffusé sous la licence CeCILL[7], et est téléchargeable à l'adresse suivante : http://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/?ZoomSur=Logiciels#Logiciels

[7] Ce :CEA ; C :CNRS ; I :INRIA ; LL :Logiciel Libre (équivalent Français de la licence GNU)

Utilisation de *GénéSyst* au sein du projet GECCOO

7

A computer lets you make more mistakes faster than any invention in human history - with the possible exceptions of handguns and tequila.

Mitch Ratcliffe, Technology Review, April 1992

Sommaire

7.1 Présentation du projet GECCOO **144**
7.2 Introduction aux cartes à puce **146**
7.2.1 Technologies des cartes à puce 146
7.2.2 JavaCard : un OS pour cartes à puce 146
7.2.3 Sécurité des cartes à puce 147
7.3 Étude de cas DEMONEY **148**
7.3.1 Description du modèle 148
7.3.2 Objectifs de sécurité 154
7.4 Vérification de propriétés de sûreté **156**
7.4.1 Vérification de propriétés invariantes 156
7.4.2 Vérification de propriétés décrites par des automates 158
7.4.3 Vérification de propriétés décrites en SEPL 160
7.5 Synthèse **166**

Cette thèse a été partiellement effectuée dans le cadre du projet de l'ACI Sécurité GECCOO. Dans ce chapitre, nous décrivons l'approche de vérification de propriétés que nous avons suivie au sein du projet. Dans un premier temps, nous présentons les objectifs du projet et la manière dont nous nous y sommes intégrés. Nous introduisons ensuite le domaine d'application de l'étude de cas choisie, les cartes à puce, ainsi que l'étude de cas elle-même. Enfin, nous décrivons les techniques de vérification de propriétés comportementales proposées, que nous illustrons sur l'étude de cas.

7.1 Présentation du projet GECCOO

Le projet GECCOO[1] [GEC07] (Génération de code certifié pour des applications orientées objet) vise à proposer des méthodes et des outils pour le développement de programmes orientés objet ayant une forte composante sécurité. Un intérêt particulier a été porté aux programmes embarqués (cartes à puce, terminaux, etc) écrits en JavaCard [BDJ+01] (sous-ensemble de Java).

La méthode mise en avant dans ce projet consiste à décrire des propriétés de sécurité dans des langages de haut niveau, puis de les traduire sous la forme d'annotations pouvant être vérifiées sur le code (en JML [LBR98, LBR99] par exemple). Le langage JML est un langage d'annotations permettant notamment de décrire les pré et post-conditions des différentes méthodes. Le langage JTPL (Java Temporal Pattern Language) [TH02] est une extension de JML qui permet de décrire des propriétés temporelles pouvant être réécrites en JML. Des outils ont été développés pour prendre en compte les différentes phases du développement :

- *JAG* est un outil permettant de réécrire une spécification JTPL en JML ;
- *JML2B* est un outil permettant de vérifier la cohérence d'une spécification JML en utilisant les outils existants pour la méthode B ;
- *JML-TT* est un outil de génération de cas de test permettant de vérifier la correction d'un code Java par rapport à sa spécification JML ;
- *Jack* [BR02] et *Krakatoa* [MPMU04] sont des générateurs d'obligations de preuve permettant de vérifier la correction d'un code Java par rapport à sa spécification JML. Le premier décharge ses obligations de preuve dans le prouveur *Simplify* [DNS05], tandis que le second utilise l'outil *Why* [Fil03], qui permet de décharger chaque obligation de preuve dans plusieurs prouveurs, dont Coq [BC04], *Simplify*, *Ergo* [CCK07] et *haRVey* [RD03].

La figure 7.1 résume les principales approches explorées et l'organisation des outils.

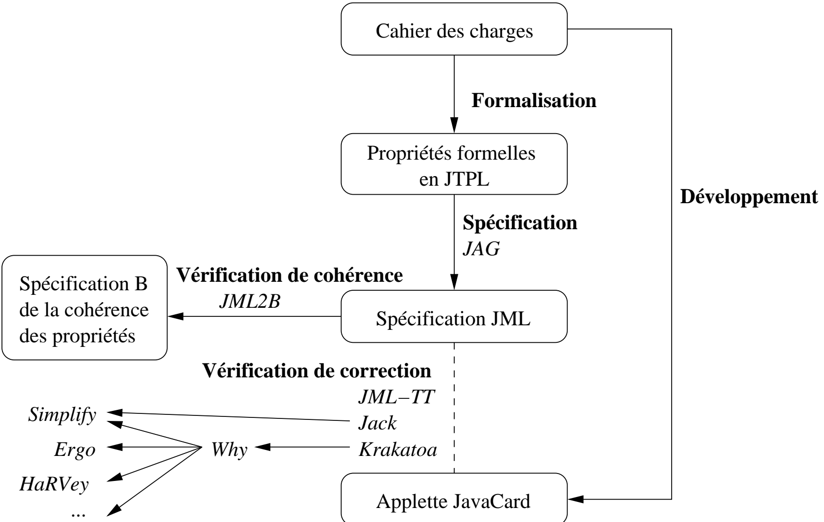


Fig. 7.1 – Principales approches explorées dans GECCOO et outils réalisés ou utilisés

[1] Collaboration entre les équipes TFC (LIFC - Besançon) et VASCO (LIG - Grenoble) et les projets CASSIS (LORIA - Nancy), Everest(INRIA - Sophia Antipolis) et ProVal(LRI/INRIA Futurs - Saclay). http://geccoo.lri.fr/.

Pour notre part, nous nous sommes intéressés à la vérification de propriétés de sécurité en utilisant *GénéSyst*. L'idée est de modéliser le système en B événementiel et de vérifier que la spécification satisfait les propriétés voulues. Ensuite, il est possible, en utilisant les prédicats des états et les conditions de franchissement du système de transitions construit, de générer les pré et post-conditions JML de manière automatique. Cette phase de modélisation peut être réalisée avant, pendant ou après le code Java, car le code JML n'est utilisé pour la vérification qu'une fois le code écrit.

De plus, dans un objectif de certification du logiciel produit, le test du logiciel et la documentation des différentes descriptions sont très importantes. En effet, tous les EALs[2] sont inclusifs. Des campagnes de tests très importantes sont donc nécessaires pour les EALs prenant en compte les modèles formels. De la même manière, les Critères Communs exigent [Com99b, pp. 30] qu'il y aient suffisamment de représentations de la conception, et à un niveau de granularité suffisant, pour démontrer, lorsque cela est demandé, que chaque niveau de raffinement est une instanciation complète et exacte des niveaux supérieurs.

Ainsi, comme notre approche passe par la description d'un modèle B, il est possible d'utiliser les résultats déjà existant autour de ce formalisme, y compris utiliser l'outil *GénéSyst* pour documenter les choix de modélisation. Il est également possible, par exemple, d'utiliser le modèle B comme oracle de test au travers de l'outil *BZ-TT* [ABC+02] ou, pour générer du code C optimisé pour cartes à puce (Résultats du projet BOM [BBP+03]). La figure 7.2 résume les différentes approches que nous avons exploré au sein du projet GECCOO.

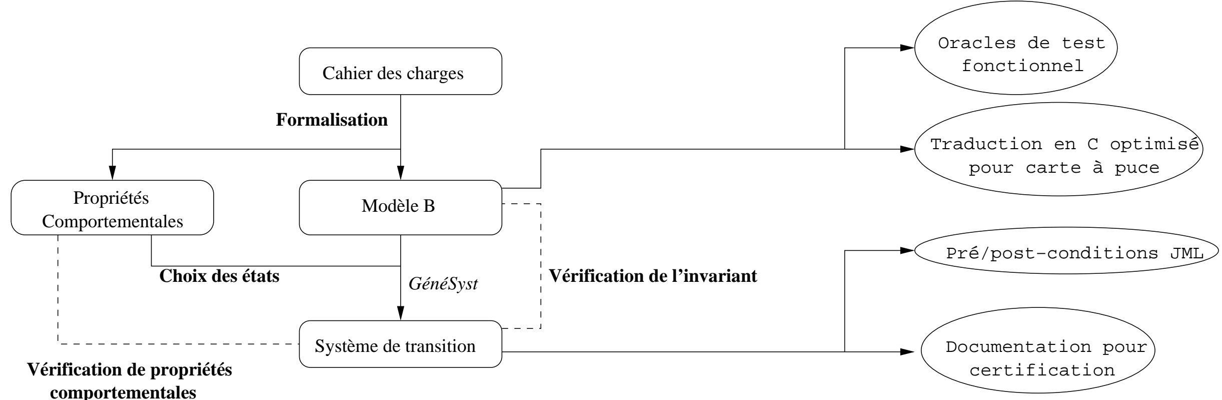


FIG. 7.2 – Approches que nous avons explorées dans GECCOO

Dans ce chapitre, nous proposons de décrire les différentes techniques que nous avons proposé pour vérifier des propriétés sur un modèle B, en utilisant l'approche *GénéSyst*. Ces techniques se basent sur les besoins de sécurité de l'étude de cas DEMONEY. C'est pourquoi, afin de familiariser le lecteur avec les problématiques liées aux cartes à puce, nous commençons par effectuer, en section 7.2, une petite introduction à ce domaine et nous présentons l'étude de cas DEMONEY.

[2]Evaluation Assurance Level

7.2 Introduction aux cartes à puce

Cette section de présentation des cartes à puce est un résumé du rapport technique [Sto06b] réalisé dans le cadre de cette thèse et du projet EDEN.

7.2.1 Technologies des cartes à puce

Inventée en 1974 par le français Roland Moreno, la carte à puce est issue de travaux conjoints entre des laboratoires français, japonais et allemands. Son interface externe est définie avec exactitude par la norme ISO 7816 [Hus01]. Il existe trois variantes de cartes à puce : les cartes à mémoire, les cartes à logiques internes câblée et les cartes à microprocesseur intégré.

Les *cartes à mémoire*, utilisées notamment pour la téléphonie, intègrent un système de fusibles qui peuvent être grillés, permettant ainsi de décrémenter le nombre d'unités présentes sur la carte. Les cartes à *logique interne câblée* sont assimilables aux cartes à *pistes magnétiques* dans le sens où les données stockées ne peuvent pas être modifiées, mais elles s'en distinguent par la masse d'informations stockables et le fait que le contact peut être maintenu indéfiniment. Enfin, les cartes à *microprocesseur intégré* contiennent de la mémoire vive, de la mémoire persistante, un processeur et parfois des co-processeurs (cryptographiques la plupart du temps), leur permettant d'exécuter des programmes et de chiffrer des communications.

C'est évidemment ce troisième type de cartes, assimilables à des ordinateurs, qui va nous intéresser ici. Leur utilisation est très répandue dans la vie de tous les jours, ce qui les rend indispensables et justifie un développement très rapide dans ce secteur de l'industrie. En effet, on trouve notamment dans cette catégorie les cartes de paiement[3] [DCS01], les cartes SIM[4] [DD04] et les cartes d'identification[5] [SG02, Fla05].

Enfin, les communications entre une carte (le serveur) et un terminal (le client) s'effectuent par l'échange de messages APDU[6]. Concrètement, un APDU est un tableau de données permettant d'encapsuler une instruction et ses paramètres ou les résultats de l'exécution d'une instruction.

7.2.2 JavaCard : un OS pour cartes à puce

Comme tout ordinateur, l'utilisation d'un système d'exploitation faisant office d'interface pour l'accès aux ressources de la carte permet de faciliter le développement de logiciels et de s'abstraire de certaines différences architecturales. Le système d'exploitation JavaCard a été lancé en 1997 et est destiné à être utilisé sur tout dispositif à mémoire limitée et en particulier sur des cartes à puce. En 2006, c'est le système d'exploitation pour cartes à puce le plus utilisé dans le monde [Ave05]. Son succès vient notamment du fait que c'est le premier système d'exploitation pour cartes à puce permettant la désinstallation ou l'installation d'applications.

[3] Carte Bleue, Moneo, etc.
[4] Carte d'identification servant pour les téléphones portables.
[5] Carte VITALE, carte d'identité électronique, passeport électronique, etc.
[6] APDU = Application Protocol Data Unit

Dans le système JavaCard, les applications (appelées applettes) sont considérées comme des objets, ce qui permet de gérer plusieurs applications sur une même carte et de désinstaller des programmes. Ces applettes sont écrites en JavaCard, sous-ensemble[7] du langage Java, puis compilées en un bytecode allégé, qui est interprété par une machine virtuelle embarquée dans l'OS : le JCRE[8]. Pour des raisons d'espace mémoire limité, certains éléments de cette machine virtuelle sont supprimés par rapport à Java, comme le Security Manager, le ramasse miettes et le vérificateur de bytecode. JavaCard fournit également un certain nombre d'API[9] permettant de diminuer les connaissances techniques qui sont nécessaires aux développeurs. Par exemple, l'API Open Platform [BWT02] (maintenant renommée GlobalPlatform[10]) est généralement livrée avec JavaCard. Elle est utilisée par DEMONEY et fournit une interface complète de gestion du chiffrage, du déchiffrage et de la signature des messages ainsi que de gestion des clefs.

7.2.3 Sécurité des cartes à puce

La sécurité des cartes à puce est définie de manière empirique sur les attaques matérielles et logicielles [BECN+04, BG01] connues et des principes de bonne conduite. Par exemple, la première règle de base est que toute donnée secrète doit rester dans la carte et être traitée par la carte.

Certaines opérations, telles que l'écriture en mémoire ou l'authentification, nécessitent d'être atomiques, afin de garantir l'intégrité des données stockées ou la correction du protocole d'authentification. En effet, l'écriture en mémoire est un processus assez lent, qui peut permettre de pervertir les données stockées si, par exemple, la carte est arrachée [SL00, SL99] ou l'opération annulée par l'utilisateur. Pour éviter cela, la méthode classique consiste à décomposer les opérations critiques en deux opérations qui sont appelés en séquence [Oes99] : la première définit l'action à effectuer, s'enquière de sa faisabilité et sauvegarde les anciennes données, tandis que la seconde exécute l'action précédemment décrite et signale si l'opération s'est correctement déroulée.

Les cartes à puce étant un système client-serveur, les pré-conditions sont gérées de manière défensives, car toutes les opérations peuvent toujours être appelées. Ainsi, un appel ne respectant pas la pré-condition d'une opération doit être traité et aboutir à l'envoi d'une exception décrivant l'erreur. Celle-ci est émise sous la forme d'un APDU résultat contenant un code erreur, dont la valeur est normalisée. Il s'ensuit que l'on caractérise une séquence d'appels autorisée par le fait qu'aucun APDU d'erreur n'est renvoyé.

Dans la section suivante, nous décrivons une version simplifiée de l'étude de cas DEMONEY. Cet exemple est celui que nous utilisons pour illustrer les sections suivantes.

[7] Notamment pas de types long, double ou float, pas de tableaux multidimensionnels et pas de caractères.
[8] JavaCard Runtime Environment.
[9] Application and Programming Interface
[10] En particulier, les industriels suivants sont les membres les plus influents de ce standard : ActivIdentity, Datacard, France Telecom, Fujitsu, Gemalto, Giesecke & Devrient, Hitachi Ltd, IBM, JCB Co. Ltd, MasterCard Worldwide, NTT Corporation, NXP Seminconductors, Oberthur Card Systems, Renesas, SERMEPA, StepNexus, STMicroelectronics, Sun Microsystems Inc, Thales et Visa International.

7.3 Étude de cas DEMONEY

DEMONEY [Mar02] est un porte-monnaie électronique développé par la société Trusted Logic dans le cadre du projet SecSafe [Sec99]. La spécification informelle de DEMONEY est publique dans la version 0.8 [MM02]. Cette applette, écrite en JavaCard, a toutes les capacités exigées pour un véritable porte-monnaie électronique. Il peut ainsi être débité depuis un terminal dans un magasin, crédité avec des espèces ou depuis un compte bancaire, ou administré depuis un terminal particulier situé dans une zone sécurisée à accès restreint, par exemple une banque. Les messages échangés sont chiffrées si nécessaire et différents niveaux de sécurité sont utilisés en fonction des actions demandées. DEMONEY supporte également les communications avec d'autres applettes sur la carte pour, par exemple, créditer des points sur une carte de fidélité électronique.

Dans cette section, nous commençons par présenter le modèle B réalisé dans le cadre du projet GECCOO avant de mettre en évidence les propriétés de sécurité que nous voulons vérifier.

7.3.1 Description du modèle

Nous avons été amenés à réaliser un modèle B de cette applette. La structuration de cette spécification formelle est présentée en figure 7.3. Cette description ne modélise pas uniquement l'applette en elle-même, mais aussi son environnement d'exécution tels que le JCRE[11], le terminal d'exécution et l'ordonnanceur externe du terminal (les actions demandées par l'utilisateur).

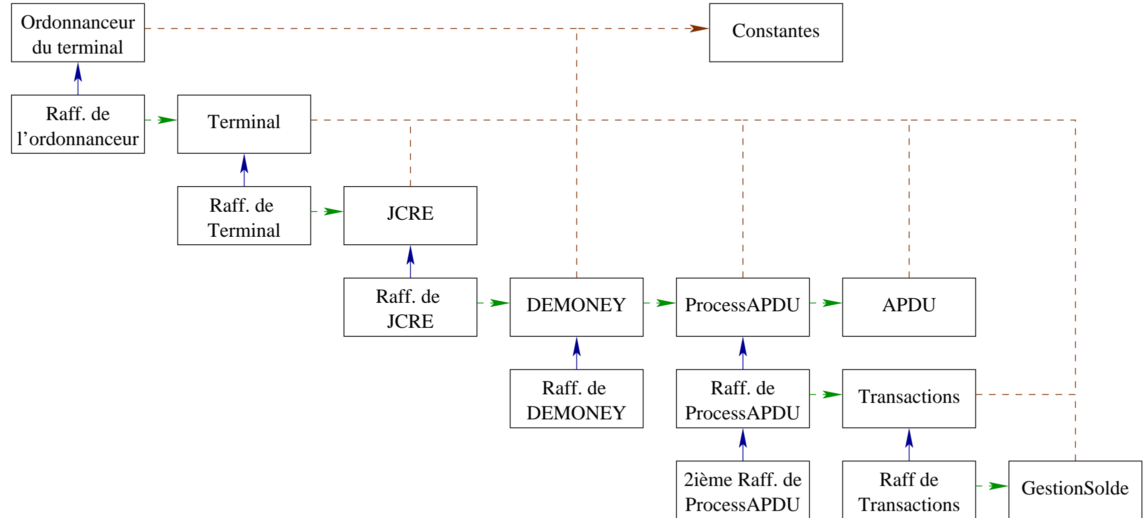


FIG. 7.3 – Structure du modèle de DEMONEY développé dans le cadre du projet GECCOO.

L'applette en elle-même est décomposée en quatre composants (le composant APDU étant mis à part). Le premier, appelé DEMONEY, fournit une interface au JCRE en décrivant les méthodes nécessaires à son installation et son utilisation (tableau 7.1). Les instructions APDU reconnues sont, quant à elles, traitées par le composant ProcessAPDU (tableau 7.2). Parmi ces instructions,

[11] JavaCard Runtime Environment : machine virtuelle embarquée sur carte et interprétant le bytecode JavaCard.

celles permettant de réaliser une transaction (crédit ou débit) ont été déportées dans le composant Transactions. Enfin, la gestion du solde à proprement parler est faite dans le composant Gestion-Solde.

Méthodes	**Rôle**
`install()`	Créé l'instance de l'applette et l'enregistre auprès du JCRE
`process(APDU)`	Reçoit les instructions APDU émises par le terminal
`processData()`	Pour la personnalisation de la carte
`select()`	Sélectionne l'applette comme applette courante (en cours d'exécution)
`deselect()`	Dé-sélectionne l'applette courante.

Tab. 7.1 – Liste et rôle des méthodes que toute applette doit implanter

Instructions APDU	**Rôle**
`Store data`	Personnalisation de l'applette
`Select`	Reçu après la sélection de l'applette
`Initialize update`	Initialise l'authentification mutuelle
`External authentication`	Termine l'authentification
`Put key`	Stockage des clefs cryptographiques
`Pin change / unlock / Verify pin`	Mise à jour / vérification du code PIN
`Initialize transaction`	Initialise un débit ou un crédit
`Complete transaction`	Finit la transaction
`Get data`	Informations publiques du porte-monnaie
`Put Data`	Mise à jour de la configuration
`Read record`	Lecture du fichier log

Tab. 7.2 – Liste et rôle des instructions APDU reconnues par DEMONEY

Afin d'alléger l'exemple, nous illustrons ce chapitre non pas avec l'intégralité du modèle, mais avec une version simplifiée, constituée uniquement d'une spécification abstraite (spécification 7.1) et d'un raffinement (spécifications 7.2-1 et 7.2-2). Le lecteur intéressé par le modèle complet peut toutefois le récupérer à l'adresse suivante :

http://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/?ZoomSur=ProjetGECCOO#ProjetGECCOO

Le modèle allégé n'est décrit que par quatre événements, qui suffisent pour mettre en évidence les différents objectifs de sécurité : *VerifyPin*, *InitializeTransaction*, *CompleteTransaction* et *GetData*. Le dernier événement permet de montrer l'interaction des événements critiques avec les autres événements du système.

Les événements *InitializeTransaction* et *CompleteTransaction* sont modélisés de telle sorte que *CompleteTransaction* ne puisse terminer sans erreur que si *InitializeTransaction* a été appelé juste avant et n'a pas non plus renvoyé d'erreur. La variable d'état *EngagedTrans* est alors utilisée comme sémaphore pour les transactions (Transaction engagée ou non). La variable *Status*, quant à elle, permet de modéliser l'APDU renvoyé par la dernière instruction traitée. Elle contient soit la valeur ISO_{Ok} (tout s'est bien passé), soit une autre valeur correspondant à un code erreur (que nous ne détaillons pas ici). Notons qu'il n'existe pas d'état d'erreur bloquant du système. La seule exception est l'authentification par code PIN, qui peut être interdite si la variable *PinBlocked* vaut `true`. Enfin, bien que nous ne nous intéressons pas à caractériser finement tous les cas d'erreur, nous devons les

prévoir. C'est pourquoi, de nombreuses structures de contrôle non-déterministes sont introduites.

