Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information

Jérémy Milhau

▶ **To cite this version:**

Jérémy Milhau. Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information. Autre [cs.OH]. Université Paris-Est, 2011. Français. NNT : 2011PEST1038 . tel-00674865

HAL Id: tel-00674865

https://theses.hal.science/tel-00674865

Submitted on 28 Feb 2012

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

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

UNIVERSITÉ PARIS-EST
ÉCOLE DOCTORALE MSTIC

UNIVERSITÉ DE SHERBROOKE
Faculté des sciences, Département informatique

Thèse de doctorat en informatique réalisée en co-tutelle

Jérémy MILHAU

Un processus formel d'intégration de politiques de contrôle d'accès dans les systèmes d'information

Thèse soutenue le 12 décembre 2011

Composition du Jury

Rapporteurs
Dominique MÉRY - Université Henri Poincaré Nancy 1
Yamine AIT-AMEUR - ENSEEIHT Toulouse

Co-directeurs
Régine LALEAU - Université Paris-Est
Marc FRAPPIER - Université de Sherbrooke
Frédéric GERVAIS - Université Paris-Est

Examinateurs
Marie-Laure POTET - ENSIMAG Grenoble
Pierre-Yves SCHOBBENS - Université de Namur
Richard ST-DENIS - Université de Sherbrooke

Résumé

La sécurité est un élément crucial dans le développement d'un système d'information. On ne peut pas concevoir un système bancaire sans préoccupation sécuritaire. La sensibilité des données d'un système hospitalier nécessite que la sécurité soit la composante majeure d'un tel logiciel. Le contrôle d'accès est un des nombreux aspects de la sécurité. Il permet de définir les conditions de l'exécution d'actions dans un système par un utilisateur. Entre les différentes phases de conception d'une politique de contrôle d'accès et son application effective sur un système déployé, de nombreuses étapes peuvent introduire des erreurs ou des failles non souhaitables. L'utilisation de méthodes formelles est une réponse à ces préoccupations dans le cadre de la modélisation de politiques de contrôle d'accès. L'algèbre de processus EB^3 permet une modélisation formelle de systèmes d'information. Son extension EB^{3}SEC a été conçue pour la spécification de politiques de contrôle d'accès. Le langage ASTD, combinaison des statecharts de Harel et des opérateurs de EB^3, permet de modéliser graphiquement et formellement un système d'information. Cependant, ces deux méthodes manquent d'outils de vérification et de validation qui permettent de prouver ou de vérifier des propriétés de sécurité indispensables à la validation de politiques de contrôle d'accès. De plus, il est important de pouvoir prouver que l'implémentation d'une politique correspond bien à sa spécification abstraite. Cette thèse définit des règles de traduction de EB^3 vers ASTD, d'ASTD vers Event-B et vers B. Elle décrit également une architecture formelle exprimée en B d'un filtre de contrôle d'accès pour les systèmes d'information. Cette modélisation en B permet de prouver des propriétés à l'aide du prouveur B ou de vérifier des propriétés avec ProB, un vérificateur de modèles. Enfin, une stratégie de raffinement B pour obtenir une implémentation de ce filtre de contrôle d'accès est présentée. Les raffinements B étant prouvés, l'implémentation correspond donc au modèle initial de la politique de contrôle d'accès.

Mots-clés : Système d'information, contrôle d'accès, B, Event-B, ASTD, EB^3.

Abstract

A Formal Integration of Access Control Policies into Information Systems.

Security is a key aspect in information systems (IS) development. One cannot build a bank IS without security in mind. In medical IS, security is one of the most important features of the software. Access control is one of many security aspects of an IS. It defines permitted or forbidden execution of system's actions by a user. Between the conception of an access control policy and its effective deployment on an IS, several steps can introduce unacceptable errors. Using formal methods may be an answer to reduce errors during the modeling of access control policies. Using the process algebra EB^3, one can formally model IS. Its extension, EB^{3}SEC, was created in order to model access control policies. The ASTD notation combines Harel's Statecharts and EB^3 operators into a graphical and formal notation that can be used in order to model IS. However, both methods lack tools allowing a designer to prove or verify security properties in order to validate an access control policy. Furthermore, the implementation of an access control policy must correspond to its abstract specification. This thesis defines translation rules from EB^3 to ASTD, from ASTD to Event-B and from ASTD to B. It also introduces a formal architecture expressed using the B notation in order to enforce a policy over an IS. This modeling of access control policies in B can be used in order to prove properties, thanks to the B prover, but also to verify properties using ProB, a model checker for B. Finally, a refinement strategy for the access control policy into an implementation is proposed. B refinements are proved, this ensures that the implementation corresponds to the initial model of the access control policy.

Keywords: Information system, access control, B, Event-B, ASTD, EB^3.