Spécification 7.1 (*Modèle simplifié de DEMONEY*) :

system *Demoney*

sets $STATUSTYPE$
constants ISO_{Ok}
properties $ISO_{Ok} \in STATUSTYPE \wedge STATUSTYPE - \{ISO_{Ok}\} \neq \emptyset$

variables *EngagedTrans*, *PinBlocked*, *Status*
invariant
$Status \in STATUSTYPE$
$\wedge\ EngagedTrans \in \mathsf{BOOL}$
$\wedge\ PinBlocked \in \mathsf{BOOL}$
$\wedge\ (Status \neq ISO_{Ok} \Rightarrow EngagedTrans = \mathsf{false})$

initialisation $Status := ISO_{Ok} \mid\mid EngagedTrans := \mathsf{false} \mid\mid PinBlocked := \mathsf{false}$
events
$GetData \mathrel{\hat=}$ **begin** $EngagedTrans := \mathsf{false} \mid\mid Status :\in STATUSTYPE$ **end** ;

$VerifyPin \mathrel{\hat=}$ **begin**
$EngagedTrans := \mathsf{false} \mid\mid$
if $PinBlocked = \mathsf{true}$ **then** $Status :\in STATUSTYPE - \{ISO_{Ok}\}$
else
choice
$Status :\in STATUSTYPE - \{ISO_{Ok}\}$
$\mid\mid PinBlocked := \mathsf{true}$ } /* *Cas du blocage du code PIN. Il y a nécessairement une erreur* */
or
$Status :\in STATUSTYPE$ } /* *Code PIN non bloqué. Il peut toutefois y avoir une autre erreur* */
end
end
end ;

$InitializeTransaction \mathrel{\hat=}$
any SW **where** $SW \in STATUSTYPE$ **then**
$Status := SW \mid\mid EngagedTrans := \mathbf{bool}(SW = ISO_{Ok})$
end ;

$CompleteTransaction \mathrel{\hat=}$
if $EngagedTrans = \mathsf{false}$ **then** $Status :\in STATUSTYPE - \{ISO_{Ok}\}$
else $Status :\in STATUSTYPE \mid\mid EngagedTrans := \mathsf{false}$
end
end

La spécification 7.2, décomposée en deux parties, est un raffinement de la spécification 7.1. Nous y précisons le nombre d'échecs consécutifs de saisie du code PIN avant blocage (*PinCounter*), le solde courant du porte-monnaie (*Solde*), le type de transaction qui est en cours (*CurTransaction*) et le niveau de sécurité du canal de communication (*Channel*). Les transactions peuvent être soit un débit (*Debit*) de la carte, soit un crédit. Dans ce second cas, nous précisons si le crédit est

effectué par espèces ($Credit$) ou par virement bancaire ($CreditIdent$). Chacune de ces transactions est associée à un niveau de sécurité dans lequel doit au moins être le canal afin de réaliser la transaction. En effet, les différents niveaux de sécurité sont inclusifs. Intuitivement, on a :

$$\text{Droits}(Public_{Lvl}) \subseteq \text{Droits}(Debit_{Lvl}) \subseteq \text{Droits}(Credit_{Lvl}) \subseteq \text{Droits}(CreditIdent_{Lvl})$$

Par exemple, il est suffisant d'entrer le code PIN ($CreditIdent_{Lvl}$) pour effectuer un achat ($Debit_{Lvl}$), alors que la connexion à un terminal de paiement ($Debit_{Lvl}$) ne permet pas de créditer le solde du porte-monnaie ($Credit_{Lvl}$).

Spécification 7.2-1 (*Données du raffinement de DEMONEY*) :

refinement $Demoney_{Raff}$
refines $Demoney$

sets $TRANSACTIONTYPE = \{Debit, Credit, CreditIdent, None\};$
$SECURITYLEVELS = \{Public_{Lvl}, Debit_{Lvl}, Credit_{Lvl}, CreditIdent_{Lvl}\}$

constants $MaxPin, SoldeMax$
properties
$MaxPin \in \mathbb{N}_1$
$\wedge\ SoldeMax \in \mathbb{N}_1$

variables $Status, CurTransaction, Channel, PinCounter, Solde$
invariant
$CurTransaction \in TRANSACTIONTYPE$
$\wedge\ ((EngagedTrans = \mathsf{true}) \Leftrightarrow (CurTransaction \neq None))$ /* Une transaction peut être de différents types */

$\wedge\ Channel \in SECURITYLEVELS$
$\wedge\ ((CurTransaction = Debit) \Rightarrow Channel \in \{Debit_{Lvl}, Credit_{Lvl}, CreditIdent_{Lvl}\})$
$\wedge\ ((CurTransaction = Credit) \Rightarrow Channel \in \{Credit_{Lvl}, CreditIdent_{Lvl}\})$
$\wedge\ ((CurTransaction = CreditIdent) \Rightarrow Channel = CreditIdent_{Lvl})$ /* Si une transaction est en cours, alors le canal est suffisamment sécurisé */

$\wedge\ PinCounter \in 0..MaxPin$
$\wedge\ ((PinCounter = MaxPin) \Leftrightarrow (PinBlocked = \mathsf{true}))$ /* PIN bloqué ssi plus d'essai possible */

$\wedge\ Solde \in 0..SoldeMax$ /* Solde courant */

initialisation
$Status := ISO_{Ok} \parallel Channel := Public_{Lvl} \parallel Solde := 0 \parallel CurTransaction := None \parallel PinCounter := 0$

Les événements, quant à eux, sont décrits dans la spécification 7.2-2. Ils exploitent les nouvelles données, permettant ainsi d'affiner leur comportement. On s'intéresse alors au décompte du nombre d'essais de saisie d'un code PIN et à la conformité du niveau de sécurité par rapport aux transactions effectuées. Enfin, notons que la variable locale tr modélise une partie de l'APDU pris en paramètre et décrit le type de transaction demandé.

Enfin, notons que la variable $Solde$ est écrite mais jamais lue. En effet, son rôle dans cette description est de fixer les évolutions possibles des raffinements à venir, en exhibant, notamment, quelles commandes peuvent être amenées à modifier le solde.

Afin d'illustrer ce modèle, nous proposons de mettre en évidence le cycle de vie du compteur d'essais de saisie du code PIN (figure 7.4). Pour la spécification abstraite, nous choisissons les états

$PinBlocked =$ true et $PinBlocked =$ false. Nous décomposons ensuite ce dernier état par raffinement pour mettre en évidence les cas aux limites de la variable *PinCounter*. Enfin, notons que l'état $PinCounter \in 1..MaxPin - 1$ est possiblement vide, car la constante *MaxPin* peut valoir 1. C'est d'ailleurs l'origine de la condition d'atteignabilité des deux transitions *VerifyPin* partant de l'état $(PinCounter = 0)$.

Spécification 7.2-2 (*Événements du raffinement de DEMONEY*) :

```
events
  GetData ≙ begin
    Status :∈ STATUSTYPE || CurTransaction := None
  end ;                                                      /* Lecture des données publiques */

  VerifyPin ≙ begin
     CurTransaction := None ||                               /* Réinitialisation du sémaphore */
     if  PinCounter = MaxPin then Status :∈ STATUSTYPE − {ISO_Ok}
     else
       choice
         Status := ISO_Ok
          || PinCounter := 0
          || Channel := CreditIdent_Lvl                      /* PIN saisi correct */
       or
         Status :∈ STATUSTYPE − {ISO_Ok}
          || PinCounter := PinCounter + 1                    /* Erreur de saisie du code */
     end
   end
  end ;                                                      /* Vérification du code PIN saisi */

  InitializeTransaction ≙
  any tr where tr ∈ {Debit, Credit, CreditIdent} then
    CurTransaction := None ||
    if (tr = Debit ∧ Channel = Public_Lvl) ∨
        (tr = Credit ∧ Channel ∈ {Public_Lvl, Debit_Lvl}) ∨
        (tr = CreditIdent ∧ Channel ∈ {Public_Lvl, Debit_Lvl, Credit_Lvl})
    then
      Status :∈ STATUSTYPE − {ISO_Ok}                        /* Niveau de sécurité trop faible */
    else
      Status :∈ STATUSTYPE ;
      if Status = ISO_Ok then CurTransaction := tr end       /* Autre erreur ou cas normal */
    end
  end ;                                                      /* Demande de transaction avec le porte-monnaie */

  CompleteTransaction ≙ begin
    if CurTransaction = None then Status :∈ STATUSTYPE − {ISO_Ok}
    else
      CurTransaction := None ; Status :∈ STATUSTYPE ;
      if Status = ISO_Ok then Solde :∈ 0..SoldeMax end
    end
  end                                                        /* Fin d'une transaction */
end
```

Dans la section suivante, nous présentons les objectifs de sécurité de cette applette, ainsi que les propriétés de sécurité qui en découlent. Nous ne nous intéressons pas, pour l'instant, à leur vérification. Ce point sera détaillé dans les sections suivantes.

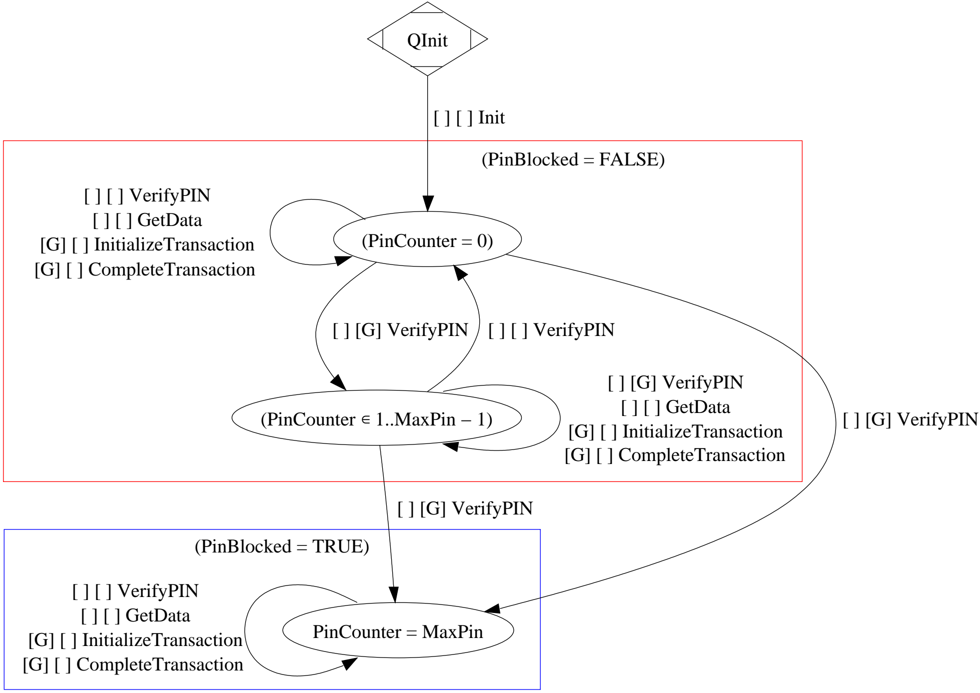


Fig. 7.4 – Exemple des comportements de DEMONEY en fonction du compteur de code PIN.

7.3.2 Objectifs de sécurité

Dans cette section nous décrivons le raisonnement qui, partant des objectifs de sécurité, permet de définir des propriétés de sécurités vérifiables sur le modèle. Les objectifs de sécurité de l'applette DEMONEY ont été décrits dans [MM02, Section 2.2] comme suit :

1. On ne peut pas créer d'argent ;
2. Il est difficile de perdre de l'argent ;
3. Seul le titulaire du compte bancaire dont les coordonnées sont présentes sur la carte peut l'utiliser pour créditer le porte-monnaie électronique.

Pour faciliter leur vérification, ces objectifs ont été décomposés, dans le cadre du projet SecSafe, en propriétés de sécurité simples [MM01]. Parmi elles, nous pouvons distinguer les propriétés cryptographiques et les propriétés non-cryptographiques. Ce sont uniquement ces dernières qui nous intéressent dans la suite, car notre modèle B ne décrit pas les routines cryptographiques. En effet, la cryptographie se base sur des fonctions mathématiques dont la sécurité est liée à la complexité (pour retrouver une clef secrète par exemple) et la méthode B n'est pas adaptée à ce type de problématiques. Ainsi, pour garantir la non création d'argent, il suffit de respecter les propriétés suivantes :

- L'opération de crédit doit récupérer l'argent[12] avant de créditer la carte ;
- L'opération de débit doit débiter la carte avant de donner l'argent (ou de valider l'achat) ;
- Le solde de la carte doit rester positif.

[12] En espèces ou depuis le compte bancaire

La non perte d'argent, quant à elle, n'est pas compatible avec la non création d'argent si l'on prend en compte le risque d'arrachage de la carte durant l'échange. On peut cependant respecter les règles suivantes, qui permettent de restreindre les risques :

– L'opération de Crédit doit vérifier que le crédit est possible avant de récupérer l'argent ;
– L'opération de Débit doit vérifier que le débit est possible avant de débiter la carte ;

Ainsi, pour interdire la création d'argent et en limiter les risques de perte, les transactions sont effectuées en trois étapes, qui sont similaires en cas de crédit ou débit :

1. Vérifier que la transaction est possible (`Initialize transaction`).
2. [Si Crédit] En cas de réussite récupérer l'argent.
3. Effectuer la transaction sur la carte (`Complete transaction`).
4. [Si Débit] En cas de réussite donner l'argent ou valider l'achat.

Les étapes 2 et 4 étant effectués par le terminal et les étapes 1 et 3 par la carte, il est donc nécessaire de faire confiance au terminal, ainsi qu'au canal de communication. On peut donc résumer ces propriétés par les spécifications comportementales suivantes :

– L'instruction `Initialize transaction` termine sans erreur si et seulement si la transaction donnée en paramètre est possible.
– L'instruction `Complete transaction` ne peut aboutir que si elle est immédiatement précédée de l'instruction `Initialize transaction` et que celle-ci s'est terminée sans erreur.
– Seule l'instruction `Complete transaction` peut modifier le solde de la carte. Cette modification étant effectuée si et seulement si l'instruction termine sans erreur.
– Les transactions ne peuvent être effectuées que si le niveau de sécurité du canal de communication est suffisant.

Pour identifier le titulaire de la carte bancaire il est proposé de faire appel à un secret que seul lui connaît : un code PIN. Pour ce faire on doit respecter les règles suivantes :

– Le nombre d'essais du code PIN est incrémenté à chaque échec ;
– Si le nombre maximum d'échecs autorisés est atteint alors il n'est plus possible de s'identifier par ce moyen. Le code PIN est dit bloqué ;
– Le compteur d'échecs est remis à zéro si et seulement si le code est correctement saisi et le code n'est pas bloqué.

Enfin, d'autres propriétés non-cryptographiques sont définies dans [MM01], mais elles font partie des hypothèses sur l'environnement d'exécution et ne sont pas issues des objectifs de sécurité. Par exemple, les calculs arithmétiques doivent être corrects (sans débordement). Le tableau 7.3 résume les propriétés de sécurité que nous proposons de vérifier par la suite.

Dans la section suivante, nous introduisons les différentes méthodes de vérification de propriétés que nous avons expérimentées dans le cadre du projet GECCOO. En particulier, nous nous focalisons sur une approche permettant de vérifier des propriétés comportementales directement sur le système de transitions représentant les comportements du modèle.

(1)	`Complete transaction` doit être précédé d'`Initialize transaction`. *(Celle-ci devant s'être terminée sans erreur).*
(2)	Seule l'instruction `Complete transaction` peut modifier le solde.
(3)	Solde modifié si et seulement si `Complete transaction` termine sans erreur.
(4)	`Initialize transaction` termine sans erreur si la transaction est possible.
(5)	Solde ≥ 0.
(6)	Compteur d'essais du code PIN incrémenté à chaque échec.
(7)	Si compteur d'essais du code PIN atteint sa borne max alors PIN bloqué.
(8)	Compteur PIN remis à zéro si et seulement si code correct et PIN non bloqué.
(9)	Les transactions nécessitent une sécurisation adéquate du canal.

TAB. 7.3 – Propriétés non-cryptographiques issues des objectifs de sécurité de DEMONEY

7.4 Vérification de propriétés de sûreté

Pour vérifier des propriétés sur un automate, il existe un certain nombre de techniques basées sur le model-checking [CS01, Par00]. Classiquement, les propriétés vérifiées par model-checking sont décrites dans des logiques temporelles qui peuvent être arborescentes, comme la CTL, ou linéaires, comme la LTL. Cependant ces formalismes ne permettent généralement pas de décrire un système en même temps en terme de ses états et de ses événements. Dans cette section, nous proposons trois approches syntaxiques de vérification, qui se basent sur une représentation symbolique des comportements du modèle, et qui sont suffisantes pour vérifier les propriétés de sécurité de DEMONEY.

Ces approches permettent de traiter différents types de propriétés. La première approche est une aide à la preuve d'invariant des modèles B événementiel. Elle consiste à mettre en évidence s'il est possible d'atteindre des états ne respectant pas l'invariant (ou plus généralement une propriété invariante sur les données du modèle). La seconde approche, quant à elle, se base sur des propriétés décrites par des automates. Enfin, la troisième approche permet de considérer des propriétés comportementales décrites dans un langage de logique.

7.4.1 Vérification de propriétés invariantes

Cette première méthode permet de vérifier qu'un système ne viole pas son invariant. Elle fournit une aide au concepteur, lorsque la cohérence du système n'a pas pu être établie par la preuve automatique. Le principe est de vérifier qu'aucun état violant l'invariant n'est atteignable depuis l'initialisation. Par extension, cette méthode permet également de fournir une aide au renforcement de l'invariant afin de le rendre *inductif* [CD07, CGK05].

Terminologie. *Un invariant est dit **inductif** s'il est vérifiable par induction : établi par l'initialisation et préservé par les événements. Un invariant n'est donc pas inductif s'il est **trop faible** et qu'il autorise certaines valuations non atteignables, mais à partir desquelles des actions peuvent violer l'invariant. On parle alors de **sous-spécification**. En B, on ne considère que des invariants inductifs.*

Si l'une des preuves de cohérence d'un événement *ev* (Section 3.6.3) n'a pas pu être déchargée automatiquement par le prouveur, alors nous voulons vérifier si c'est un défaut de preuve, une sous-spécification ou une erreur de spécification. La méthode proposée consiste à choisir un espace d'états qui soit complet par rapport au domaine de définition des variables. Il suffit ensuite de construire l'ensemble des comportements du modèle B et de vérifier si seuls les états respectant l'invariant sont atteignables. Pour faciliter l'analyse des résultats, il est important que chaque état soit correct ou incorrect vis-à-vis de l'invariant.

Terminologie. *Nous désignons un état E comme* ***correct*** *par rapport à un invariant I si le prédicat de définition de E vérifie I ($\mathcal{D}ef(E) \Rightarrow I$). E est dit* ***incorrect*** *s'il vérifie le complément de l'invariant ($\mathcal{D}ef(E) \Rightarrow \neg I$). Enfin, si un état n'est ni correct, ni incorrect il est* ***partiellement correct****.*

L'analyse du **STES** produit est alors la suivante :

- ***Si aucun état incorrect n'est atteignable par ev***, alors aucune exécution du système ne permet à cet événement de violer l'invariant. L'échec de la preuve a deux causes possibles :
  - soit l'invariant est trop faible (sous-spécification) ;
  - soit c'est un défaut de preuve.

  Dans le premier cas, on peut automatiquement renforcer l'invariant pour le rendre inductif. En effet, si tous les états atteignables du **STES** vérifient l'invariant, alors l'union de leurs prédicats de définition forme un invariant inductif du modèle. Dans le second cas, l'espace d'états peut être utilisé dans une tactique de preuve par cas, où l'ensemble des états forme l'ensemble des cas. La preuve est automatisable, puisque c'est ainsi qu'a été construit le **STES**.
- ***Si un état incorrect est atteignable par*** *ev* alors :
  - soit l'invariant est trop faible (sous-spécification) ;
  - soit l'invariant est trop fort (sur-spécification) ;
  - soit la spécification de *ev* est fausse.

  Dans les trois cas, l'état de départ de la transition et ses conditions de franchissement permettent de caractériser finement les valuations permettant de violer l'invariant. Cela permet de faciliter la correction de l'invariant ou de l'événement.

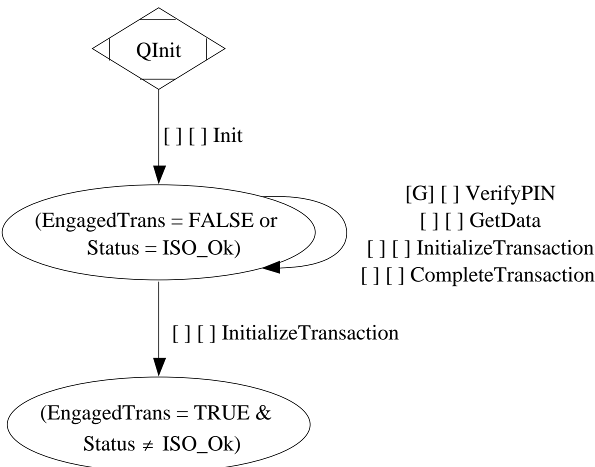
De manière générale, cette approche se base sur le fait que les états utilisés sont généralement décrits par des prédicats simples. Il s'ensuit que le prouveur automatique est plus efficace pour établir l'existence d'une transition entre deux états que pour prouver la cohérence du système. Ainsi, cette méthode est utilisable même si l'*AtelierB* n'arrive pas à décharger toutes les obligations de preuve de cohérence de manière automatique.

Par exemple, dans la figure 7.5.A nous avons modifié le système *Demoney* de telle sorte que l'événement *InitializeTransaction* viole l'invariant. Il est alors possible que cette commande termine en renvoyant un code erreur, mais que la transaction soit quand même engagée. Nous avons choisi de construire l'ensemble des comportements du modèle (figure 7.5.B) en utilisant 2 états : l'invariant ($EngagedTrans = \mathsf{false} \vee Status \neq ISO_{Ok}$) et son complémentaire ($EngagedTrans = \mathsf{true} \wedge Status = ISO_{Ok}$).

Ce faisant, nous mettons en évidence que *InitializeTransaction* permet de violer l'invariant. Comme la condition d'atteignabilité de cette transition est vraie et que l'événement *InitializeTransaction* permet de franchir une autre transition depuis le même état, il est possible de déduire que l'erreur vient probablement d'un choix non-déterministe, comme c'est effectivement le cas.

system *Demoney*
/ Inchangé */*
events
  $GetData \mathrel{\hat{=}}$ */* Inchangé */*
  $VerifyPin \mathrel{\hat{=}}$ */* Inchangé */*
  $InitializeTransaction \mathrel{\hat{=}}$ **begin**
    $Status :\in STATUSTYPE \;||$
    $EngagedTrans :\in$ **bool**
  end ;
  $CompleteTransaction \mathrel{\hat{=}}$ */* Inchangé */*
end

A. *Spécification incorrecte*



B. ***STES*** *exhibant une violation de l'invariant*

FIG. 7.5 – Détection d'une violation d'invariant par utilisation d'un espace d'états incorrect.

Dans le cas où le **STES** généré ne serait pas minimal[13], la méthode est la même, à la seule différence que, si un état incorrect est atteignable par un défaut de preuve, alors on ne peut pas syntaxiquement décider de la cohérence du système. Il faut alors se ramener aux cas décrits précédemment soit en décomposant plus finement l'espace d'états pour faciliter les preuves automatiques, soit en vérifiant l'obligation de preuve de manière interactive.

7.4.2 Vérification de propriétés décrites par des automates

Dans cette section, nous présentons une méthode permettant de vérifier syntaxiquement qu'un modèle B événementiel est correct vis-à-vis d'une propriété de sûreté. Les propriétés décrites sont ici des **STES** n'ayant pas de conditions de franchissement et dont les états n'ont pas de prédicat de définition. Afin d'alléger le texte, nous parlerons d'*automate* d'une propriété, par opposition au **STES** d'un modèle B. Nous définissons la sémantique d'une propriété comme l'ensemble de ses séquences de noms d'états/événements de la forme :

$$\mathsf{q}_{Init} \;;\; Init \;;\; E_1 \;;\; ev_1 \;;\; E_2 \;;\; ev_2 \;;\; \ldots$$

où chaque ev_i est le nom d'un événement et chaque E_i est le nom d'un état.

Nous définissons alors le respect d'une propriété T_P par un **STES** T_M en se basant sur l'inclusion de leurs traces, modulo du bégaiement. En effet, il est possible que des événements du modèle ne soient pas présents dans la propriété. Cependant, de tels événements ne doivent pas permettre de changer le système d'état. Afin de ne pas imposer que les espaces d'états soient égaux, nous

[13] Définition 15, section 4.2.2.4.

utilisons une notion de raffinement structurel[14] basée sur le raffinement B. Le lien entre le système de transitions T_M et la propriété T_P est alors fixé par une fonction $\mathcal{S}up$ associant chaque état de T_M à un état de T_P, formant ainsi une hiérarchie des états. Vérifier le respect de la propriété revient alors à vérifier que T_M est un raffinement de T_P. Pour mettre en œuvre cette approche, nous utilisons la définition de raffinement structurel proposée par D. Cansell, D. Méry et S. Merz [CMM01, Def. 4], dont nous supprimons les notions d'équité et d'ordonnancement. Nous définissons alors :

Définition 24 (Raffinement structurel des systèmes de transitions étiquetées) *Soit une propriété* $T_P \mathrel{\hat{=}} (\mathbb{E}_P, \mathbb{Q}_P, \mathsf{q}_{Init\,P}, \mathbb{L}_P, \mathbb{R}_P)$*, un* ***STES*** $T_M \mathrel{\hat{=}} (\mathbb{V}, \mathbb{E}_M, \mathbb{Q}_M, \mathsf{q}_{Init\,M}, \mathcal{D}ef_M, \mathbb{L}_M, \mathbb{R}_M)$ *et* $\mathcal{S}up$ *une fonction faisant le lien entre les espaces d'états des deux systèmes de transitions* ($\mathcal{S}up \in \mathbb{Q}_M \rightarrow \mathbb{Q}_P$)*. On dit que* T_M *raffine* T_P *selon* $\mathcal{S}up$ *si :*

1. *$\mathcal{S}up$ forme une hiérarchie d'états :*
   - *Les états initiaux sont synchronisés :* $\mathcal{S}up(\mathsf{q}_{Init\,M}) = \mathsf{q}_{Init\,P}$
   - *$\mathcal{S}up$ est un fonction totale et surjective*[a] *:* $\mathcal{S}up \in \mathbb{Q}_M \twoheadrightarrow \mathbb{Q}_P$
2. *Pour toute transition* $(q_M^1, (D, A, ev), q_M^2) \in \mathbb{R}_M$ *du modèle, alors :*
   - *si ev est présent dans* $\mathbb{E}_P$ *alors la transition doit être prévue dans la propriété :*
     $ev \in \mathbb{E}_P \Rightarrow (\mathcal{S}up(q_M^1), ev_M, \mathcal{S}up(q_M^2)) \in \mathbb{R}_P$
   - *si ev n'est pas présent dans* $\mathbb{E}_P$ *alors ce doit être du bégaiement :*
     $ev \notin \mathbb{E}_P \Rightarrow \mathcal{S}up(q_M^1) = \mathcal{S}up(q_M^2)$

[a] Une fonction est **surjective** si chaque élément de son co-domaine a au moins un antécédent.

Étant donnés un modèle M et une propriété T_P, la vérification se décompose en deux étapes :

Algorithme 11 (*Vérification du respect d'une propriété par un modèle*) :

1. Construire T_M, un **STES** du modèle, et définir $\mathcal{S}up$, pour associer chaque état de T_M aux états de T_P ;
2. Vérifier syntaxiquement, pour chaque transition de T_M, si elle est autorisée dans la propriété ou si elle correspond à du bégaiement.

Prenons l'exemple de la propriété *(2)* (tableau 7.3) issue des objectifs de sécurité de DEMONEY :

« *Seule l'instruction* `Complete transaction` *peut modifier le solde.* »

Pour représenter cette propriété, nous proposons de mettre en évidence une valeur quelconque du solde du porte-monnaie et de vérifier qu'elle ne peut être atteinte ou quittée que par l'action de l'événement *CompleteTransaction*. Pour caractériser ces états en termes des variables du modèle, nous utilisons la technique d'exhibition d'un témoin (Section 4.4.2.4). Nous introduisons la constante SS qui n'est pas valuée, mais qui est définie sur le même domaine que *Solde*. En utilisant les états $Solde = SS$ et $Solde \neq SS$, nous mettons donc en évidence que seul *CompleteTransaction*

[14] Raffinement dans lequel la représentation des données est préservée.

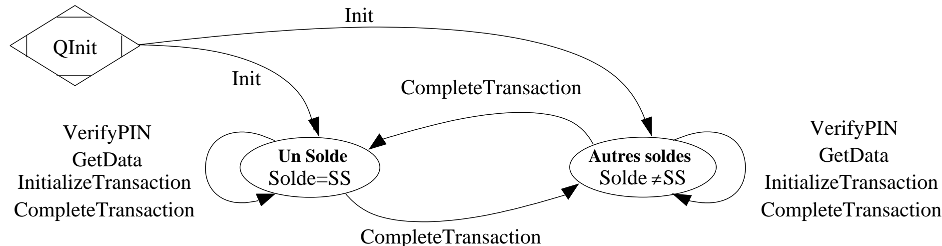
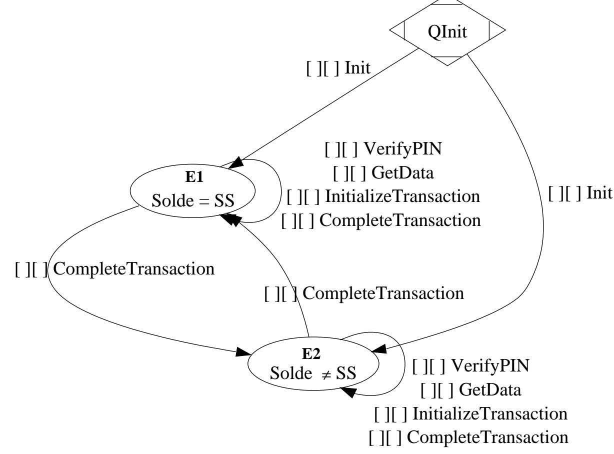


FIG. 7.6 – Représentation de la propriété (2)

peut modifier le *Solde* (figure 7.6). En effet, la constante SS n'étant pas valuée et n'étant pas présente dans la description du modèle, alors les événements autres que *CompleteTransaction* ne peuvent pas modifier solde, car sinon ils pourraient changer le système d'état.

Nous construisons ensuite l'ensemble des comportements de DEMONEY à partir de l'espace d'états de la propriété (figure 7.7.B) en introduisant la nouvelle constante SS non valuée dans le modèle (figure 7.7.A). Enfin, nous établissons que le modèle vérifie la propriété, car chacune des transitions du modèle est soit prévue dans la propriété, soit réflexive et étiquetée par un événement non présent dans $\mathbb{E}_P$.

system *Demoney*
constants SS
properties $SS \in 0..SoldeMax$
/ Le reste du modèle est inchangé */*
end



A. *Modèle de Demoney incluant SS* **B.** *Ensemble des comportements de DEMONEY*

FIG. 7.7 – Comportements de *Demoney* construits à partir de l'espace d'états de la propriété (2)

Enfin, si certaines conditions de franchissement n'ont pas pu être établies (cas de défaut de preuve), alors la méthode de vérification peut quand même être utilisée. Si l'une des transitions associées à ces conditions n'est pas prévue dans la propriété, alors le modèle peut quand même être correct. Pour le décider, il faut soit affiner l'espace d'états, soit évaluer interactivement la condition.

7.4.3 Vérification de propriétés décrites en SEPL

Dans cette section, nous proposons de vérifier des propriétés décrites par des formules logiques. Les langages classiques de logique temporelle (CTL, LTL, etc.) ne permettent d'exprimer des propriétés qu'en termes d'enchaînements d'états. Cependant, nous voudrions pouvoir caractériser des formules basées en même temps sur les états et les événements d'un modèle. K. Trentelman et

M. Huisman [TH02] ont introduit le langage JTPL, qui est une extension au langage JML permettant d'exprimer certaines propriétés par le biais de formules basées sur la logique temporelle et portant sur les attributs et les méthodes du programme Java. Ces propriétés peuvent ensuite être réduites en formules du premier ordre et vérifiées par des outils de vérification associés à JML, tels que *Krakatoa* [MPMU04] ou *Jack* [BR02]. En nous inspirant de ces travaux, nous avons introduit le langage SEPL (States/Events Properties Language) permettant d'exprimer des propriétés portant sur des états et des événements [BPS05]. Ce langage est basé sur quatre primitives qui peuvent être évaluées syntaxiquement sur le système de transitions extrait du modèle B, si celui-ci est minimal.

Dans un premier temps, nous introduisons notre langage. Nous décrivons ensuite comment établir si un modèle B événementiel vérifie une propriété décrite dans ce langage. Enfin, nous terminons en illustrant cette approche sur certaines propriétés de DEMONEY (tableau 7.3).

7.4.3.1 **SEPL** *: un langage de description de propriétés*

De nombreux comportements peuvent être décrits en utilisant la logique du premier ordre et le calcul de plus faible pré-condition. Par exemple, la propriété *(2)* des objectifs de sécurité de DEMONEY « *seule l'instruction CompleteTransaction peut modifier le solde* » peut s'exprimer comme décrit dans l'exemple suivant. Par rapport à la même propriété décrite par des automates dans la section précédante, nous imposons ici qu'il existe au moins une exécution de *CompleteTransaction* qui modifie le solde.

Exemple 7.1 (*Expression de la propriété (2)*) :

Soit I l'invariant du système et $\mathcal{B}$ les propriétés de ses constantes. Cette propriété peut alors s'écrire :

1. Toute instruction APDU, autre que *CompleteTransaction*, laisse le solde inchangé :

$$\bigwedge_{ev \in (\mathsf{Interface}(Demoney) - \{CompleteTransaction\})} (\mathcal{B} \ \wedge \ I \ \wedge \ Solde = SS \Rightarrow [ev](Solde = SS))$$

2. *CompleteTransaction* peut changer le solde :

$$\mathcal{B} \ \wedge \ I \ \wedge \ Solde = SS \Rightarrow \langle CompleteTransaction \rangle (Solde \neq SS)$$

Nous proposons d'abstraire ce type de propriétés logiques en introduisant quatre primitives exprimant soit la capacité d'un événement à être déclenché depuis un état (*Enabled* et *AlwaysEnabled*), soit l'existence d'une transition entre deux états (*Crossable* et *AlwaysCrossable*).

Définition 25 (*Enabled*, *AlwaysEnabled*, *Crossable* et *AlwaysCrossable*) *Soit E_1 et E_2 deux noms d'états, $\mathcal{D}ef(E_1)$ et $\mathcal{D}ef(E_2)$ leurs prédicats de définition respectifs et ev un nom d'événement. On définit alors les propositions suivantes :*

$Enabled(E_1, ev)$	$\hat{=}\ \exists x \cdot (\mathcal{D}ef(E_1) \wedge \mathsf{Garde}(ev))$	*(ev est déclenchable depuis E_1)*
$AlwaysEnabled(E_1, ev)$	$\hat{=}\ \forall x \cdot (\mathcal{D}ef(E_1) \Rightarrow \mathsf{Garde}(ev))$	*(ev est toujours déclenchable depuis E_1)*
$Crossable(E_1, ev, E_2)$	$\hat{=}\ \exists x \cdot (\mathcal{D}ef(E_1) \wedge \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(E_2))$	*(ev peut atteindre E_2 depuis E_1)*
$AlwaysCrossable(E_1, ev, E_2)$	$\hat{=}\ \forall x \cdot (\mathcal{D}ef(E_1) \Rightarrow \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(E_2))$	*(ev peut toujours atteindre E_2 depuis E_1)*

où x représente les variables du système.

Pour décrire les propriétés, nous introduisons le langage SEPL (figures 7.8 et 7.9). Nous définissons alors les propriétés considérées de la manière suivante :

Définition 26 (Propriétés) *Une propriété est un triplet $(P, \mathbb{Q}, \mathbb{E})$ où P est un prédicat décrit en SEPL portant sur un ensemble $\mathbb{Q}$ de noms d'états et un ensemble $\mathbb{E}$ de noms d'événements.*

Nous verrons dans la section suivante que le lien entre un modèle et une propriété est défini par la fonction $\mathcal{S}up$, qui lie les états du **STES** du modèle aux états de la propriété.

Dans le langage SEPL (figure 7.8), les opérateurs binaires $\wedge$, $\vee$ et $\Rightarrow$ sont associatifs à gauche et les priorités sont celles de la logique classique : le $\neg$ est plus prioritaire que le $\wedge$ et le $\vee$, qui sont eux mêmes plus prioritaires que le $\Rightarrow$. Pour ce qui est des formules quantifiées (figure 7.9), nous imposons syntaxiquement que la quantification porte soit sur l'ensemble fini des noms d'états $\mathbb{Q}$, soit sur celui des noms d'événements $\mathbb{E}$.

/* Formules et prédicats */
$F \quad ::= F \ \wedge \ F \ \mid \ F \ \vee \ F \ \mid \ F \Rightarrow F \ \mid \ P$
$P \quad ::= \neg P \ \mid \ (F) \ \mid \ Atome \ \mid \ FQ_{ev} \ \mid \ FQ_q$
/* Atomes */
$Atome \ ::= Enabled(nom_q, nom_{ev})$
$\mid \ Crossable(nom_q, nom_{ev}, nom_q)$
$\mid \ AlwaysEnabled(nom_q, nom_{ev})$
$\mid \ AlwaysCrossable(nom_q, nom_{ev}, nom_q)$
/* Termes de base */
$nom_{ev} \in \mathbb{E}$ et $nom_q \in \mathbb{Q}$

FIG. 7.8 – Syntaxe des formules et des atomes du langage SEPL

Comme les quantificateurs de ce langage portent sur des ensembles finis, ils se réduisent donc à des conjonctions ou des disjonctions de formules. Pour la même raison, il est toujours possible de ramener un ensemble Ens_{ev} ou Ens_q à une définition en extension. Nous proposons donc la définition suivante :

/* Quantification sur les noms d'événements*/	/* Quantification sur les noms d'états */
$FQ_{ev} ::= \forall_{\mathbb{E}}\ nom_{ev} \cdot (nom_{ev} \in Ens_{ev} \Rightarrow F)$	$FQ_q ::= \forall_{\mathbb{Q}}\ nom_q \cdot (nom_q \in Ens_q \Rightarrow F)$
$\mid\ \exists_{\mathbb{E}}\ nom_{ev} \cdot (nom_{ev} \in Ens_{ev}\ \wedge\ F)$	$\mid\ \exists_{\mathbb{Q}}\ nom_q \cdot (nom_q \in Ens_q\ \wedge\ F)$
$Ens_{ev} ::= \mathbb{E}$	$Ens_q ::= \mathbb{Q}$
$\mid\ \{Enum_{ev}\}$	$\mid\ \{Enum_q\}$
$\mid\ \mathbb{E} - \{Enum_{ev}\}$	$\mid\ \mathbb{Q} - \{Enum_q\}$
$Enum_{ev} ::= nom_{ev}$	$Enum_q ::= nom_q$
$\mid\ nom_{ev}, Enum_{ev}$	$\mid\ nom_q, Enum_q$

Fig. 7.9 – Syntaxe des quantificateurs du langage SEPL

Définition 27 (Sémantique des quantificateurs introduits) *Soit $nom_{ev1}, \ldots, nom_{evn}$ des noms d'événements de $\mathbb{E}$, $nom_{q_1}, \ldots, nom_{q_m}$ des noms d'états de $\mathbb{Q}$ et P et Q deux propriétés SEPL portant respectivement sur un événement ou un état. Alors on a :*

$$\forall_{\mathbb{E}}\ nom_{ev} \cdot (nom_{ev} \in \{nom_{ev1}, \ldots, nom_{evn}\} \Rightarrow P(nom_{ev})) \ \hat{=}\ P(nom_{ev1}) \wedge P(nom_{ev2}) \wedge \ldots P(nom_{evn})$$
$$\exists_{\mathbb{E}}\ nom_{ev} \cdot (nom_{ev} \in \{nom_{ev1}, \ldots, nom_{evn}\} \wedge P(nom_{ev})) \ \hat{=}\ P(nom_{ev1}) \vee P(nom_{ev2}) \vee \ldots P(nom_{evn})$$
$$\forall_{\mathbb{Q}}\ nom_q \cdot (nom_q \in \{nom_{q_1}, \ldots, nom_{q_m}\} \Rightarrow Q(nom_q)) \ \hat{=}\ Q(nom_{q_1}) \wedge Q(nom_{q_2}) \wedge \ldots Q(nom_{q_m})$$
$$\exists_{\mathbb{Q}}\ nom_q \cdot (nom_q \in \{nom_{q_1}, \ldots, nom_{q_m}\} \wedge Q(nom_q)) \ \hat{=}\ Q(nom_{q_1}) \vee Q(nom_{q_2}) \vee \ldots Q(nom_{q_m})$$

La sémantique d'une propriété décrite dans le langage SEPL est donnée par sa réécriture en logique du premier ordre, comme décrit dans l'algorithme suivant :

Algorithme 12 (*Validation d'un modèle par rapport à une propriété SEPL*) :

1. Définir chaque état par rapport aux variables du modèle
2. Éliminer les quantificateurs (Définition 27)
3. Évaluer la valeur des atomes par la preuve (Condition 10)
4. Évaluer la formule de logique propositionnelle obtenue, sous hypothèses de l'invariant du modèle

Ce langage permet notamment de spécifier toutes les propriétés descriptibles par les automates de la section précédente. En effet, il est possible de réécrire les automates en SEPL, puisqu'il suffit de conserver l'espace d'états, d'interdire les transitions, qui ne sont pas prévues par l'automate :

$$E \in \mathbb{Q}_P\ \wedge\ F \in \mathbb{Q}_P\ \wedge\ ev \in \mathbb{E}_P\ \wedge\ (E, ev, F) \notin \mathbb{R} \quad \rightsquigarrow \quad \neg Crossable(E, ev, F)$$

et de considérer le bégaiement en interdisant les transitions entre deux états différents, qui sont franchies par un événement non présent dans l'automate de la propriété :

$$E \in \mathbb{Q}_P\ \wedge\ F \in \mathbb{Q}_P - \{E\}\ \wedge\ ev \in \mathbb{E}_M - \mathbb{E}_P \quad \rightsquigarrow \quad \neg Crossable(E, ev, F)$$

De cette manière, la propriété *(2)* issue des objectifs de sécurité de DEMONEY et décrite précédemment peut alors être reformulée en SEPL comme suit :

Exemple 7.2 (*Reformulation de la propriété (2)*) :

Définissons l'ensemble des noms d'événements $\mathbb{E}$ par les noms des événements de DEMONEY et l'ensemble des noms d'états $\mathbb{Q}$ par les deux états $\{Un\ solde, Autres\ soldes\}$. On peut alors caractériser la propriété (2) de la manière suivante :

1. Toute instruction autre que *CompleteTransaction* laisse le solde inchangé :

$$\forall_{\mathbb{E}}\ ev \cdot (ev \in \mathbb{E} - \{CompleteTransaction\} \Rightarrow \neg Crossable(Un\ solde, ev, Autres\ soldes))$$

2. *CompleteTransaction* peut changer le solde :

$$Crossable(Un\ solde, CompleteTransaction, Autres\ soldes)$$

7.4.3.2 Validation d'un système de transitions par rapport à une propriété SEPL

Comme nous l'avons dit, une propriété décrite en SEPL peut être établie sur un modèle B par preuve de la formule logique en laquelle elle se réécrit. Dans cette section, nous proposons une méthode qui se base sur une évaluation syntaxique de la valeur des atomes, à partir d'un système de transitions associé au modèle B à vérifier. Dans un premier temps, nous définissons (condition 10) syntaxiquement la valeur des atomes en fonction d'un système de transition minimal. Ensuite, nous faisons le lien entre l'espace d'états de la propriété vérifiée et celui du **STES** (Propriété 3).