Remerciements

À tous ceux qui m'ont un jour dit « Tu n'y arriveras pas » ainsi qu'à tous ceux qui m'ont dit « Vise le meilleur, puis vise un peu plus haut », je dédie cette thèse aux premiers en tant que pied de nez et aux seconds en guise de résultat.

Une thèse est le travail d'une personne au sein d'une équipe, soutenu par ses amis et ses proches. C'est le travail le plus collectif qui m'ait été donné à réaliser mais surtout le plus enrichissant. Je remercie donc infiniment tous les acteurs de ce formidable processus.

Je tiens particulièrement à remercier mes co-directeurs de thèse pour leur encadrement, leurs compétences et leur présence tout au long de ces mois de recherche et de rédaction. Si la recherche n'avance pas sans le travail des thésards, le travail du thésard n'avance pas sans ses directeurs. Je remercie le professeur Marc Frappier qui m'a prolongé sa confiance et proposé cette aventure en 2008. La sympathie et le dynamisme de Marc sont des atouts dans les phases difficiles, son amitié et sa bonne humeur un agréable accompagnement le reste du temps. Je remercie également la professeure Régine Laleau pour m'avoir proposé d'effectuer cette thèse en co-tutelle. Régine est d'une aide précieuse dans les phases de rédaction d'article de part son recul sur les travaux menés et ses remarques toujours constructives. Son dynamisme permet également de toujours aller de l'avant et de motiver toute une équipe quand cela est nécessaire. Finalement, je remercie mon troisième directeur, Frédéric Gervais pour son suivi régulier de mes travaux, sa bonne humeur, son humour et tous les repas que nous avons partagés. Bien que sa présence fut moins importante du fait de ses tâches administratives lors des derniers mois, j'espère que mes travaux lui rappelleront combien la recherche est plus intéressante et captivante que les problématiques bureaucratiques.

Je souhaite également remercier le professeur Richard St-Denis pour son suivi rigoureux et ses remarques toujours pertinentes visant à améliorer la qualité de mes communications scientifiques. Merci à Benoît Fraikin pour ses conseils tout au long de mon cursus universitaire québécois.

Merci à toutes celles qui m'ont accompagné sur le plan administratif : Lynn, Lise, Flore, Nathalie et Kelthoum. Merci à tous les collègues de l'IUT Sénart-Fontainebleau pour m'avoir accueilli dans leur lieu de travail et plus particulièrement Denis, Pierre, Régis et Patricia.

Je remercie mes amis : Pierre pour avoir été plus qu'un colocataire ; Pierrik pour ta simplicité et ta gentillesse ; Vanessa pour ton grain de folie ; Olivier et Mélanie pour être aussi adorables ; Fabrice et Grégoire pour avoir été là à chacun de mes retours en France ; Jérôme pour qui tu es ; Maxime pour toutes les heures non productives passées ensemble.

Je remercie ma famille : Maman, Papa ainsi qu'Élisabeth et Michel pour leur soutien et leur amour durant ces trois années.

À Alexandre.

Abréviations

ANR : Agence Nationale de la Recherche

AOSD : *Aspect-Oriented Software Development*

AST : arbre de syntaxe abstraite, *Abstract Syntax Tree*

ASTD **:** *Algebraic State Transition Diagram*

BPEL : *Business Process Execution Language*

CIM : modèle indépendant de l'informatique, *Computation Independent Model*

CRSNG : Conseil de Recherches en Sciences Naturelles et en Génie du Canada

CSP : communicating sequential processes

EB^3 **:** *Entity-Based Black-Box*

EB^3**SEC :** EB^3 *Secured*

IS : système d'information, *Information System*

MDA : architecture dirigée par les modèles, *Model Driven Architecture*

MDE : ingénierie dirigée par les modèles, *Model Driven Engineering*

PE : expression de processus, *Process Expression*

PIM : modèle indépendant de la plateforme, *Platform Independent Model*

PSM : modèle spécifique à la plateforme, *Platform Specific Model*

SI : système d'information

SOA : architecture orientée services Web, *Service Oriented Architecture*

UML : *Unified Modeling Language*

Table des matières

Résumé iii

Abstract v

Remerciements vii

Abréviations ix

Table des matières xi

Table des figures xvii

Liste des tableaux xxi

Introduction 1

1 Contexte et état de l'art 9

1.1 Les projets EB^{3}SEC et Selkis . . . 11
1.1.1 Le projet EB^{3}SEC . . . 12
1.1.2 Le projet Selkis . . . 13
1.1.3 Comparaison des projets . . . 15
1.2 Les langages formels de spécification . . . 16
1.2.1 EB^3 . . . 16
1.2.2 ASTD . . . 22
1.2.3 EB^{3}SEC et ASTDsec . . . 24
1.2.4 La méthode B . . . 25