Dans un souci de simplification, nous considérons, dans la condition suivante, que s'il n'existe pas de transition par ev entre deux états E et F, alors la transition existe dans la relation de transition avec ses conditions de franchissement D et A à bfalse.

Condition 10 (Évaluation syntaxique des atomes à partir d'un STES minimal)
Soit ev le nom d'un événement et E_1 et E_2 deux noms d'états.

1. $Enabled(E_1, ev) \Leftrightarrow \exists(D, A, E_2) \cdot ((E_1, (D, A, ev), E_2) \in \mathbb{R} \wedge D \not\equiv \mathsf{bfalse})$
2. $AlwaysEnabled(E_1, ev) \Leftrightarrow \exists(D, A, E_2) \cdot ((E_1, (D, A, ev), E_2) \in \mathbb{R} \wedge D \equiv \mathsf{btrue})$
3. $Crossable(E_1, ev, E_2) \Leftrightarrow \exists(D, A) \cdot ((E_1, (D, A, ev), E_2) \in \mathbb{R} \wedge D \not\equiv \mathsf{bfalse} \wedge A \not\equiv \mathsf{bfalse})$
4. $AlwaysCrossable(E_1, ev, E_2) \Leftrightarrow \exists(D, A) \cdot ((E_1, (D, A, ev), E_2) \in \mathbb{R} \wedge D \equiv \mathsf{btrue} \wedge A \equiv \mathsf{btrue})$

Les règles d'évaluation décrites dans la condition 10 ont été établies à partir des obligations de preuve de construction des conditions de franchissement présentées en section 4.2.2.3 (tableaux 4.1 et 4.2) et de la définition des atomes (Définition 25).

Notons que si le **STES** n'est pas minimal, alors la valeur des atomes n'est pas définie pour les transitions ayant une condition en défaut de preuve.

Pour faire le lien entre l'espace d'états de la propriété et celui du modèle, nous utilisons une fonction $\mathcal{S}up$. Afin de permettre la vérification d'une propriété dont les états ne sont pas strictement ceux du **STES**, et permettre ainsi d'utiliser un même **STES** pour vérifier différentes propriétés, nous n'imposons pas que la fonction $\mathcal{S}up$ soit injective. Ainsi, de même que pour la technique de vérification de propriétés décrites par des automates, il suffit que les états du **STES** soient des sous-états des états de la propriété. Si la fonction $\mathcal{S}up$ associe chaque état du **STES** à son super état, alors la vérification peut être effectuée syntaxiquement en utilisant les règles suivantes :

Propriété 3 (Propriétés des prédicats sur la hiérarchie d'états) *Soit $\mathcal{S}up$ une fonction de hiérarchie qui associe chaque état q_M du modèle M à son super-état q_P de la propriété P, ($\mathcal{S}up \in \mathbb{Q}_M \twoheadrightarrow \mathbb{Q}_P$). On a alors les équivalences suivantes :*

$$
\begin{aligned}
Enabled(q_P, ev) &\Leftrightarrow \bigvee_{q_M^i \in \mathcal{S}up^{-1}[\{q_P\}]} Enabled(q_M^i, ev) \\
AlwaysEnabled(q_P, ev) &\Leftrightarrow \bigwedge_{q_M^i \in \mathcal{S}up^{-1}[\{q_P\}]} AlwaysEnabled(q_M^i, ev) \\
Crossable(q_P^1, ev, q_P^2) &\Leftrightarrow \bigvee_{q_M^i \in \mathcal{S}up^{-1}[\{q_P^1\}]} \bigvee_{q_M^j \in \mathcal{S}up^{-1}[\{q_P^2\}]} Crossable(q_M^i, ev, q_M^j) \\
AlwaysCrossable(q_P^1, ev, q_P^2) &\Leftrightarrow \bigwedge_{q_M^i \in \mathcal{S}up^{-1}[\{q_P^1\}]} \bigwedge_{q_M^j \in \mathcal{S}up^{-1}[\{q_P^2\}]} AlwaysCrossable(q_M^i, ev, q_M^j)
\end{aligned}
$$

où q_M un état du modèle M et q_P, q_P^1 et q_P^2 des états de la propriété P.

Si le **STES** est minimal, alors la méthode de vérification syntaxique que nous proposons (Algorithme 13) est décidable.

Algorithme 13 (*Validation syntaxique d'un modèle*) :

1. Construire un **STES** minimal du modèle et définir le lien $\mathcal{S}up$ avec l'espace d'états de la propriété
2. Éliminer les quantificateurs (Définition 27)
3. Ré-exprimer la propriété en termes des états du **STES** (Propriété 3)
4. Évaluer les atomes syntaxiquement sur le **STES** (Condition 10)
5. Évaluer la formule de logique propositionnelle obtenue

7.4.3.3 Exemple de vérification de propriété

Nous proposons d'établir que le modèle de DEMONEY respecte la propriété (2). Pour ce faire, nous voulons vérifier la formule suivante :

/ Toute instruction autre que CompleteTransaction laisse le solde inchangé */*

$$\forall_{\mathbb{E}}\ ev \cdot (ev \in \mathbb{E} - \{CompleteTransaction\} \Rightarrow \neg Crossable(Un\ solde, ev, Autres\ soldes)) \quad (1)$$

/ CompleteTransaction peut changer le solde */*

$$\wedge\ Crossable(Un\ solde, CompleteTransaction, Autres\ soldes) \quad (2)$$

Pour ce faire, nous construisons le **STES** de DEMONEY présenté en figure 7.7 (Section 7.4.2) où les états E_1 et E_2 sont définis par $\mathcal{D}ef(E_1) \mathrel{\hat=} SS = Solde$ et $\mathcal{D}ef(E_2) \mathrel{\hat=} SS \neq Solde$. Nous associons ensuite $\mathcal{S}up(E_1) = Un\ solde$ et $\mathcal{S}up(E_2) = Autres\ soldes$. Pour définir la valeur de la formule précédente, on déplie les quantifications, on ré-exprime la propriété en termes des états du **STES**, on évalue les atomes et on simplifie la formule. Par utilisation de la condition 10, la formule (2) s'évalue à `true` sur la figure 7.7, puisqu'il existe une transition allant de E_1 vers E_2 par *CompleteTransaction*. La formule (1), quant à elle, se réécrit en :

$$
\begin{cases}
\neg Crossable(E_1, GetData, E_2) \\
\ \wedge \neg Crossable(E_1, VerifyPin, E_2) \\
\ \wedge \neg Crossable(E_1, InitializeTransaction, E_2)
\end{cases}
$$

qui est également vrai, par application de la condition 10. Le modèle décrit est donc correct vis-à-vis de la propriété (2).

De la même manière, les autres propriétés de sécurité de DEMONEY (tableau 7.3) sont également exprimables en SEPL. Le lecteur intéressé pourra se référer à [BPS05].

7.5 Synthèse

Dans le domaine des systèmes réactifs, et en particulier des cartes à puce, les programmes et leur politiques de sécurité sont principalement décrits en termes de leurs comportements. Leur représentation à l'aide de systèmes de transitions permet donc de donner une vue pertinente pour vérifier leur validité vis-à-vis de leur cahier des charges.

Dans ce chapitre, nous avons décrit les objectifs du projet GECCOO et les principes généraux du fonctionnement d'une carte à puce. Nous avons ensuite présenté l'étude de cas DEMONEY qui modélise un porte-monnaie électronique. Pour illustrer ce chapitre nous avons introduit une version simplifiée de ce modèle, ne prenant en compte qu'un sous-ensemble des commandes APDU[15]. Ce choix de description a été effectué de telle sorte que les principales propriétés de sécurité décrites par le cahier des charges soient exprimables sur le modèle.

Nous avons ensuite introduit trois méthodes de vérification de propriétés qui nous ont permis, dans le cadre du projet GECCOO, de valider le modèle de DEMONEY par rapport à ses propriétés de sécurité. La première de ces méthodes s'intéresse à des propriétés invariantes et donne un cadre méthodologique pour le renforcement de l'invariant, si celui-ci est trop faible. La seconde méthode s'intéresse au raffinement d'une propriété décrite par un automate. On vérifie alors l'inclusion des comportements du modèle dans ceux autorisés par la propriété, modulo le bégaiement. Enfin, la troisième méthode permet de vérifier des propriétés décrites par des formules logiques en SEPL. Ces trois méthodes sont décidables si le **STES** utilisé est minimal.

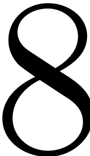
Ces méthodes ont été présentées dans un ordre d'expressivité croissante. En effet, les propriétés invariantes sont exprimables dans la méthode basée sur des automates, et les propriétés descriptibles par des automates sont exprimables en SEPL. Cependant, seules les propriétés exprimées par les deux premières méthodes sont préservées par raffinement. En effet, la troisième méthode permet d'imposer l'existence d'une transition entre deux états, ce qui n'est pas préservé par raffinement. En effet, durant le processus de raffinement, il est possible de réduire les comportements d'un événement, qui peut ne plus pouvoir atteindre un état.

[15] Le lecteur intéressé par le modèle complet peut toutefois le récupérer à l'adresse suivante : http://www-lsr.imag.fr/Les.Personnes/Nicolas.Stouls/?ZoomSur=ProjetGECCOO#ProjetGECCOO

Quatrième partie

Bilan

Conclusion et perspectives



A computer will do what you tell it to do, but that may be much different from what you had in mind.

Joseph Weizenbaum

Sommaire

8.1 Conclusion 169
8.2 Perspectives 172

Pour conclure ce manuscrit, nous faisons un bilan de l'approche que nous avons suivie et des travaux que nous avons menés, avant de terminer en proposant différentes perspectives d'extension et d'utilisation de ce travail.

8.1 Conclusion

Étant donné l'omniprésence croissante des systèmes informatiques et la criticité de leurs tâches, la sécurité des logiciels est un problème particulièrement d'actualité. Comme nous l'avons exposé, la qualité du processus de développement d'un système est sa première garantie de sécurité. Nous avons évoqué l'exemple des Critères Communs [Com05], qui permettent de certifier la sécurité d'un système d'informations en se basant sur la maîtrise de son développement. Dans ce processus, les concepteurs doivent fournir une documentation de conception décrivant les différents niveaux de raffinement, ainsi qu'un argumentaire permettant de convaincre les évaluateurs de la correcte implantation des fonctions de sécurité. Pour l'obtention des notes les plus basses, ces documents peuvent être décrits en langue naturelle, tandis que pour obtenir les notes les plus élevées ces documents doivent être complétés notamment avec la preuve formelle de certains points, tels que la correction et la complétude de l'implantation par rapport à la cible de sécurité.

Conformément à cette approche de certification, l'objectif de cette thèse est d'aider les concepteurs de modèles B en fournissant, tout au long du développement par raffinement, un cadre méthodologique d'aide à la compréhension et à la validation. Notre approche se base principalement sur la construction d'un second point de vue du modèle développé. Un modèle B étant décrit en termes du traitement de ses données, alors une représentation de l'ensemble de ses comportements est une vue complémentaire apportant une aide précieuse à sa compréhension et à sa

documentation. Le formalisme utilisé pour décrire cette seconde vue est une dérivation des StateCharts, que nous avons appelés *Systèmes de Transitions Étiquetées Hiérarchique* (**STEH**). Dans le chapitre 4, nous avons décrit les systèmes de transitions étiquetées symboliques (**STES**), qui représentent les comportements d'une spécification abstraite B événementiel. Dans le chapitre 5, nous avons ensuite ajouté la notion de hiérarchie pour définir les **STEH** en fonction des **STES**, et ainsi prendre en compte les modèles construits par hiérarchie.

Afin d'obtenir une représentation aussi précise que possible, nous avons défini les **STES** avec deux conditions de franchissement sur chaque transition : une condition de déclenchabilité et une condition d'atteignabilité. La première est la condition sous laquelle un événement peut se déclencher depuis l'état de départ et la seconde définit la condition sous laquelle un événement peut atteindre l'état d'arrivée depuis son état de départ. La garde des transitions telle qu'elle est classiquement définie, dans les StateCharts par exemple est la conjonction de ces deux conditions. Leur utilisation permet au concepteur d'obtenir une information plus précise, puisqu'elles permettent de distinguer, parmi les valuations autorisant le déclenchement de l'événement, celles permettant d'atteindre l'état d'arrivée d'une transition et les autres. Nous avons ensuite introduit une méthode de construction d'un système de transitions représentant les comportements d'une spécification abstraite B. Cette méthode se base sur la génération et la résolution d'obligations de preuve. Étant donné un espace d'états fourni par l'utilisateur, nous calculons si un événement permet de passer d'un état à l'autre. Nous avons alors caractérisé trois valeurs particulières possibles pour chaque condition de franchissement : btrue, bfalse et *conditionnée*. Chacune des ces valeurs est associée à une obligation de preuve. Si aucune des trois obligations de preuve associée à une condition ne peut être résolue, alors la condition est en *défaut de preuve*.

L'espace d'états utilisé détermine la vue que l'on a des comportements du système. Ce choix est donc effectué à partir des aspects du modèle que l'on veut mettre en évidence. C'est pourquoi, nous proposons de laisser l'utilisateur fournir l'espace d'états. Toutefois, nous avons décrit, section 4.4.2, un certain nombre de stratégies de construction d'un espace d'états. Certaines d'entre elles ont également été intégrées dans l'outil *GénéEtat*, réalisé dans l'équipe, et permettant de générer automatiquement un espace d'états.

Dans le chapitre 5, nous avons pris en compte le processus de raffinement. Pour ce faire, nous avons introduit la notion de hiérarchie dans les **STES** pour définir les **STEH**. Cependant, contrairement à l'approche classique consistant à associer des **STES** par la fonction de hiérarchie, nous n'avons hiérarchisé que les états. La relation de transition est donc définie sur l'ensemble des états, quel que soit leur niveau de hiérarchie. Cette méthode permet de décrire finement les transitions externes[1]. La méthode de construction que nous proposons se base sur la modification du **STEH** abstrait, permettant ainsi de préserver la vue que l'on peut avoir à un niveau donné. Elle consiste à projeter la définition des états abstraits sur les variables du raffinement, puis à les subdiviser pour ajouter un niveau de hiérarchie. La relation de transition est ensuite calculée entre les états-feuille, avant de factoriser les transitions ou de les mettre en relation avec des sous-états initiaux ou finaux. Lors du calcul des transitions, nous exploitons certaines propriétés du raffinement pour diminuer le nombre d'obligations de preuve à vérifier. Par exemple, une transition n'est franchissable dans le raffinement que si elle l'est aussi dans l'abstraction, entre les mêmes états.

[1] Transitions entre des états n'étant pas dans le même super état

Dans le chapitre 6, nous présentons l'outil *GénéSyst*, qui a été développé au cours de cette thèse et qui implante les différents algorithmes présentés dans les chapitres 4, 5 et 6. En particulier, un gros travail d'optimisation des obligations de preuve générée et d'aide à la preuve a été réalisé pour limiter au maximum les cas de défaut de preuve. *GénéSyst* utilise différents outils existants pour charger les composants B, les manipuler et résoudre les obligations de preuve produites. Bien qu'il contienne encore certaines limitations de représentation graphique[2], il permet tout de même de prendre en compte des composants de raffinement et peut produire des systèmes de transitions hiérarchiques.

Les méthodes de construction décrites ont également été adaptées pour représenter les comportements de composants B classique. Pour ce faire, nous avons proposé des règles de réécriture pour traduire un composant B classique en système B événementiel. Contrairement aux événements, les opérations peuvent avoir des paramètres entrants ou sortants. Nous avons alors introduit deux méthodes pour prendre en compte les paramètres. La première consiste à masquer les paramètres, tandis que la seconde les externalise, permettant ainsi de caractériser les états aussi en termes des entrées/sorties. Cependant, la construction que nous proposons pour les composants B classique est une sur-approximation des séquences d'appel d'opérations qu'ils autorisent. En effet, il existe une différence sémantique entre un enchaînement d'événements et un enchaînement d'opérations. Dans le premier cas, un événement ev_2 peut être déclenché après un événement ev_1 s'il existe une valeur atteignable par ev_1 et depuis laquelle ev_2 est déclenchable. À l'inverse, une opération op_2 ne peut être appelée après une opération op_1 que si toutes les valuations atteignables par op_1 vérifient la pré-condition de op_2.

Enfin, cette thèse a été partiellement effectuée dans le contexte du projet de l'ACI sécurité GECCOO. Dans le chapitre 7, nous décrivons l'approche de vérification de propriétés de sûreté que nous avons suivi au sein de ce projet. Nous avons proposé trois techniques qui se basent sur l'utilisation de l'outil *GénéSyst* pour vérifier syntaxiquement le respect d'une propriété par un modèle, pour des formes différentes de propriétés. La première technique est une aide à la vérification de propriétés invariantes. Elle consiste à mettre en évidence la non-atteignabilité des états violant la propriété. La seconde technique s'intéresse à des propriétés décrites par des automates, dont la sémantique est définie en terme de traces d'états/événements. Le respect d'une telle propriété par une spécification est alors définie par une notion de raffinement structurel avec bégaiement. Enfin, la troisième technique proposée se base sur un langage de logique permettant de décrire les comportements attendus d'un modèle. Ce langage, nommé SEPL, est basé sur quatre primitives : Enabled, Crossable, AlwaysEnabled et AlwaysCrossable. La méthode proposée consiste à évaluer syntaxiquement ces primitives sur le système de transitions étiquetées extrait du modèle B, nous ramenant ainsi à des formules décidables. Ces différentes techniques ont été utilisées dans le cadre du projet GECCOO pour vérifier la conformité du modèle B de l'applette DEMONEY par rapport aux propriétés de sécurité définies dans son cahier des charges [MM02].

Pour clore ce bilan, précisons que les travaux menés durant cette thèse ont donné lieu à plusieurs publications :

– La génération de systèmes de transitions étiquetées représentant les comportements d'un

[2] Dans son état actuel, il permet de factoriser des transitions et de choisir les sous-états initiaux et finaux, mais il ne fourni ces deux sorties qu'en mode texte.

modèle B événementiel et les premières idées de prise en compte du raffinement ont été publiées dans [PS04] ;

– L'outil *GénéSyst*, ainsi que notre approche d'aide à la spécification et au développement formel de systèmes, ont été présentés dans [MPS04, Sto06a] ;

– L'approche *GénéSyst* et son utilisation pour la vérification de propriétés de sécurités décrites par des formules logiques ont été développés dans [BPS05] ;

– L'étude de cas du moniteur réseau réalisée en partenariat avec des participants au projet POTESTAT a été présentée dans [SD05, SD06, SP07] ;

– Enfin, deux rapports techniques ont été rédigés au cours de cette thèse. Le premier est une introduction au domaine des cartes à puce [Sto06b] et le second et une introduction aux Critères Communs [Mer00].

8.2 Perspectives

Les résultats de cette thèse ont été outillés et sont utilisables, puisqu'ils ont été exploités dans différentes études de cas pour aider notamment à leur vérification et à leur documentation (Section 6.3.2). Toutefois, certains points de ces résultats pourraient être améliorés ou adaptés pour être appliqués dans d'autres domaines. Nous développons ici ces deux types de perspectives.

Améliorations

– L'utilisation de deux conditions de franchissement permet d'améliorer la compréhension du lien entre le modèle et ses comportements, car cela permet de distinguer les valuations ne permettant pas de déclencher l'événement et celles ne permettant pas d'atteindre un état donné. Cependant, ces deux conditions portent sur l'état des variables avant le déclenchement de l'événement. Il serait intéressant de caractériser également les valuations de l'état d'arrivée qui sont atteignables. Cette information serait particulièrement intéressante pour aider à traiter des problèmes d'atteignabilité d'une valuation à travers une séquence d'occurrences d'événements. Pour caractériser ce prédicat, il est alors possible soit d'utiliser le calcul de plus forte post-condition, décrit initialement dans [Dij76, Hes92], puis adapté à B par S. Dunne [Dun03], soit le $\mathcal{WP}$. Dans ce dernier cas, il serait possible d'exprimer la condition T caractérisant que « *toutes les valuations de l'état F sont atteignables par l'événement ev depuis l'état E* » par l'obligation de preuve suivante :
$$T \mathrel{\hat{=}} \forall x' \cdot \exists x \cdot (E \ \wedge \ D \ \wedge \ A \ \wedge \ \langle \mathsf{Action}(ev) \rangle [x := x'] F)$$

– L'une des heuristiques introduites tout au long de ce manuscrit est de réduire le nombre de défauts de preuve sans restreindre le langage B événementiel considéré. Pour améliorer encore les résultats obtenus, nous avons alors expérimenté une utilisation de tactiques utilisateur dans la version actuelle de *GénéSyst*. Pour chaque condition de franchissement, si le prouveur automatique échoue à prouver une formule, alors nous proposons d'utiliser successivement les tactiques suivantes dans le prouveur interactif de l'*AtelierB* ou de *B4free*, jusqu'à ce que

toutes les cinq soient tentées ou que le but soit prouvé :

1. `dd && pp(rp.0)`
2. `re && pp(rp.0)`
3. `re && dd && pp(rp.1)`
4. `re && dd && pp(rp.2)`
5. `re && pp(rp.1)`

La commande `dd` permet de ne considérer que le but de la preuve et de réunir toutes les hypothèses dans un ensemble d'*hypothèses globales*, tandis que la commande `pp` est un appel au prouveur de prédicats. Le paramètre `rp.x` permet alors de préciser le sous-ensemble des hypothèses globales qu'il peut utiliser[3]. Enfin, la commande `re` permet de recommencer la preuve à zéro. L'utilisation de ces tactiques a permis de résoudre la quasi-totalité des cas de défaut de preuve de la batterie de tests (section 6.3.1). Dans cette version prototype de *GénéSyst*, les temps de calcul sont démesurés (la construction des comportements du canal de communication passe de 4 secondes à 77 secondes), mais ils pourraient être diminués en n'utilisant les tactiques que dans les cas de défaut de preuve avéré. Nous n'avons pas encore pu tester cette approche sur les plus gros modèles, mais le taux de réduction des cas de défaut de preuve semble très prometteur.

– Une autre approche pour minimiser le nombre de défauts de preuve consisterait à utiliser d'autres prouveurs, ayant chacun leur propres tactiques. Pour ce faire, nous pourrions nous intéresser aux démarches outillées telles que *Why* [Fil03], qui est un générateur d'obligations de preuve pour de multiples prouveurs (*Simplify* [DNS05], *Ergo* [CCK07], *haRVey* [RD03], etc). Cet outil prend en entrée une spécification *Why* décrite en termes de pré et post-conditions et génère des obligations de preuve dans le langage de chacun des prouveurs pris en compte. Il suffit alors d'axiomatiser les symboles non interprétés de la logique B et d'exprimer les assertions à l'aide du langage des prédicats de *Why*. En ce sens, ce travail pourrait s'inspirer de ce qui a été fait avec succès dans l'outil *Barvey* [CDD+04, CDGR04] et plus récemment dans [CD07].

Utilisations

– En section 2.3.1, nous avons vu que différentes approches ont été proposées pour dériver un modèle B à partir d'une description comportementale [MS99, GFL07, But00]. Il serait intéressant de coupler ces approches avec celle de *GénéSyst* pour proposer une méthode de développement multi-vues. La principale difficulté de ce type d'approche est de maintenir une certaine cohérence entre les différentes vues. Pour ce faire, l'idée serait de dériver le squelette B à partir d'une spécification comportementale, puis de vérifier le modèle développé en comparant ses comportements effectifs avec ceux attendus. Cette validation peut alors être effectuée avec *GénéSyst* après le développement de chaque raffinement. Typiquement,

[3] `rp.0` : aucune hypothèse ; `rp.1` hypothèses qui ont une variable libre en commun avec le but ; `rp.2` : hypothèses qui ont une variable libre en commun soit avec le but, soit avec une hypothèse ayant une variable libre en commun avec le but.

l'intégration de *GénéSyst* dans la plateforme Rodin [JCI+05, BLS05] sous la forme d'un plugin permettrait d'atteindre facilement ce résultat.
D'autres approches de développement multi-vues ont déjà été proposées. Nous pouvons citer par exemple les approches CSP——B [ST05] et CSP+B [BL05] qui consistent à séparer les aspects contrôle, décrit en CSP, et données, décrit en B. Il est alors possible de développer séparément les deux aspects, modulo des preuves de cohérence. L'approche UML-B [OJS05] permet également de considérer en même temps une vue UML et une vue B du modèle. Ces deux vues sont ici considérées comme toujours cohérentes. Développer un tel modèle consiste alors à appliquer des règles de réécriture modifiant conjointement les deux vues, en préservant leur cohérence. Contrairement à ces approches, la méthode que nous proposons aurait l'avantage de conserver la séparation des trois phases du cycle de développement en V : conception, développement et validation. Il serait alors possible de concevoir le système en UML avant de développer le modèle formel et de valider ensuite la conformité de chaque raffinement avec le cahier des charges.

- Différents aspects de la méthode de vérification de propriétés de sécurité basée sur le langage SEPL (présenté en section 7.4.3) pourraient être développés. Il serait notamment intéressant d'exploiter la hiérarchie des systèmes de transitions pour vérifier des propriétés par partie et ainsi simplifier leur vérification. Par extension, cela reviendrait à vérifier des propriétés sur le modèle abstrait et de les préserver par raffinement. Par exemple dans [MMJ00, Lan05], les auteurs caractérisent une classe de propriétés logiques pouvant être vérifiées par partie sur un modèle modulaire. Intuitivement, cette approche devrait pouvoir être adaptée aux systèmes de transitions hiérarchiques.

- Enfin, nous avons commencé, à travers le stage d'ingénieur de deuxième année d'Évelyne Altariba, à étudier l'impact de la modularité sur la construction des comportements d'un modèle B classique. Cependant, les premiers résultats n'ont pas encore pu être exploités. Intuitivement, dans un modèle défini par un ensemble de composants communicants, c'est le composant appelant qui va contraindre les comportements du composant appelé. De la même manière les résultats des appels d'opérations peuvent influencer les comportements de l'appelant. Il serait donc intéressant de permettre un calcul des comportements d'un composant B en prenant en compte son environnement d'exécution. De plus, nous avons utilisé la hiérarchie pour décrire les comportements d'un modèle raffiné. Ce type de représentation pourrait également être utilisé pour représenter les comportements d'un modèle composé de plusieurs modules. En effet, si une transition représente l'exécution d'une opération, alors une vue plus fine du système permettrait de visualiser les opérations appelées lors de cette exécution.

Cinquième partie

Annexes

Complément sur les substitutions généralisées

A.1 Calcul de la terminaison

Le tableau A.1 résume le résultat du calcul de la terminaison des principales substitutions primitives.

Substitution	Condition de terminaison
$\mathsf{trm}(x := E)$	btrue
$\mathsf{trm}(x, y := E, F)$	btrue
$\mathsf{trm}(\mathbf{skip})$	btrue
$\mathsf{trm}(P \mid S)$	$P \wedge \mathsf{trm}(S)$
$\mathsf{trm}(P \Longrightarrow S)$	$P \Rightarrow \mathsf{trm}(S)$
$\mathsf{trm}(S_1 \,[\!]\, S_2)$	$\mathsf{trm}(S_1) \wedge \mathsf{trm}(S_2)$
$\mathsf{trm}(@z \cdot S)$	$\forall z \cdot \mathsf{trm}(S)$
$\mathsf{trm}(S \,;\, S_2)$	$[S]\mathsf{trm}(S_2)$

TAB. A.1 – Terminaison des principales substitutions

A.2 Calcul du prédicat avant-après

Le tableau A.2 résume le résultat du calcul du prédicat avant-après sur les principales substitutions primitives.

Substitution	Prédicat avant-après	Condition
$\mathsf{prd}_x(x := E)$	$x' = E$	
$\mathsf{prd}_{x,y}(x := E)$	$x' = E \wedge y' = y$	
$\mathsf{prd}_{x,y}(x, y := E, F)$	$x' = E \wedge y' = F$	
$\mathsf{prd}_x(\mathbf{skip})$	$x' = x$	
$\mathsf{prd}_x(P \mid S)$	$P \Rightarrow \mathsf{prd}_x(S)$	
$\mathsf{prd}_x(P \Longrightarrow S)$	$P \wedge \mathsf{prd}_x(S)$	
$\mathsf{prd}_x(S \,[\!]\, T)$	$\mathsf{prd}_x(S) \vee \mathsf{prd}_x(T)$	
$\mathsf{prd}_x(@z \cdot S)$	$\exists z \cdot \mathsf{prd}_x(S)$	si $z \backslash x'$ et z non modifié par S
$\mathsf{prd}_x(@y \cdot S)$	$\exists y, y' \cdot \mathsf{prd}_{x,y}(S)$	si $y \backslash x'$ et y modifié par S
$\mathsf{prd}_x(S \,;\, T)$	$\langle S \rangle \mathsf{prd}_x(T)$	

TAB. A.2 – Prédicat avant-après des principales substitutions

A.3 Calcul de $\mathcal{WP}$ conjugué

Le conjugué du calcul de $\mathcal{WP}$ (noté $\mathcal{WP}^{cg}$) a été introduite par C. Morgan [WADJ90], puis adaptée en B par M. Butler [But00]. Cette notation étant fréquemment utilisée, nous montrons, tableau A.3, les résultats de ce calcul sur les principales substitutions généralisées primitives.

Cas de substitution	Réduction	Condition
$\langle x := E \rangle R$	$[x := E]R$	
$\langle x, y := E, F \rangle R$	$\langle z := F \rangle \langle x := E \rangle \langle y := z \rangle R$	$z \backslash\ E, F, R$
$\langle \mathbf{skip} \rangle R$	R	
$\langle P \mid S \rangle R$	$P \Rightarrow\ \langle S \rangle R$	
$\langle P \Longrightarrow S \rangle R$	$P\ \wedge\ \langle S \rangle R$	
$\langle S \parallel T \rangle R$	$\langle S \rangle R\ \vee\ \langle T \rangle R$	
$\langle @\ z \cdot S \rangle R$	$\exists z \cdot \langle S \rangle R$	$z \backslash\ R$
$\langle S\ ;\ T \rangle R$	$\langle S \rangle \langle T \rangle R$	

TAB. A.3 – Définition du $\mathcal{WP}^{cg}$ sur les principales substitutions primitives.

Démonstrations

B

B.1 Forme normalisée d'un événement

En section 4.1.3, on définit la forme normalisée des événements comme suit :

Définition 28 (Forme normalisée d'un événement) *Tout événement $ev \hat{=} S$ peut se mettre sous une forme normalisée $\mathcal{F}(ev)$:*

$$\mathcal{F}(ev) \hat{=} \mathsf{Garde}(ev) \Longrightarrow \mathsf{Action}(ev)$$

où $\mathsf{Garde}(ev) = \mathsf{fis}(S)$ et $\mathsf{Action}(ev) = S$.

Pour vérifier que tout événement $ev \hat{=} S$ est équivalent à sa forme normalisée $\mathcal{F}(ev)$ nous proposons de montrer que, pour tout prédicat R, on a $[S]R \Leftrightarrow [\mathcal{F}(ev)]R$.

Démonstration 2 (Correction de la forme normalisée d'un événement (définition 12))

Montrons que :

$$[S]R \;\Leftrightarrow\; [\mathcal{F}(ev)]R$$

Par application de la définition de $\mathcal{F}$, de Garde *et de* Action *(Définition 12), cela revient à montrer :*

$$\equiv \quad [S]R \;\Leftrightarrow\; [\mathsf{fis}(S) \Longrightarrow S]R$$

Comme tout événement B termine ($\mathsf{trm}(S) \Leftrightarrow \mathsf{btrue}$)*, alors le passage à la forme normalisée des substitutions généralisées (section 3.3.3) nous ramène à montrer que :*

$$\equiv \quad \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']R) \;\Leftrightarrow\; (\mathsf{fis}(S) \Rightarrow \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']R))$$

$$\equiv \quad \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']R) \;\Leftrightarrow\; (\exists x' \cdot (\mathsf{prd}_x(S)) \Rightarrow \forall x' \cdot (\mathsf{prd}_x(S) \Rightarrow [x := x']R))$$

Si $\exists x' \cdot (\mathsf{prd}_x(S))$ est vrai alors les parties gauche et droite de l'équivalence sont identiques. Sinon les deux formules sont vraies, car leurs antécédents sont faux.

□

B.2 Trace d'un système B événementiel

En section 4.3.1, le lemme suivant est défini pour caractériser si une séquence d'occurrences d'événements est une trace d'un système B événementiel. La preuve de la correction de ce lemme par rapport à la définition 16 des traces d'un système B événementiel est donnée ci-après.

Lemme 2 (Caractérisation d'une trace) *Soit S un système B événementiel, Init l'initialisation de S et oc_1 à oc_n des occurrences d'événements de S, alors :*

$$Init \ ; \ oc_1 \ ; \ \ldots \ ; \ oc_n \in \mathsf{Traces}(S) \Leftrightarrow$$
$$\exists x_1, \ldots, x_{n+1} \cdot ([x' := x_1]\mathsf{prd}_x(Init) \wedge \bigwedge_{i=1}^{n}([x := x_i]\mathsf{Garde}(oc_i) \wedge [x, x' := x_i, x_{i+1}]\mathsf{prd}_x(\mathsf{Action}(oc_i))))$$

avec x_i une valuation des variables du système S.

Démonstration 3 (Démonstration du lemme 1) *En définition 16, nous avons vu qu'une suite d'occurrences d'événements Init ; oc_1 ; … ; oc_n est une trace d'un système B si et seulement si :*

$$\mathsf{fis}(Init \ ; \ oc_1 \ ; \ \ldots \ ; \ oc_n)$$

et en définition 12 que tout événement ev peut être mis sous une forme normalisée :

$$\mathsf{Garde}(ev) \Longrightarrow \mathsf{Action}(ev)$$

or on peut retrouver à partir de la définition de la faisabilité (Définition 11) que :

$$\mathsf{fis}(\mathsf{Garde}(ev_1) \Longrightarrow \mathsf{Action}(ev_1); \mathsf{Garde}(ev_2) \Longrightarrow \mathsf{Action}(ev_2)) \Leftrightarrow$$
$$\exists x_1, x_2 \cdot (\mathsf{Garde}(ev_1) \wedge [x' := x_1]\mathsf{prd}(\mathsf{Action}(ev_1)) \wedge [x := x_1]\mathsf{Garde}(ev_2) \wedge [x, x' := x_1, x_2]\mathsf{prd}(\mathsf{Action}(ev_2)))$$

Il s'ensuit alors que :

$$\mathsf{fis}(Init; oc_1; \ldots; oc_n) \Leftrightarrow$$
$$\exists x_1, \ldots, x_{n+1} \cdot ([x := x_1]\mathsf{prd}(Init) \wedge \bigwedge_{i=1}^{n}([x := x_i]\mathsf{Garde}(oc_i) \wedge [x, x' := x_i, x_{i+1}]\mathsf{prd}(\mathsf{Action}(oc_i))))$$

□

B.3 Prise en compte du raffinement : implication des conditions des transitions

Dans cette section, nous proposons de démontrer la correction de la propriété 1 (Section 5.2.2.1). Cette propriété, rappelée ci-dessous, permet de simplifier la construction d'une relation de transition raffinée en garantissant qu'une transition $(E, (D_R, A_R, ev), F)$ ne peut être valide dans le raffinement que s'il existe une transition $(\mathcal{S}up(E), (D_S, A_S, ev), \mathcal{S}up(F))$ valide dans la spécification.

Propriété 4 (Rappel de la propriété 1) *Soit la transition $(E, (D_R, A_R, ev), F)$ du système raffiné T_R et la transition $(\mathcal{S}up(E), (D_S, A_S, ev), \mathcal{S}up(F))$ du système abstrait T_S. S'il a été prouvé que R raffine S, alors on a nécessairement :*

$$L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ D_R \Rightarrow D_S \qquad (1.1)$$
$$L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ D_R \ \wedge \ A_R \Rightarrow A_S \qquad (1.2)$$

Démonstration 4 (Propriété 1.1)

Par définition du raffinement on a :

$$L \ \wedge \ \mathsf{Garde}_R(ev) \Rightarrow \mathsf{Garde}_S(ev)$$

Par renforcement des hypothèses, on se ramène à :

$$\Rightarrow \quad L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ \mathsf{Garde}_R(ev) \Rightarrow \mathsf{Garde}_S(ev)$$

Enfin, par réécriture selon la définition de la déclenchabilité (Def. 13, section 4.2.2.1) :

$$\equiv \quad L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ D_R \Rightarrow D_S$$

□

Démonstration 5 (Propriété 1.2)

À partir de la définition du raffinement d'un événement décrite en section 3.6.4, nous avons :

$$L \ \wedge \ \mathsf{prd}_y(\mathsf{Action}_R(ev)) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(\mathsf{Action}_S(ev)) \ \wedge \ [y, x := y', x']L)$$

Comme l'espace d'états-feuille est complet vis-à-vis de l'invariant J du raffinement[1] *(Conséquence de la condition 7, page 108), il existe donc nécessairement un état-feuille F de $\mathbb{Q}$ tel que $[y := y']\mathcal{D}ef_R(F)$. Comme $[y, x := y', x']L$ est vérifié, alors le super-état de F vérifie $[x := x']\mathcal{D}ef_S(\mathcal{S}up(F))$ dans T_S. Nous avons :*

$$\equiv \quad L \ \wedge \ \mathsf{prd}_y(\mathsf{Action}_R(ev)) \Rightarrow \exists x' \cdot \left(\begin{array}{c} \mathsf{prd}_x(\mathsf{Action}_S(ev)) \ \wedge \ [y, x := y', x']L \ \wedge \\ [x := x']\mathcal{D}ef_S(\mathcal{S}up(F)) \ \wedge \ [y := y']\mathcal{D}ef_R(F) \end{array} \right)$$

$$\Rightarrow \quad L \ \wedge \ \mathsf{prd}_y(\mathsf{Action}_R(ev)) \Rightarrow \left(\begin{array}{c} \exists x' \cdot (\mathsf{prd}_x(\mathsf{Action}_S(ev)) \ \wedge \ [x := x']\mathcal{D}ef_S(\mathcal{S}up(F))) \\ \wedge \ [y := y']\mathcal{D}ef_R(F) \end{array} \right)$$

$$\equiv \quad \left(\begin{array}{l} (L \ \wedge \ \mathsf{prd}_y(\mathsf{Action}_R(ev)) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(\mathsf{Action}_S(ev)) \ \wedge \ [x := x']\mathcal{D}ef_S(\mathcal{S}up(F)))) \ \wedge \\ (L \ \wedge \ \mathsf{prd}_y(\mathsf{Action}_R(ev)) \Rightarrow [y := y']\mathcal{D}ef_R(F)) \end{array} \right)$$

Ce qui implique la correction de la formule suivante :

$$\Rightarrow \quad \big(L \wedge \mathsf{prd}_y(\mathsf{Action}_R(ev)) \wedge [y := y']\mathcal{D}ef_R(F)\big) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(\mathsf{Action}_S(ev)) \wedge [x := x']\mathcal{D}ef_S(\mathcal{S}up(F)))$$

Or il existe nécessairement un état E de $\mathbb{Q}$ tel que y vérifie $\mathcal{D}ef_R(E)$ (condition 7). Comme L est en hypothèse, alors x vérifie $\mathcal{D}ef_S(\mathcal{S}up(E))$.

$$\equiv \quad L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ \mathsf{Garde}_R(ev) \ \wedge \ \langle \mathsf{Action}_R(ev) \rangle \mathcal{D}ef_R(F) \Rightarrow \langle \mathsf{Action}_S(ev) \rangle \mathcal{D}ef_S(\mathcal{S}up(F))$$

$$\equiv \quad L \ \wedge \ \mathcal{D}ef_S(\mathcal{S}up(E)) \ \wedge \ D_R \ \wedge \ A_R \Rightarrow A_S$$

□

[1] En B, l'invariant de liaison L est la conjonction de l'invariant I de l'abstraction et l'invariant J du raffinement.

B.4 *GénéSyst* : Simplification du calcul des conditions d'atteignabilité

Le corollaire 1 (section 6.1.2.2), rappelé ci-dessous, permet de diminuer le nombre d'obligations de preuve à vérifier lors de la construction de la relation de transition d'un système de transitions.

Corollaire 2 (Rappel du corollaire 1) *Si un événement ev est déclenchable dans un état E et qu'il ne peut mener dans aucun autre état que F, alors la condition d'atteignabilité A de la transition $(E, (D, A, ev), F)$ est réductible à* btrue.

Ce corollaire se déduit trivialement de la propriété 5 suivante :

Propriété 5 (Union des conditions d'atteignabilité pour un état et un événement) *Si T est l'ensemble des transitions partant de l'état E et étiquetées par l'événement ev, alors la disjonction des conditions d'atteignabilité de T est réductible à* btrue.

Intuitivement, cette propriété correspond au fait que, si un événement est faisable dans un état, alors il mène forcément quelque part. La démonstration de cette propriété est donné ci-dessous :

Démonstration 6 (Preuve de la propriété 5)

Comme le respect de l'invariant par les événements des composants B *utilisés a été prouvé, alors tout événement ev du système est tel que :*

$$I \Rightarrow [ev]I$$

$$\equiv \; I \;\wedge\; \mathsf{Garde}(ev) \Rightarrow [\mathsf{Action}(ev)]I$$

En mettant la substitution $\mathsf{Action}(ev)$ *sous forme normalisée (Section 3.3.3), on a :*

$$\equiv \; I \;\wedge\; \mathsf{Garde}(ev) \Rightarrow \forall x' \cdot (\mathsf{prd}_x(\mathsf{Action}(ev)) \Rightarrow [x := x']I)$$

Or $\mathsf{Garde}(ev) = \mathsf{fis}(ev)$ *(Définition 12) et* $\mathsf{fis}(S) \Leftrightarrow \exists x' \cdot (\mathsf{prd}_x(S))$. *On en déduit donc :*

$$I \;\wedge\; \mathsf{Garde}(ev) \Rightarrow \exists x' \cdot (\mathsf{prd}_x(\mathsf{Action}(ev)) \;\wedge\; [x := x']I)$$

$$\equiv \; I \;\wedge\; \mathsf{Garde}(ev) \Rightarrow \langle \mathsf{Action}(ev) \rangle I$$

Comme l'invariant I est l'union des états du système (conditions 1 et 2), alors on peut le remplacer par la disjonction $\bigvee_{j=1}^{n} \mathcal{D}ef(E_j)$ des n états du système n'étant pas q_{Init}. *On a alors :*

$$\equiv \; \left(\bigvee_{j=1}^{n} \mathcal{D}ef(E_j)\right) \;\wedge\; \mathsf{Garde}(ev) \Rightarrow \langle \mathsf{Action}(ev) \rangle \left(\bigvee_{i=1}^{n} \mathcal{D}ef(E_i)\right)$$

$$\equiv \; \bigwedge_{j=1}^{n} \left(\mathcal{D}ef(E_j) \;\wedge\; \mathsf{Garde}(ev) \Rightarrow \bigvee_{i=1}^{n} \langle \mathsf{Action}(ev) \rangle \mathcal{D}ef(E_i) \right)$$

Notons $A_{(E_j \xrightarrow{ev} E_i)}$ la condition d'atteignabilité associée à la transition allant de E_j à E_i par ev. Par remplacements (selon la définition 13, section 4.2.2.1), nous avons alors :

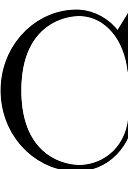
$$\equiv \ \bigwedge_{j=1}^{n} \left(\mathcal{D}ef(E_j) \ \wedge \ D \Rightarrow \bigvee_{i=1}^{n} A_{(E_j \xrightarrow{ev} E_i)} \right)$$

$$\equiv \ \bigwedge_{j=1}^{n} \left(\mathcal{D}ef(E_j) \ \wedge \ D \Rightarrow \left(\left(\bigvee_{i=1}^{n} A_{(E_j \xrightarrow{ev} E_i)} \right) \Leftrightarrow \mathsf{btrue} \right) \right)$$

Ainsi, l'union des conditions d'atteignabilité des transitions déclenchables par un événement ev depuis tout état E_j du système est réductible à btrue.

□

Glossaire



APDU :

Application Protocol Data Unit. Défini par la norme ISO 7816 [Hus01], c'est le format des messages échangés par une carte à puce et le terminal.

API :

Application and Programming Interface. Bibliothèque de fonctions de bas niveau, permettant au développeur de s'abstraire des contraintes matériel.

Applette :

Nom donné aux petites applications, aussi appelées appliquettes. Par transitivité, c'est également le nom donné aux programmes développés en JavaCard (*Applet* en Anglais).

Atteignabilité :

Condition caractérisant, parmi les valuations de l'état de départ qui permettent de déclencher l'événement *ev*, celles permettant à *ev* d'atteindre l'état d'arrivée.

bfalse / btrue :

Les notations btrue et bfalse symbolisent les prédicats respectivement toujours vrai et toujours faux, par opposition aux valeurs booléennes true et false.

Bytecode :

Langage de bas niveau généré par un compilateur Java et pouvant être interprété par une machine virtuelle appelée Java Runtime Environment.

Cohérence :

De manière générale, un modèle formel est dit *cohérent* s'il ne contient pas de contradiction (s'il en existe un modèle). Il est cependant intéressant de noter qu'en B on distingue l'existence d'un modèle de l'invariant du système et le respect de l'invariant par les actions. Le premier cas est la *consistance* du modèle, tandis que le second est sa *cohérence* ou sa *correction*.

Comportements :

Dans cette thèse, nous avons choisis d'appelé les **comportements** d'un modèle, l'ensemble de ses traces d'exécution ; c'est-à-dire l'ensemble des séquences d'événements qu'il autorise.

$$\begin{cases} Seq_1 = \xrightarrow{oc1} \xrightarrow{oc2} \xrightarrow{oc3} \xrightarrow{oc4} \xrightarrow{oc5} \cdots\cdots \xrightarrow{ocn} \\ Seq_2 = \xrightarrow{oc'1} \xrightarrow{oc'2} \xrightarrow{oc'3} \xrightarrow{oc'4} \xrightarrow{oc'5} \cdots\cdots \xrightarrow{oc'm} \\ \ldots \end{cases}$$

Déclenchabilité :

Condition caractérisant les valuations de l'état de départ permettant de déclencher l'événement *ev*.

Défaut de preuve :

On appelle défaut de preuve le cas où un démonstrateur termine en n'ayant pas établi la véracité d'une formule vraie. On parle aussi de faux négatif.

État composé :

Un état composé (ou état hiérarchique ou cluster) est un état contenant un système de transitions.

État-feuille :

État appartenant à un système de transitions hiérarchique et n'étant pas composé.

État racine :

État appartenant à un système de transitions hiérarchique et n'ayant pas de super-état.

Faisabilité :

Ensemble des valuations des variables pour lesquelles il existe une valuation après *ev* (pour lesquelles *ev* peut calculer un état d'après). Ce prédicat est défini en section 4.1.3.

Forêt :

Une Forêt est un ensemble d'arbres. Pour qu'une relation décrive une forêt, il suffit que ce soit une fonction associant son père à chaque fils et qui ne contienne pas de cycle.

Génie logiciel :

Il est défini dans le journal officiel comme « *l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi* » [mdd84]

GlobalPlatform :

Nouveau nom de l'API Open Platform [BWT02]. Souvent livrée avec JavaCard, celle-ci fournit une interface complète de gestion du chiffrage, du déchiffrage et de la signature des messages ainsi que de gestion des clefs.

Les spécifications de GlobalPlatform sont issus des besoins de l'industrie. En particulier, les industriels suivants sont les membres les plus influents de ce standard : ActivIdentity, Datacard, France Telecom, Fujitsu, Gemalto, Giesecke & Devrient, Hitachi Ltd, IBM, JCB Co. Ltd, MasterCard Worldwide, NTT Corporation, NXP Seminconductors, Oberthur Card Systems, Renesas, SERMEPA, StepNexus, STMicroelectronics, Sun Microsystems Inc, Thales et Visa International.

Invariant inductif :

Un invariant est dit inductif s'il est vérifiable par induction : établi par l'initialisation et préservé par les événements. En B, on ne considère que des invariants inductifs.

JavaCard :

JavaCard est en même temps un systèmes d'exploitation pour carte à puce et un langage de développement, sous-ensemble du langage Java. Il est décrit en section 7.2.2.

JavaCard Runtime Environment :

Machine virtuelle pouvant interprété du bytecode JavaCard.

JML :

Le langage JML (Java Modelling Language) est un langage d'annotations permettant notamment de décrire les pré et post-conditions des différentes méthodes.

JTPL :

Le langage JTPL (Java Temporal Pattern Language) [TH02] est une extension de JML qui permet de décrire des propriétés temporelles pouvant être réécrites en JML.

Méthode B :

Méthode formelle de développement et langage de spécification basé sur l'affectation. Elle est décrite au chapitre 3.

Minimal :

Un **STES** est dit minimal (Définition 15, section 4.2.2.4) si et seulement si aucune de ses conditions de franchissement n'est en défaut de preuve (cas $(3')$ et $(6')$ du tableau 4.3).

Modèle :

Un modèle formel est la description d'un système, informatique ou non. Il peut être décomposé en plusieurs niveaux de raffinement et plusieurs modules.

Niveau de hiérarchie :

Dans un **STEH**, un niveau de hiérarchie est l'ensemble des états ayant la même profondeur ; c'est-à-dire le même nombre de super-états. Le niveau 0 est l'ensemble des états racine.

Occurrence libre d'une variable :

L'occurrence d'une variable x est dite *libre* dans un prédicat P si elle est présente dans P et qu'elle n'est pas sous la portée un quantificateur ($\exists$, $\forall$, λ, $\{x \mid \ldots\}$).

Occurrence non-libre d'une variable :

L'occurrence d'une variable x est dite *non libre* dans R si toutes les occurrences de x dans R sont *liées* ou si x est absente de R.

Occurrence liée d'une variable :

L'occurrence d'une variable x est dite *liée* dans R si toutes les occurrences de x dans R sont introduites par un quantificateur.

Ordinateur :

Moyen conçu pour accélérer et automatiser les erreurs *(Extrait du Nouveau dictionnaire du pirate).*

Partiellement franchissable :

Une transition est dite toujours franchissable si elle est franchissable depuis toutes les valuations de sont état d'origine. Sinon elle est dite partiellement franchissable.

Plus faible pré-condition :

Notée $\mathcal{WP}$ (Weakest Precondition), c'est la condition la plus faible telle que la substitution termine et vérifie la post-condition donnée (Section 3.3.1).

Profondeur :

Dans un **STEH**, un niveau de hiérarchie est l'ensemble des états ayant la même profondeur ; c'est-à-dire le même nombre de super-états. Le niveau 0 est l'ensemble des états racine.

Projet BOM :

Projet RNTL : B *Optimisant la mémoire*. Initialement lancé par GemPlus, il a été réalisé par Gemplus, Steria[a], l'équipe VASCO (LIG - Grenoble) et l'équipe TFC (LIFC - Besançon).

Ce projet vise à optimiser les programmes générés avec l'*AtelierB*, pour être embarqués sur des cartes à puce.

http://lifc.univ-fcomte.fr/~tatibouet/WEBBOM/

[a] Nouvellement ClearSy

Projet EDEMOI :

Projet de l'ACI sécurité : *Élaboration d'une DÉmarche et d'outils pour la MOdélisation Informatique, la validation et la restructuration de réglementations de « sûreté »(sécurité), et la détection des biais dans les aéroports. (Du 01/10/03 au 01/01/07).* Il a été réalisé en collaboration entre l'ONERA, le CNAM Paris, ENST Paris, l'université Paris 12, le LIFC et le LSR IMAG.

http://www-lsr.imag.fr/EDEMOI/

Projet GECCOO :

Projet de l'ACI sécurité :*Génération de code certifié pour des applications orientées objet. Spécification, raffinement, preuve et détection d'erreurs (Du 01/07/03 au 01/07/06).* Ce projet a été réalisé par l'équipe TFC (LIFC - Besançon), le projet CASSIS (LORIA - Nancy), le projet Everest(INRIA - Sophia Antipolis) le projet ProVal(LRI - Orsay et INRIA Futurs - Saclay) et l'équipe VASCO (LIG - Grenoble).

Il vise à proposer des méthodes et des outils pour le développement de programmes orientés objet ayant une forte composante sécurité. Un intérêt particulier est porté sur les programmes embarqués (cartes à puce, terminaux, etc). Le langage de programmation des exemples est le Java.

http://geccoo.lri.fr/

Projet POTESTAT :

Projet de l'ACI sécurité : *POlitiques de sécurité : TEST et Analyse par le Test de systèmes en réseau ouvert (Du 01/09/04 au 01/09/07).* Il a été réalisé par l'équipe VASCO (LIG - Grenoble), l'équipe SCD (VERIMAG - Grenoble) et les équipes Vertecs, Lande et DistribCom (IRISA - Rennes).

Ce projet s'intéresse à l'expression formelle de politiques de sécurité pour des logiciels d'un réseau et au test de ces logiciels.

http://www-lsr.imag.fr/POTESTAT/

Propriétés de sûreté :

Toute propriété de sécurité peut être décomposé en deux composantes : la sûreté et la vivacité [AS85]. La sûreté exprime le fait que rien de mauvais n'arrivera jamais. Ces propriétés sont dites invariantes et peuvent être conservées par raffinement.

Propriétés de vivacité :

Toute propriété de sécurité peut être décomposé en deux composantes : la sûreté et la vivacité [AS85]. La vivacité exprime le fait que quelque chose de bien arrivera fatalement un jour.

Raffinement :

La notion de raffinement est comparable à la modification de la résolution d'un appareil photos. Par exemple, prenons une photo ayant une taille de 640 points horizontaux par 480 points verticaux (nous noterons 640x480). Chaque point contient une information qui est sa couleur. En prenant la même photo en changeant uniquement la résolution de l'appareil en 1280x960, alors la nouvelle photo contient quatre fois plus d'information. Chaque point de la première est devenue 4 points dans la seconde, avec chacune une couleur, de telle sorte que la moyenne des quatre points donne la couleur du point de la première photo. On dira que la photo de faible résolution est une abstraction de la photo de forte résolution, alors appelée raffinement.

La raffinement est présenté en sections 1.2.2.2, 3.5 et 3.6.4.

Réduction :

Une condition de déclenchabilité ou d'atteignabilité est réductible à un prédicat P (Classiquement btrue ou bfalse) si celui-ci vérifie l'équation de caractéristique de la condition (Définition 13).

Remonter :

Une transition est remontée sur un super-état E si elle est représentée comme atteignant E, bien qu'elle soit définie comme atteignant le sous-état initial de E ou qu'elle prenne son origine dans le sous-état final de E.

Représentation exacte :

L'ensemble des séquences d'actions acceptées par un modèle sont appelées ses traces d'exécution. Un automate est une représentation de l'ensemble des comportements d'un modèle M si et seulement si chaque transition de l'automate est associée à l'exécution d'une action du modèle, et chaque état de l'automate est associé à une ou plusieurs configurations de M. Cette représentation est dite exacte si l'ensemble des traces d'exécution du modèle est égale à l'ensemble des chemins du système de transitions.

SEPL :

States/Events Properties Language. Langage permettant d'exprimer des propriétés portant sur des états et des événements [BPS05].

Sous-approximation :

Sous-approximer les comportements d'un modèle M consiste à caractériser un ensemble de séquences d'actions inclus dans les traces d'exécution de M. À l'inverse, sur-approximer ses comportements consiste à caractériser un ensemble de séquences d'actions contenant au moins les traces d'exécution de M.

Sous-état :

État contenu dans un état composé.

Sous-spécification :

Spécification trop faible pour pouvoir être prouvée directement par induction. Voir *Invariant inductif.*

Substitutions généralisées :

Dans le langage B, les instructions utilisées pour la spécification et le développement de programmes sont appelées des substitutions généralisées. L'assignation $x := E$ est la substitution simple. Les substitutions généralisées primitives sont décrites en section 3.3.1.

Super état :

État contenant des sous-états.

Sur-approximation :

Voir *Sous-approximation.*

Sûreté de fonctionnement :

La sûreté de fonctionnement a pour objectif de répondre aux attentes de disponibilité, fiabilité, sécurité-innocuité, confidentialité, intégrité et maintenabilité des logiciels critiques.

Système de transitions étiquetées Hiérarchique :

Système de transitions étiquetées symbolique où les états sont structurés par la fonction de hiérarchisation $\mathcal{S}up$. Celle-ci associe un super-état à chaque état. Ce choix permet de décrire aisément des transitions entre des états n'ayant pas le même super-état. Chaque état hiérarchique est donc un arbre d'états et la fonction $\mathcal{S}up$ forme une forêt.

Système de transitions étiquetées Symboliques :

Un système de transitions symbolique est un système de transitions hiérarchique dont aucun état n'est hiérarchisé ($\mathcal{S}up = \emptyset \ \wedge \ \mathbb{Q}_{Final} = \emptyset \ \wedge \ \mathbb{Q}_{Init} = \emptyset$). La définition est en section 4.1.2.

Toujours franchissable :

Une transition est dite toujours franchissable si elle est franchissable depuis toutes les valuations de sont état d'origine. Sinon elle est dite partiellement franchissable.

Trace d'exécution :

L'ensemble des séquences d'actions acceptées par un modèle sont appelées ses traces d'exécution. Un automate est une représentation de l'ensemble des comportements d'un modèle M si et seulement si chaque transition de l'automate est associée à l'exécution d'une action du modèle, et chaque état de l'automate est associé à une ou plusieurs configurations de M. Cette représentation est dite exacte si l'ensemble des traces d'exécution du modèle est égale à l'ensemble des chemins du système de transitions.

Transition externe :

Transition franchissant les limites d'un état composé.

Valuer :

Valuer une donnée (variable ou constante) consiste à lui associer une valeur. Nous serons également amenés à parler d'une configuration ou d'un état d'un modèle pour désigner une valuation de l'ensemble de ses données.

Index

APDU, 146, 185
API, 147, 185
Applet, voir Applette
Applette, 147, 185

bfalse, 52, 129, 185
btrue, 52, 129, 185
Bytecode, 147, 185

Cohérence, 9, 185
Comportements, 17, 26, 186
Conditions de franchissement
    Atteignabilité, 70, 185
    Déclenchabilité, 70, 186

Défaut de preuve, 38, 79, 186

État
    Cluster, voir état composé
    Correct, 157
    État composé, 31, 186
    État hiérarchique, voir état composé
    État-feuille, 31, 100, 186
    Incorrect, voir Correct
    Nomenclature, 27, 107
    Racine, 100, 186
    Sous-état, 31, 190
    Super état, 31, 190

Faisabilité, 72, 186
Forêt, 99, 186

Génie logiciel, 7, 186
GlobalPlatform, 147, 187

Invariant inductif, 156, 187

JavaCard, 146, 187
JavaCard Runtime Environment, 148, 187
JCRE, voir JavaCard Runtime Environment

Langages
    JavaCard, 45, 144, 187
    JML, 144, 187
    JTPL, 144, 187
    SEPL, 161, 190

Méthode B, 43, 187
Minimal, 79, 187
Modèle, 15, 187

Niveau de hiérarchie, 105, 187

Occurrence d'une variable
    Libre, 48, 188
    Liée, 48, 188
    Non-libre, 48, 188
Open Platform, voir GlobalPlatform

Partiellement franchissable, 28, 188
Plus faible pré-condition, 48, 49, 188
Profondeur, 105, 188
Projet
    BOM, 14, 45, 188
    EDEMOI, 139, 188
    GECCOO, 148, 155, 189
    POTESTAT, 139, 189
Propriétés de sécurité
    Sûreté, 10, 90, 189
    Vivacité, 10, 90, 189

Raffinement, 15, 54, 189
Réduction, 75, 190

Remonter, 103, 190
Représentation exacte, 26, 190

Sous-approximation, 11, 33, 190
Sous-spécification, 156, 190
STEH, voir Système de transitions étiquetées hiérarchique
STES, voir Système de transitions étiquetées symboliques
Substitutions généralisées, 48, 190
Sur-approximation, 11, 33, 190
Sûreté de fonctionnement, 6, 191
Système de transitions étiquetées
    Hiérarchique, 99, 191
    Symboliques, 71, 191

Toujours franchissable, 28, 191
Trace d'exécution, 26, 191
Transitions externes, 32, 191

Valuer, 25, 191

Weakest pre-condition, voir Plus faible pre-condition
WP, voir Plus faible pre-condition

Liste des illustrations

Figure 1.1	Exemples de défaillances de systèmes informatiques	6
Figure 1.2	Principe du cycle de développement en V	8
Figure 1.3	Processus général de développement selon la méthode B	16
Figure 1.4	Intégration de notre approche dans le processus de développement	17
Figure 1.5	Comportements d'une carte de paiement	19
Figure 1.6	Deux approches de vérification basées sur le raffinement	20
Figure 2.1	Exemple d'un système de transitions concret	27
Figure 2.2	Exemple d'un système de transitions symbolique	28
Figure 2.3	Ajout de gardes sur les transitions partiellement franchissables	30
Figure 2.4	Exemple de StateCharts	31
Figure 2.5	Choix de la granularité de représentation d'un StateCharts	32
Figure 2.6	Diagramme d'état transitions utilisant des sous-états finaux	33
Tableau 3.1	Opérateurs ensemblistes	44
Tableau 3.2	Constructeurs de relations ou de couples	44
Tableau 3.3	Opérateurs définis sur les relations	45
Tableau 3.4	Opérateurs de définition des fonctions	45
Spec 3.1	Machine de gestion des classes	46
Exemple 3.1	Substitution simple	48
Tableau 3.5	Substitutions primitives	49
Tableau 3.6	Quelques substitutions dérivées	50
Exemple 3.2	Calcul de $\mathcal{WP}$	50
Exemple 3.3	Forme normalisée	51
Figure 3.1	Forme générale d'une machine abstraite B	52

Spec 3.2-1 Structure de données du raffinement proposé 54
Spec 3.2-2 Initialisation raffinée 55
Spec 3.2-3 Opérations *InstanceOf* et *Charger* raffinées 55
Exemple 3.4 Raffinement de substitutions 56
Figure 3.2 Définition d'une machine M et de son raffinement N 57
Tableau 3.7 Obligations de preuve du raffinement 57
Spec 3.3 Canal de communication 59
Figure 3.3 Dessin informel représentant les comportements attendus 60
Spec 3.4 Raffinement de la spécification 3.3 61
Figure 4.1 Étapes menant à la construction des comportements d'un modèle B . . . 68
Spec 4.1 Canal de communication présenté en section 3.6.2 68
Figure 4.2 Représentation symbolique à un seul état du canal de communication . . 69
Figure 4.3 Seconde représentation avec un autre espace d'états 69
Figure 4.4 Canal de communication (avec conditions de franchissement) 71
Exemple 4.1 Forme normalisée d'un événement 72
Algo 1 Calcul de la relation de transition 76
Figure 4.4 Obligations de preuve de déclenchabilité 78
Tableau 4.2 Obligations de preuve d'atteignabilité 78
Tableau 4.3 Caractérisation des cas de défaut de preuve des conditions 79
Algo 2 Trouver D et A 79
Tableau 4.4 Coût de la génération d'un **STES** 80
Figure 4.5 Première vue du canal de communication 84
Figure 4.6 Seconde vue du canal de communication 84
Exemple 4.2 Construction par énumération de la variable *TailleEnvoi* 85
Exemple 4.3 Construction par utilisation des gardes 85
Exemple 4.4 Construction par choix aux limites 86
Exemple 4.5 Cycle de vie d'une classe 87
Exemple 4.6 Séquences d'opérations ou d'événements 90
Exemple 4.7 Traduction d'une machine en système 90
Figure 4.7 Séquences d'opérations autorisées dans l'exemple 4.7 92
Spec 4.2 Externalisation des paramètres 93

Figure 4.8 Mise en évidence de la condition permettant la réussite du chargement . 95
Spec 5.1 Rappel de la spécification 3.4, section 3.6.5 97
Figure 5.1 Transitions ne pouvant pas être factorisées 101
Figure 5.2 Exemple de transition factorisée 102
Figure 5.3 Exemple de transition en relation avec un sous-état feuille initial 103
Figure 5.4 Exemple de transition en relation avec un sous-sous-état initial 103
Figure 5.5 Exemple de représentation des différents types de transitions 104
Figure 5.6 Différentes granularités de représentation des états hiérarchiques. 105
Figure 5.7 Représentation hiérarchique du canal de communication. 106
Algo 3 Construction incrémentale de l'espace d'états de T_R 107
Algo 4 Trouver D et A – Exploitation de $\mathbb{R}_S$ 109
Algo 5 Trouver D et A – Gestion des nouveaux événements 110
Tableau 5.1 Coût de la génération d'un **STEH** 111
Figure 5.8 Calcul des conditions de franchissement d'une factorisation. 112
Algo 6 Factorisation des transitions 112
Algo 7 Construction de $\mathbb{Q}_{Init}$ 114
Figure 5.9 Cas particulier d'un unique sous-état : plusieurs représentations possibles 116
Algo 8 Réduction du nombre de transitions externes 116
Spec 5.2 Données du second raffinement 117
Figure 5.10 Comportements du second raffinement du canal de communication 118
Exemple 6.1 Déclaration des états de la figure 6.1 124
Algo 9 Factorisation du calcul des conditions de déclenchabilité 125
Algo 10 Trouver D et A – sans défaut de preuve 126
Exemple 6.2 Déclenchabilité de *Envoyer* depuis l'état $\mathit{TailleEnvoi} = 0$ 128
Figure 6.1 **STES** généré par *GénéSyst* pour la spécification 3.3 129
Figure 6.2 Description affinée d'une transition 130
Tableau 6.1 Nombre d'obligations de preuve (OP) vérifiées pour générer la figure 6.1. 130
Exemple 6.3 Définition des états de la figure 6.3 131
Figure 6.3 Représentation du canal de communication générée par *GénéSyst*. 132
Tableau 6.2 Nombre d'obligations de preuve pour calculer la figure 6.3 132
Tableau 6.3 Utilisation des techniques de choix d'espaces d'états dans les tests 133

Tableau 6.4 Les tests de développement de *GénéSyst* en chiffres 133
Figure 6.4 Comportements du modèle de la centrale de réservation 134
Figure 6.5 Comportements du modèle de l'écluse 134
Figure 6.6 Comportements du modèle de la machine à chocolat 135
Figure 6.7 Comportements du modèle du parking 136
Figure 6.8 Comportements du modèle de l'ordonnanceur 136
Figure 6.9 Comportements du modèle de canal de communication 137
Figure 7.1 Principales approches explorées dans GECCOO et outils réalisés ou utilisés 144
Figure 7.2 Approches que nous avons explorées dans GECCOO 145
Figure 7.3 Structure du modèle de DEMONEY 148
Tableau 7.1 Liste et rôle des méthodes que toute applette doit implanter 149
Tableau 7.2 Liste et rôle des instructions APDU reconnues par DEMONEY 149
Spec 7.1 Modèle simplifié de DEMONEY 150
Spec 7.2-1 Données du raffinement de DEMONEY 151
Spec 7.2-2 Événements du raffinement de DEMONEY 152
Figure 7.4 Comportements de DEMONEY 154
Tableau 7.3 Propriétés non-cryptographiques de DEMONEY 156
Figure 7.5 Utilisation d'un espace d'états incorrect. 158
Algo 11 Vérification du respect d'une propriété par un modèle 159
Figure 7.6 Représentation de la propriété (2) 160
Figure 7.7 Comportements de *Demoney* à partir de la propriété (2) 160
Exemple 7.1 Expression de la propriété *(2)* 161
Figure 7.8 Syntaxe des formules et des atomes du langage SEPL 162
Figure 7.9 Syntaxe des quantificateurs du langage SEPL 163
Algo 12 Validation d'un modèle par rapport à une propriété SEPL 163
Exemple 7.2 Reformulation de la propriété *(2)* 163
Algo 13 Validation syntaxique d'un modèle 165
Tableau A.1 Terminaison des principales substitutions 177
Tableau A.2 Prédicat avant-après des principales substitutions 177
Tableau A.3 Définition du $\mathcal{WP}^{cg}$ sur les principales substitutions primitives. 178

Liste des théorèmes, définitions, lemmes, conditions et propriétés

Déf 1 Chemins d'un système de transitions 26
Déf 2 Système de transitions concret 26
Déf 3 Système de transitions symbolique simple 28
Déf 4 Système de transitions symbolique avec conditions de franchissement et affectations 29
Déf 5 Chemins associés à une trace d'exécution 30
Déf 6 Axiomes du calcul de $\mathcal{WP}$ sur les substitutions primitives 49
Déf 7 Terminaison et prédicat avant/après d'une substitution 51
Déf 8 Forme normalisée d'une substitution 51
Déf 9 Raffinement des substitutions 56
Déf 10 Système de transitions étiquetées symbolique 71
Déf 11 Faisabilité d'une substitution 72
Déf 12 Forme normalisée d'un événement 72
Cond 1 Complétude de l'espace d'états 73
Cond 2 Correction de l'espace d'états 74
Cond 3 États non-vides 74
Déf 13 Caractérisation des conditions 75
Déf 14 Validité d'une transition 75
Déf 15 Minimalité d'un **STES** 79
Déf 16 Trace d'un système B événementiel 81
Lem 1 Caractérisation d'une trace 81
Déf 17 Franchissement d'une transition 81

Déf 18 Chemins d'un système de transitions étiquetées 82
Théo 1 Égalité des traces 82
Déf 19 Système de transitions hiérarchique 99
Cond 4 Cohérence de la hiérarchie 100
Déf 20 Transition factorisée 100
Déf 21 Sémantique d'une transition factorisée 101
Déf 22 Sémantique d'un **STEH** 102
Cond 5 Représentation d'une transition atteignant un sous-état initial 103
Cond 6 Représentation d'une transition partant d'un sous-état final 104
Déf 23 Projection d'un état 107
Cond 7 Cohérence incrémentale de la hiérarchie 108
Prop 1 Implication des conditions des transitions 109
Prop 2 Cas de réflexivité des nouveaux événements 110
Cond 8 Cas de factorisation 111
Cond 9 Contraintes sur le choix des sous-état initiaux ou finaux 114
Coro 2 Simplification des calculs d'atteignabilité 126
Déf 24 Raffinement structurel des systèmes de transitions étiquetées 159
Déf 25 *Enabled*, *AlwaysEnabled*, *Crossable* et *AlwaysCrossable* 162
Déf 26 Propriétés 162
Déf 27 Sémantique des quantificateurs introduits 163
Cond 10 Évaluation syntaxique des atomes à partir d'un **STES** minimal 164
Prop 3 Propriétés des prédicats sur la hiérarchie d'états 165
Déf 28 Forme normalisée d'un événement 179
Lem 2 Caractérisation d'une trace 180
Prop 4 Rappel de la propriété 1 180
Coro 4 Rappel du corollaire 1 182
Prop 5 Union des conditions d'atteignabilité pour un état et un événement 182

Bibliographie

[ABC$^+$02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-testing tools : A tool-set for test generation from Z and B using constraint logic programming. In *Formal Approaches to Testing of Software (FATES'02)*, pages 105–120. INRIA, 2002.
Cité en sections 1.2.2.1, 2.3.2.2, 3.7 et 7.1

[Abr80] J-R. Abrial. The Specification Language Z : Syntax and Semantics. Programming research group, Oxford University, 1980.
Cité en sections 1.2.2.1 et 3.3.3

[Abr96a] J.R. Abrial. Extending B without Changing it (for Developping Distributed Systems). In Nantes H. Habrias, editor, *First Conference on the B method*, pages 169–190, 1996.
Cité en sections 3.6 et 4.3.1

[Abr96b] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
Cité en sections 1.2.2.1, 1.3, 3.3.3, 3.4, 3.5.2.1, 3.5.2.2, 5.4 et 6.1.1

[AC03] Jean-Raymond Abrial and Dominique Cansell. Click'n Prove : Interactive Proofs within Set Theory. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
Cité en section 3.7

[AM98] J.R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*, pages 83–128. Springer-Verlag, 1998.
Cité en sections 2.4, 3.6, 4.1.3 et 4.1.3

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4) :181–185, October 1985 1985.
Cité en section C

[Ave05] Y. Avenel. Les OS pour cartes à puce entre ouverture et diversification. *Electronique*, 163 :34–38, 11 2005.
Cité en section 7.2.2

[BA05] F. Badeau and A. Amelot. Using B in a High Level Programming Language in an Industrial Project : Roissy VAL. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005 : Formal Specification and Development in Z and*

B, 4th International Conference of B and Z Users, volume 3455 of *Lecture Notes in Computer Science*, pages 334–354. Springer-Verlag, 2005.
Cité en section 1.3.1

[Bac78] R.J. Back. *On the Correctness of Refinement in Program Development*. Phd. thesis, University of Helsinki, 1978.
Cité en section 3.5

[BB02] H. Ruíz Barradas and D. Bert. Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In *Integrated Formal Methods : Third International Conference, IFM 2002, Turku*, volume 2335 of *Lecture Notes in Computer Science*, page 360, 2002.
Cité en section 2.4

[BB06] Héctor Ruíz Barradas and Didier Bert. Propriétés dynamiques avec hypothèses d'équité en B événementiel. *Technique et Science Informatiques, RSTI, série TSI*, 25(1) :73–102, 2006.
Cité en section 2.4

[BBB+04] F. Badeau, D. Bert, S. Boulmé, C. Métayer, M-L. Potet, N. Stouls, and L. Voisin. Traduction de B vers des langages de programmation. *Approches formelles pour le développement de logiciels*, 23(7) :879–903, 2004.
Cité en section 1.3.1

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J-M. Meynadier. Météor : A Successful Application of B in a Large Project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer-Verlag, 1999.
Cité en section 1.3.1

[BBH68] F. L. Bauer, L. Bolliet, and H. J. Helms. Software engineering. In Peter Naur and Brian Randell, editors, *Report on a conference sponsored by the NATO Science Committee (Garmisch, Germany)*, 1968.
Cité en section 1.2.1

[BBLV06] Didier Bert, Fabrice Bouquet, Yves Ledru, and Sylvie Vignes. Validation of Regulation Documents by Automated Analysis of Formal Models. In *International Workshop on Regulations Modelling and their Validation and Verification (REMO2V'06), in conjunction with CAiSE'06*. Presses Universitaires de Namur, 2006.
Cité en section 6.3.2

[BBP+03] D. Bert, S. Boulmé, M.L. Potet, A. Requet, and L. Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 94–113. Springer, 2003.
Cité en sections 1.3.1 et 7.1

[BC00] D. Bert and F. Cave. Construction of Finite Labelled Transition Systems from B Abstract Systems. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
Cité en sections 2.2, 2.2.2, 2.3.2.3, 2.5 et 4.1.1

[BC04] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions. Texts in Theoretical Computer Science*. Springer Verlag, 2004.
Cité en sections 1.2.2.1 et 7.1

[BCR03] L. Burdy, L. Casset, and A. Requet. Développement formel d'un vérificateur embarqué de byte-code Java. In D. Bert, V. Donzeau-Gouge, and H. Habrias, editors, *Développement rigoureux de logiciel avec la méthode B*, volume 22. Technique et Science Informatiques, 2003.
Cité en section 1.3.1

[BDJ+01] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. *Lecture Notes in Computer Science*, 2028 :302, 2001.
Cité en section 7.1

[BDJ+05] Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fähndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System : Challenges and Directions. In *International Conference on Verified Software : Theories, Tools, Experiments*, October 2005. Manuscript KRML 156.
Cité en section 1.2.2.1

[BDL05] Fabrice Bouquet, Frédéric Dadeau, and Bruno Legeard. How Symbolic Animation Can Help Designing an Efficient Formal Model. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005*, volume 3785 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2005.
Cité en section 2.3.2.2

[BDLU05a] Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and Mark Utting. JML-Testing-Tools : A Symbolic Animator for JML Specifications Using CLP. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 551–556. Springer, 2005.
Cité en section 2.3.2.2

[BDLU05b] Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and Mark Utting. Symbolic Animation of JML Specifications. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005 : Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005.
Cité en section 2.3.2.2

[BDM97] P. Behm, P. Desforges, and F. Mejia. Application de la méthode B dans l'industrie du ferroviaire. In *Application des techniques formelles au logiciel*. Observatoire Français des Techniques Avancées, 1997.
Cité en section 1.3.1

[BDM98] P. Behm, P. Desfroges, and J-M. Meynadier. Meteor : An industrial Success in formal Development. In *Second Conference on the B Method*, volume 1393 of *Lecture Notes in Computer Science*, page Invited Lecture. Springer-Verlag, 1998.
Cité en section 1.3.1

[BECN+04] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. In *Workshop on Fault Detection and Tolerance in Cryptography, Italy*, 2004.
Cité en section 7.2.3

[Beh96] Patrick Behm. Développement formel des logiciels sécuritaires de Météor. In Henri Habrias, editor, *Proceedings of the 1st Conference on the B method*, pages 3–10, November 1996.
Cité en section 1.3.1

[Ber06] Didier Bert. Modèle formel B de l'aéroport : Amdt11 Projet ACI Sécurité Informatique : EDEMOI. Rapport interne, décembre 2006.
Cité en section 6.3.2

[BF03] Jean-Paul Bodeveix and Mamoun Filali. Machines virtuelles pour le B événementiel. In Jean-Marc Jézéquel, editor, *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'03)*, pages 227–242, January 2003.
Cité en section 4.3.1

[BG01] C. Bidan and P. Girard. La securite des cartes a microprocesseur. *Revue de l'Electricité et de l'Electronique (REE)*, 5 :60–65, 5 2001.
Cité en section 7.2.3

[BH06] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods ...Ten Years Later. *Computer, IEEE Computer Society*, 39(1) :40–48, 2006.
Cité en section 1.2.3.1

[BJK00] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Ready-Simulation Is Not Ready to Express a Modular Refinement Relation. In T. S. E. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'2000), Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany*, volume 1783 of *Lecture Notes in Computer Science*, pages 266–283. Springer-Verlag, 2000.
Cité en section 4.3.1

[BL05] Michael J. Butler and Michael Leuschel. Combining CSP and B for Specification and Property Verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005 : Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2005.
Cité en sections 2.3.1 et 8.2

[BL07] Jens Bendisposto and Michael Leuschel. A Generic Flash-Based Animation Engine for ProB. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users*, volume 4355 of *Lecture Notes in Computer Science*, pages 266–269. Springer, 2007.
Cité en sections 1.4.3 et 2.1

[BLP02] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux. CLPS-B - A Constraint Solver for B. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2002.
Cité en section 2.3.2.2

[BLS05] M. Butler, M. Leuschel, and C. Snook. Tools for system validation with b abstract machines. In *ASM 2005 : 12th International Workshop on Abstract State Machines (in press)*, 2005.
Cité en section 8.2

[Boo94] Grady Booch, editor. *Object Oriented Analysis and Design, With Applications - second edition*. The Benjamin/Cummings Publishing Co., Inc., 1994.
Cité en section 2.2.3

[Bou03] Jean-Louis Boulanger. ABTOOLS : Another B Tool. In *3rd International Conference on Application of Concurrency to System Design (ACSD 2003)*, pages 231–232. IEEE Computer Society, 2003.
Cité en section 3.7

[BP00] P. Bontron and M-L. Potet. Automatic Construction of Validated B Components from structured Developments. In J. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000 : Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 127–147. Springer-Verlag, 2000.
Cité en section 6.4

[BP03] D. Bert and M-L. Potet. *Spécification en B, Support de cours de l'ENSIMAG et du DEA ISC*, 2003. Institut National Polytechnique de Grenoble.
Cité en sections 6.3.1 et 6.3.1

[BPS05] D. Bert, M-L. Potet, and N. Stouls. GénéSyst : a Tool to Reason about Behavioral Aspects of B Event Specifications. Application to Security Properties. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 299–318. Springer-Verlag, 2005.
Cité en sections 7.4.3, 7.4.3.3, 8.1 et C

[BR02] L. Burdy and A. Requet. JACK (Java Applet Correctness Kit). In *Gemplus Developer Conference*, 2002. http://www.gemplus.com/smart/r_d/trends/jack.html.
Cité en sections 7.1 et 7.4.3

[BR03] L. Burdy and A. Requet. Extending B with Control Flow Breaks. In D. Bert et al. [DJSM03], pages 513–527.
Cité en section 1.3.1

[But00] M. Butler. csp2B : A Practical Approach to Combining CSP and B. *Formal Aspects of Computing*, 12(3) :182–198, 2000.
Cité en sections 2.3.1, 3.3.3, 3.6, 3.7, 8.2 et A.3

[BWT02] D. Brewer, C. Wang, and P. Tsai. Proving Protection Profile Compliance for the CCL/ITRI Visa Open Platform Smart Card. In *3rd InternationalN Common Criteria Conference*, 2002.
Cité en sections 7.2.2 et C

[CA+04] D. Cansell, J-R Abrial, et al. B4free. *A set of tools for B development*, 2004.
http://www.b4free.com *(Valide au 20/11/06).*
Cité en section 3.7

[Cas02] L. Casset. *Construction Correcte de Logiciels pour Carte à Puce. Développement formel d'un vérifieur embarqué de byte code Java Card à l'aide de la méthode B.* PhD thesis, Université d'Aix-Marseille II, 2002.
http://www.atelierb.societe.com/liens/theseLudovic.pdf *(Valide au 20/11/06).*
Cité en section 1.3.1

[CBR02] L. Casset, L. Burdy, and A. Requet. Formal Development of an Embedded Bytecode Verifyier. In *International Conference on Dependable Systems & Networks (DSN'02)*, pages 51–58. IEEE Press, 2002.
Cité en section 1.3.1

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
Cité en sections 1.2.2.3 et 2.3.2.3

[CCK07] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. CC(X) : Efficiently Combining Equality and Solvable Theories without Canonizers. In *5th International Workshop on Satisfiability Modulo*, Berlin, Germany, July 2007.
Cité en sections 7.1 et 8.2

[CD07] Jean-François Couchot and Frédéric Dadeau. Guiding the correction of parameterized specifications. In Springer, editor, *Integrated Formal Methods*, Lecture Notes in Computer Sciences, 2007.
Cité en sections 7.4.1 et 8.2

[CDD+04] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and Debugging Set-Based Specifications. In A. Cavalcanti and P. Machado, editors, *WMF'03, Sixth Brazilian Workshop on Formal Methods*, volume 95 of *Electronic Notes in Theoretical Computer Science*, pages 189–208, Campina Grande, Brazil, May 2004.
Cité en sections 3.7 et 8.2

[CDGR04] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Barvey : Vérification automatique de consistance de machines abstraites B. In J. Julliand, editor, *AFADL'04, Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 369–372, Besançon, France, June 2004. Session outils.
Cité en sections 3.7 et 8.2

[CGK05] J.-F. Couchot, A. Giorgetti, and N. Kosmatov. A Uniform Deductive Approach for Parameterized Protocol Safety. In *ASE'05 : Procs of the 20th IEEE/ACM Int. Conf.*

on Automated Software Engineering, pages 364–367. IEEE Computer Society, 2005. isbn 1-59593-993-4.
Cité en section 7.4.1

[Cha88] K.M. Chandy. *Parallel program design : a foundation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1988.
Cité en section 2.4

[Châ01] Frédéric Châtel. Conception et réalisation d'éléments d'une boite à outils. Rapport de stage en laboratoire, licence de génie informatique, Université de la Rochelle / Laboratoire LSR, équipe SCOP, 2001. Tuteur : Frédéric Bertrand / Maitre de stage : Didier Bert.
Cité en sections 3.7 et 6.4

[Cle01] Clearsy. System Engineering Atelier B, Version 3.6, 2001.
http://www.atelierb.societe.com *(Valide au 22/03/07)*.
Cité en sections 1.3.1 et 3.7

[Cle05] ClearSy. http ://www.composys.fr/, 2005.
Cité en section 2.1

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Massachusetts, 1988.
Cité en section 2.3.2.3

[CMM00a] D. Cansell, D. Méry, and S. Merz. Predicate Diagrams for the Verification of Reactive Systems. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
Cité en sections 2.3.2.3 et 4.2.2.3

[CMM00b] D. Cansell, D. Méry, and S. Merz. Verifying Reactive Systems Using Predicate Diagrams. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods - Tools session*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
Cité en section 2.3.2.3

[CMM01] D. Cansell, D. Méry, and S. Merz. Diagram Refinements for the Design of Reactive Systems. *Journal of Universal Computer Science*, 7(2) :159–174, 2001.
Cité en sections 2.3.3 et 7.4.2

[Com99a] Common Criteria. *Common Criteria for Information Technology Security Evaluation - Version 2.1*, 1999.
Cité en section 1.1

[Com99b] Common Criteria. *Common Criteria for Information Technology Security Evaluation, Norme ISO 15408 - Part 1 : Introduction and General Model - Version 2.1*, Aout 1999. CCIMB-99-031.
Cité en section 7.1

[Com05] Common Criteria. *Common Criteria for Information Technology Security Evaluation - Version 3.0 Rev. 2*, 2005. CCMB-2005-07-001.
Cité en sections 1.1, 1.2.2.4 et 8.1

[Cou00] Patrick Cousot. Interprétation abstraite. *Technique et Science Informatique, Hermès*, 19(1-2-3) :155–164, Janvier 2000.
Cité en section 1.2.2.3

[CS01] E. M. Clarke and B-H. Schlingloff. Model Checking. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1635–1790. Elsevier and MIT Press, 2001.
Cité en section 7.4

[CSBSD01] Janette Cardoso, Christophe Sibertin-Blanc, and Chantal Soulé-Dupuy. Une sémantique formelle des diagrammes d'interaction d'UML via les réseaux de Petri. In R. Valette G. Juanolle, editor, *Colloque Francophone sur la Modélisation des Systèmes Réactifs (MSR'01)*, pages 497–512, Toulouse, France, octobre 2001. Hermes.
Cité en section 2.2

[CU89] W. Chen and J. T. Udding. Toward a Calculus of Data Refinement. In Jan L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 197–218, 1989.
Cité en section 3.5.2.1

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, The Netherlands, 1996.
Cité en section 2.3.2.3

[DCS01] Direction centrale de la sécurité des systèmes d'information DCSSI. *Carte mixe MONEO/CB, rapport de certification 2001/10*, Octobre 2001.
Cité en section 7.2.1

[DD04] C. Demoulin and M. Van Droogenbroeck. Principes de base du fonctionnement du réseau GSM. *Bulletin scientifique - Association des ingénieurs électriciens sortis de l'Institut életrotechnique Montefiore*, 4 :3–18, 2004.
Cité en section 7.2.1

[DFG+05] V. Darmaillacq, J.-C. Fernandez, R. Groz, L. Mounier, and J.-L. Richier. Eléments de modélisation pour le test de politiques de sécurité. In *Colloque sur les RIsques et la Sécurité d'Internet et des Systèmes, CRiSIS, Bourges, France*, 2005.
Cité en section 6.3.2

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
Cité en sections 3.3, 3.5 et 8.2

[Dil94] A. Diller. *Z : an introduction to formal methods*. Wiley, 1994.
Cité en sections 1.2.2.1 et 3.3.3

[DJSM03] D. Bert, J.P. Bowen, S. King, and M. Waldén, editors. *ZB 2003 : Formal Specification and Development in Z and B, Third International Conference of B and Z Users*, volume 2651 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
Cité en section C

[DM94] B. Dehbonei and F. Meijia. Formal Methods in the Railways Signalling Industry. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94 : Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe*, volume 873 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
Cité en section 1.3.1

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *Journal of the ACM*, 52(3) :365–473, 2005.
Cité en sections 7.1 et 8.2

[Dun03] S. Dunne. Introducing Backward Refinement into B. In D. Bert et al. [DJSM03], pages 178–196.
Cité en section 8.2

[EGK+01] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer, 2001.
Cité en section 6.4

[Fil03] J.-C. Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
Cité en sections 7.1 et 8.2

[Fla05] M. Flamenbaum. La carte nationale d'identité électronique. Rapport de master 2 droit du cyberspace, Université de Lille 2 Faculté des sciences juridiques politiques et sociales, 2005.
Cité en section 7.2.1

[Fra86] N. Francez. *Fairness*. Springer-Verlag, 1986.
Cité en section 2.4

[GBJ06] T.Le Gall and H. Marchand B. Jeannet. Contrôle de systèmes symboliques, discrets ou hybrides. *Technique et Science Informatiques, RSTI, série TSI*, 25(3) :293–319, 2006.
Cité en section 2.2.2

[GEC07] GECCOO. Génération de code certifié pour des applications orientées objet. spécification, raffinement, preuve et détection d'erreurs. Rapport final - http ://geccoo.lri.fr, 2007.
Cité en section 7.1

[GFL07] Frédéric Gervais, Marc Frappier, and Régine Laleau. Refinement of eb3 Process Patterns into B Specifications. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users*, volume 4355 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2007.
Cité en sections 2.3.1 et 8.2

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
Cité en section 1.2.2.1

[GM93] P. H. B. Garnier and C. Morgan. A Single Complete Rule for Data Refinement. In *Formal Aspects of Computing*, pages 367–392, 1993.
Cité en section 3.5.2.1

[GN99] E. R. Gannsner and S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. Technical report, Laboratoires AT&T, Etats unis, 1999.
Cité en section 6.4

[GP85] D. Gries and J. Prins. A new Notion of Encapsulation. In *Symposium on Languages Issues in Programming Environments, SIGLPAN*, 1985.
Cité en section 3.5.2.1

[Gro07] Julien Groslambert. Verification of LTL on B Event Systems. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users*, volume 4355 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2007.
Cité en section 2.4

[GS97] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Computer-Aided Verification (CAV'97)*, volume 1254 of *LNCS*. Springer-Verlag, 1997.
Cité en sections 2.2, 2.2.2 et 2.3.2.3

[Gur85] Y. Gurevich. A new thesis. *American Mathematical Society Abstracts*, 6(4) :68–203, 1985.
Cité en section 2.2.2

[Hab01] Henry Habrias. Préface de Fernando Méjia. In *Spécification formelle avec B*. Hermès Science Publications, 2001.
Cité en section 1.3.1

[Ham02] Smaïne Hamdane. Génération de systèmes de transition étiquetés à partir de la description d'un système d'évènements décrits avec le langage B. Rapport de licence, Université Joseph Fourrier, Grenoble 1, France, mai 2002.
Cité en sections 4.4.3 et 6.4

[Ham03] Smaïne Hamdane. Système de transitions d'un système abstrait : méthode de calcul des états. Rapport de maîtrise, Université Joseph Fourrier, Grenoble 1, France, 2003.
Cité en sections 4.4.3 et 6.1.1

[Har87] David Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.
Cité en sections 2.2, 2.2.3 et 4.1.2

[Hes92] W. H. Hesselink. Programs, Recursion and Unbounded Choice. *Cambridge Tracts in Theoretical Computer Science*, 27, 1992.
Cité en section 8.2

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10) :576–580, 1969.
Cité en section 3.3

[Hoa78] C.A.R. Hoare. Communicating Sequential Processing. *Communication of ACM*, 21(8) :666–677, 1978.
Cité en sections 1.4.2, 2.2 et 2.3.1

[Hus01] D. Husemann. Standards in the Smart Card World. *Computer Networks*, 36(4) :476–487, 2001.
Cité en sections 7.2.1 et C

[HWS00] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL : Toward a Standard Exchange Format. In *Seventh Working Conference on Reverse Engineering (WCRE'00), 23-25 November 2000, Brisbane, Australia.*, pages 162–171. IEEE Computer Society, 2000.
Cité en section 6.4

[Ida06a] Akram Idani. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B*. PhD thesis, Université Joseph Fourier, 2006.
Cité en section 4.4.3

[Ida06b] Akram Idani. Couplage de spécifications B et de descriptions UML pour l'aide aux développements formels des Systèmes d'Information. In *Actes du XXIVème Congrès INFORSID*, pages 577–593, Hammamat, Tunisie, 2006.
Cité en section 4.4.3

[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, Mars 2006. Elsevier.
Cité en sections 1.4.3 et 4.4.3

[JCI+05] J.Coleman, C.Jones, I.Oliver, A.Romanovsky, and E.Troubitsyna. Rodin (rigorous open development environment for complex systems). project number ist 2004-511599. In *Fifth European Dependable Computing Conference : EDCC-5 supplementary volume.*, pages 23–26, 2005.
Cité en section 8.2

[Jon86] C.B. Jones. *Systematic Software Developpement using VDM*. Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK, 1986.
Cité en section 3.3.3

[LAB+01] Michael Leuschel, Laksono Adhianto, Michael Butler, Carla Ferreira, and Leonid Mikhailov. Animation and Model Checking of CSP and B using Prolog Technology. In Michael Leuschel, Andreas Podelski, C.R. Ramakrishnan, and Ulrich Ultes-Nitsche, editors, *Second International Workshop on Verification and Computational Logic (VCL'2001)*, pages 97–109, 2001.
Cité en section 2.3.2.2

[Lam94a] L. Lamport. A Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, may 1994.
Cité en section 3.3.3

[Lam94b] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, may 1994.
Cité en sections 1.2.2.1, 1.3.3, 2.3.2.3 et 3.6.4

[Lam95] Leslie Lamport. TLA in Pictures. *IEEE Transactions on Software Engineering*, 21(9) :768–775, 1995.
Cité en section 2.3.2.3

[Lan05] Arnaud Lanoix. *Systèmes à composants synchronisés : contribution à la vérification compositionnelle du raffinement et des propriétés*. PhD thesis, UFR des sciences et techniques de l'université de Franche-Comté, 2005.
Cité en section 8.2

[Lap95] J-C. Laprie, editor. *Guide de la Sûreté de Fonctionnement*. Cépaduès, 1995.
Cité en section 1.1

[LB03] M. Leuschel and M. Butler. ProB : A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.
Cité en sections 2.2, 2.2.1, 2.3.2.2, 2.3.2.2 et 3.7

[LBR98] G.T. Leavens, A.L. Baker, and C. Ruby. JML : a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, 1998.
Cité en sections 1.2.2.1 et 7.1

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML : a notation for deltailed design. In Haim Kilov, Bernhard Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publisher, 1999.
Cité en sections 1.2.2.1 et 7.1

[LCGP89] J-C. Laprie, B. Courtois, M-C. Gaudel, and D. Powel. *Sûreté de Fonctionnement des Systèmes Informatiques Matériels et Logiciels*. DUNOD informatique, 1989.
Cité en section 1.1

[Leb00] J. Lebray. Modélisation de Systèmes en B : Proposition de guides méthodologiques pour la décomposition d'événements. Rapport de DEA, Institut National Polytechnique de Grenoble, France, 2000.
Cité en section 4

[Liv78] C. Livercy. *Théorie des programmes*. Dunod, Paris, 1978.
Cité en section 1.4.2

[LL98] J-L Lanet and P Lartigue. The Use of Formal Methods for SmartCards, a Comparison between B ans SDL to Model the T=1 Protocol. In *Proceedings of International Workshop on Comparing Systems Specification Techniques*, Nantes, March 1998.
Cité en section 1.3.1

[LP01] Regine Laleau and Fiona Polack. A Rigorous Metamodel for UML Static Conceptual Modelling of Information Systems. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001*, volume 2068 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2001.
Cité en section 4.4.3

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME 2002 : Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002.
Cité en sections 6.3.1 et 6.3.1

[LR98] J-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference (CARDIS '98)*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
Cité en section 1.3.1

[LT05] Michael Leuschel and Edd Turner. Visualising Larger State Spaces in Pro B. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005 : Formal Specification and Development in Z and B, 4th International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 6–23, 2005.
Cité en sections 1.4.3, 2.2.1 et 2.3.2.2

[Mar91] F. Maraninchi. The Argos language : Graphical Representation of Automata and Description of Reactive Systems. In *IEEE Workshop on Visual Languages*, oct 1991.
Cité en section 5.1.1.3

[Mar92] F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In Rance Cleaveland, editor, *CONCUR'92, Third International Conference on Concurrency Theory*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564. Springer, 1992.
Cité en section 5.1.1.3

[Mar02] R. Marlet. DEMONEY : Java Card Implementation. Public technical report, SECSAFE project, 11 2002.
Cité en section 7.3

[mdd84] Arrêté ministériel du 30 décembre 1983. Enrichissement du vocabulaire de l'informatique. *Journal officiel*, 19 février, 1984.
Cité en sections 1.2.1 et C

[Mer00] A. Merle. *Evaluation des produits suivant les critères communs*, 2000.
Cité en section 8.1

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.
Cité en section 2.2

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.
Cité en sections 6.3.1 et 6.3.1

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical Automata as Model for Statecharts. In R. Shyamasundar and K. Ueda, editors, *Advances in Computing Science - ASIAN '97, Third Asian Computing Science Conference*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, December 1997.
Cité en sections 5.1.1.1 et 5.1.1.3

[MM01] R. Marlet and D. Le Metayer. Security Properties and Java Card Specificities To Be Studied in the SecSafe Project. Public technical report, SECSAFE project, 08 2001.
Cité en sections 7.3.2 et 7.3.2

[MM02] R. Marlet and C. Mesnil. DEMONEY : A demonstrative Electronic Purse - Card Specification. Public technical report, SECSAFE project, 11 2002.
http://www.doc.ic.ac.uk/~siveroni/secsafe/docs/secsafe-tl-007-0.8.pdf *(Valide au 20/11/06)*.
Cité en sections 6.3.2, 7.3, 7.3.2 et 8.1

[MMJ00] P.A. Masson, H. Mountassir, and J. Julliand. Modular verification for a class of PLTL properties. In *2nd international conference on Integrated Formal Methods (IFM)*, volume 1945 of *Lecture Notes in Computer Science*, pages 398–419. Springer Berlin / Heidelberg, 2000.
Cité en sections 2.4 et 8.2

[Moc06] John Mocenigo. Grappa : A Java Graph Package. http ://www.research.att.com/ john/Grappa/, 2006.
Cité en section 6.4

[Moh04] Hounayda Mohamed. Extension de l'outil GénéSyst. Rapport de t.e.r, Université Joseph Fourier / Laboratoire LSR, équipe VASCO, rue de la Passerelle, 38402 St Martin d'hères, août 2004. Encadrants : Nicolas Stouls et Didier Bert.
Cité en sections 6 et 6.4

[Mor04] Xavier Morselli. Vérification et optimisation de l'outil GénéSyst. Rapport de t.e.r, Université Joseph Fourier / Laboratoire LSR, équipe VASCO, rue de la Passerelle, 38402 St Martin d'hères, mai 2004. Encadrants : Nicolas Stouls et Didier Bert.
Cité en sections 6 et 6.4

[Mot00] S. Motré. A B automaton for Authentification Process. In *WITS : Workshop on Issues in the Theory of Security*, Genève, Suisse, 2000.
Cité en section 1.3.1

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *J. Log. Algebr. Program.*, 58(1-2) :89–106, 2004.
Cité en sections 7.1 et 7.4.3

[MPS04] Xavier Morselli, Marie-Laure Potet, and Nicolas Stouls. GénéSyst : Génération d'un système de transitions étiquetées à partir d'une spécification B événementiel. In J. Julliand, editor, *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'04) - Session outils*, pages 317–320, June 2004.
Cité en sections 3.7, 6, 6.4 et 8.1

[MR01] Florence Maraninchi and Yann Remond. Argos : an automaton-based synchronous language. *Computer Languages*, 27(1-3) :61–92, 2001.
Cité en section 5.1.1.3

[MS99] Eric Meyer and Jeanine Souquières. A Systematic Approach to Transform OMT Diagrams to a B Specification. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 875–895. Springer, 1999.
Cité en sections 2.3.1 et 8.2

[MT00] S. Motré and C. Téri. Using Formal and Semi-Formal Methods for a Common Criteria Evaluation. In *EUROSMART*, Marseille, France, juin 2000.
Cité en section 1.3.1

[Nah01] J. Nahoum. Outils d'assistance à la construction de systèmes dans la méthode B. Rapport de DEA, Institut National Polytechnique de Grenoble, France, 2001.
Cité en section 4

[Oes99] M. Oestreicher. Transactions in Java Card. In *Computer Security Applications Conference (ACSAC '99)*, pages 291–298, 1999.
Cité en section 7.2.3

[OJS05] Dieu Donné Ossami, Jean-Pierre Jacquot, and Jeanine Souquières. Consistency in UML and B Multi-view Specifications. In Judi Romijn, Graeme Smith, and Jaco van de Pol, editors, *Integrated Formal Methods, 5th International Conference, IFM 2005*, volume 3771 of *Lecture Notes in Computer Science*, pages 386–405, Eindhoven, The Netherlands, December 2005. Springer.
Cité en section 8.2

[OMG01] OMG. *Unified Modeling Language Specification*, septembre 2001. Version 1.4.
Cité en sections 2.2 et 2.2.3

[Par74] D. M.R. Park. Finiteness is Mu-ineffable. Technical report, University of Warwick, Coventry, UK, UK, 1974.
Cité en sections 1.4.2 et 2.2

[Par00] B. Parreaux. *Vérification de systèmes d'événements B par model-checking PLTL*. Thèse de doctorat, LIFC, Université de Franche-Comté, Décembre 2000. Rapporteurs : H. Habrias (Nantes), D. Mery (Nancy 1). Examinateurs : F. Bellegarde, J. Julliand, P. Schnoebelen (LSV - ENS Cachan). Directeur : J. Julliand.
Cité en sections 2.4 et 7.4

[PLN03] C. Poerschke, D.E. Lightfoot, and J.L. Nealon. A Formal Specification in B of a Medical Decision Support System. In D. Bert et al. [DJSM03], pages 497–512.
Cité en section 1.3.1

[PM93] C. Paulin-Mohring. Inductive Definitions in the system Coq - Rules and Properties. In *TLCA'93 : Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag.
Cité en section 1.2.2.1

[PP03] G. Pouzancre and J-P. Pitzalis. Modélisation en B événementiel des fonctions mécaniques, électriques et informatiques d'un véhicule. *TSI : Méthode B*, 22 :119–128, 2003.
Cité en sections 1.3.1 et 1.3.2

[PPS06] Florent PATIN, Guilhem Pouzancre, and Thierry Servat. Approche formelle pour la réalisation d'un système sécuritaire de contrôle commande de façades de quais. In *4ème Conférence Annuelle d'Ingénierie Système, Efficacité des entreprises et satisfaction des clients (AFIS'06)*, 2006.
Cité en section 2.1

[PS04] Marie-Laure Potet and Nicolas 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, June 2004.
Cité en sections 6, 6.3.1, 6.3.1 et 8.1

[RB00] A. Requet and G. Bossu. Embedding Formally Proved Code in a Smart Card : Converting B to C. In *Third IEEE International Conference on Formal Engineering Methods (ICFEM'00)*, pages 15–22, 2000.
Cité en section 1.3.1

[RD03] S. Ranise and D. Deharbe. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In *1st International Conference on Software Engineering and Formal Methods (SEFM' 20'03)*, pages 220–228. IEEE Computer Society Press, 2003.
Cité en sections 3.7, 7.1 et 8.2

[Req00] A Requet. A B Model for Ensuring Soundness of the Java Card Virtual Machine. In *FMICS'2000*, Berlin, March 2000.
Cité en section 1.3.1

[Rou99] Y. Rouzaud. Interpreting the B-Method in the Refinement Calculus. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 411–430, 1999.
Cité en section 3.5.2.1

[SC-92] RTCA Committee SC-176. *Software Considerations in Airbornbe systems and Equipment Certification. Draft DO-178-B.7.* RTCA (Requirements and Technical Concepts for Aviation), 1140 Connecticut Ave, NW, Suite 1020, Washington DC 20036, 1992.
Cité en section 1.1

[SD05] N. Stouls and V. Darmaillacq. Développement formel d'un moniteur. In S. Saget and A. Vautier, editors, *Majecstic : 3ème manifestation des jeunes chercheurs en Sciences et Technologies de l'Information et de la Communication.*, pages 397–401. Imprimerie de l'université, 2005.
Cité en sections 6.3.2 et 8.1

[SD06] Nicolas Stouls and Vianney Darmaillacq. Développement formel d'un moniteur détectant les violations de politiques de sécurité de réseaux. In S. Vignes and V. Viguié Donzeau-Gouge, editors, *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'06)*, pages 179–193, March 2006.
Cité en sections 6.3.2 et 8.1

[Sec99] SecSafe. SecSafe Project Home Page, 1999.
http://www.doc.ic.ac.uk/~siveroni/secsafe/ *(Valide au 20/11/06).*
Cité en section 7.3

[Ser06] Thierry Servat. BRAMA : A New Graphic Animation Tool for B Models. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007 : Formal Specification and Development in B, 7th International Conference of B Users*, volume 4355 of *Lecture Notes in Computer Science*, pages 274–276. Springer, 2006.
Cité en section 2.1

[SG02] E. Soulier and C. Grenier. A Political Model for the Co-operative Production of Knowledge in the Design process : the Shared Medical File (SMF). In *ECAI'02 Workshop on Knowledge Management and Organizational Memories*, 2002.
Cité en section 7.2.1

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computer Survey*, 25(3) :225–262, 1993.
Cité en section 2.4

[SL99] D. Sabatier and P. Lartigue. The Use of the B Formal Method for the Design and the Validation of the Transaction Mechanism for Smart Card Applications. In *FM'99 - Formal Methods -volume 1*, volume 1708 of *Lecture Notes in Computer Science*, pages 348–387. Springer-Verlag, september 1999.
Cité en sections 1.3.1 et 7.2.3

[SL00] D. Sabatier and P. Lartigue. The Use of the B Formal Method for the Design and the Validation of the Transaction Mechanism for Smart Card Applications. *Formal Methods in System Design*, 17(3) :245–272, 12 2000.
Cité en sections 1.3.1 et 7.2.3

[SP07] Nicolas Stouls and Marie-Laure Potet. Security Policy Enforcement Through Refinement Process. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 216–231. Springer-Verlag, 2007.
Cité en sections 6.3.2 et 8.1

[Spi93] J.M. Spivey. *The Z notation : A Reference Manual*. Prentice Hall, 1993. Second edition.
Cité en sections 1.2.2.1 et 3.3.3

[ST05] S. Schneider and H. Treharne. CSP Theorems for Communicating B Machines. *Formal Aspects of Computing*, 17(4) :390–422, 2005.
Cité en sections 2.3.1 et 8.2

[Sto02] Nicolas Stouls. Compte rendu de stage de maîtrise, Chapitre 4 : La Boîte à outils B. Master's thesis, Université Joseph Fourier / Laboratoire LSR, équipe VASCO, rue de la Passerelle, 38402 St Martin d'hères, juillet 2002. Encadrant : Marie-Laure Potet.
Cité en sections 3.7 et 6.4

[Sto06a] Nicolas Stouls. Aide à la spécification et au développement formel de systèmes. 16ème rencontres régionales de la recherche en rhône-alpes, 2006. Poster.
Cité en sections 6 et 8.1

[Sto06b] Nicolas Stouls. Introduction aux cartes à puce. Rapport technique, LSR-IMAG, Grenoble, France, Septembre 2006.
http://www-lsr.imag.fr/users/Nicolas.Stouls/Productions/CartesAPuce/CartesAPuce.ps.gz.
Cité en sections 7.2 et 8.1

[Sys] Siemens Transportation Systems. http://www.siemens-ts.com/. *(Valide au 20/11/06)*.
Cité en section 1.3.1

[TH02] K. Trentelman and M. Huisman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, LNCS 2422, pages 334–348. Springer-Verlag, 2002.
Cité en sections 7.1, 7.4.3 et C

[TS06] Ninh-Thuan Truong and Jeanine Souquières. Verification of UML Model Elements Using B. *Journal of Information Science and Engineering*, 22(2) :357–373, 2006.
Cité en section 2.3.1

[Voi04] Jean-Christophe Voisinet. *Contribution au processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers des langages à objets*. PhD thesis, Université de Franche-Comté, spécialité automatique et informatique, 2004.
Cité en sections 2.2, 2.3.2.1, 2.3.3 et 4.4.2

[VT03] J.-C. Voisinet and B. Tatibouet. Generating Statecharts from B Specifications. In *16th Int Conf. on Software and System Engineering and their applications (ISCEA 2003)*, volume 1, 2003.
Cité en sections 2.3.2.1 et 2.3.3

[VTH02] J-C. Voisinet, B. Tatibouet, and A. Hammad. jBTools : An Experimental Platform for the Formal B Method. In *PPPJ'02*, pages 137–140. Trinity College, Dublin, Ireland, Juin 2002.
Cité en sections 3.7 et 6.4

[WADJ90] W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors. *Beauty is our Business : A Birthday Salute to Edsger W. Dijkstra*, chapter C. Morgan : Of wp and CSP. Springer-Verlag, 1990.
Cité en sections 3.3.3 et A.3

[Yan00] Mihalis Yannakakis. Hierarchical State Machines. In Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito, editors, *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2000.
Cité en section 5.1.1.1