1.2.5 La méthode Event-B 28
1.3 Traductions de langages formels 30
1.3.1 Traduction EB^3 vers B – Attributs 30
1.3.2 Traduction EB^3 vers B – Expressions de processus 31
1.3.3 csp2B 32
1.3.4 UML-B 34
1.3.5 Statecharts vers B 35
1.3.6 Coder une algèbre de processus en Event-B 36
1.3.7 Synthèse 38
1.4 De la spécification à l'implémentation 38
1.4.1 Algèbre de politiques de contrôle d'accès et génération de politique XACML 39
1.4.2 Modélisation Event-B de politiques de sécurité et implémentation par raffinement 40
1.4.3 Implémentation WS-BPEL de politiques de sécurité exprimées avec la notation ASTD 41
1.4.4 Synthèse 42
1.5 Conclusion 43

I Traductions de modèles **45**

2 Traduction d'expressions de processus EB^3 en ASTD **47**
2.1 Introduction 49
2.1.1 Notations pour EB^3 50
2.1.2 Notations pour les ASTD et les automates 51
2.1.3 Notations pour les algorithmes 53
2.2 Traduction naïve 53
2.2.1 Algorithme 53
2.2.2 Exemple 56
2.2.3 Critique 56
2.3 Optimisations 57
2.3.1 Choix et séquences 57

2.3.2 Fermeture de Kleene . . . 61
2.3.3 Gardes . . . 62
2.3.4 Patron PMC . . . 64
2.3.5 Itération de séquences . . . 64
2.4 Conclusion . . . 66
2.4.1 Traduction de EB^{3}SEC vers ASTDsec . . . 66
2.4.2 Bilan . . . 66
2.4.3 Travaux futurs . . . 67

3 Règles de traduction systématique des ASTD en Event-B 69
3.1 Introduction . . . 73
3.2 Event-B Background . . . 74
3.3 ASTD Background . . . 75
3.3.1 ASTD Operators . . . 75
3.3.2 An ASTD Case Study . . . 77
3.3.3 Motivations . . . 79
3.4 Translation . . . 80
3.4.1 Automata . . . 81
3.4.2 Sequence . . . 83
3.4.3 Choice . . . 84
3.4.4 Kleene Closure . . . 84
3.4.5 Synchronization Over a Set of Action Labels . . . 85
3.4.6 Quantified Interleaving . . . 86
3.4.7 Quantified Choice . . . 87
3.4.8 Guard . . . 87
3.4.9 Process call . . . 88
3.5 Animation and Model Checking of the Case Study . . . 89
3.6 Limitations, Conclusion and Future Work . . . 90

4 Traduction d'ASTD vers B 93
4.1 Introduction . . . 97
4.2 Notations . . . 97
4.2.1 Résumé . . . 98

4.2.2 Conventions . . . 98
4.3 Traduction . . . 100
4.3.1 Choix de modélisation . . . 101
4.3.2 Automate . . . 101
4.3.3 Séquence . . . 115
4.3.4 Choix . . . 120
4.3.5 Fermeture de Kleene . . . 127
4.3.6 Synchronisation paramétrée . . . 133
4.3.7 Choix quantifié . . . 138
4.3.8 Synchronisation quantifiée . . . 139
4.3.9 Garde . . . 144
4.3.10 Appel d'ASTD . . . 145
4.4 Principes de la preuve de traduction . . . 146
4.4.1 Cas de base . . . 146
4.4.2 Pas d'induction . . . 148
4.5 Conclusion . . . 149

II Une modélisation B des politiques de contrôle d'accès 151

5 Combinaison d'UML, ASTD et B 153
5.1 Introduction . . . 158
5.2 Context . . . 159
5.2.1 The Selkis Project . . . 159
5.2.2 Illustrative Example . . . 159
5.3 Graphical Models for Access Control . . . 160
5.3.1 SecureUML . . . 161
5.3.2 The ASTD Notation . . . 162
5.4 Translations into B Specifications . . . 165
5.4.1 The B Method . . . 165
5.4.2 A Formal Functional Model . . . 166
5.4.3 A Formal Static Access Control Model . . . 169
5.4.4 Dynamic Access Control Model Translation . . . 172

5.5 Specification of the Access Control Filter 174
5.5.1 Abstract Access Control Filter Specification 174
5.5.2 Refinement of the Access Control Filter 174
5.5.3 Verification and Validation Purpose 177
5.6 Conclusion 178

6 Un méta-modèle B **181**
6.1 Introduction 185
6.1.1 Combining Static and Dynamic rules 186
6.1.2 A Formal Notation for AC rules 186
6.2 A Proposal for AC Modeling using B 187
6.2.1 Combinaison of B Machines 188
6.2.2 The Static Filter 191
6.2.3 The Dynamic Filter 192
6.3 Conclusion 193

7 Une stratégie de raffinement **195**
7.1 Introduction 199
7.2 MM_{imp} : Un méta-modèle d'implémentation de politiques de contrôle d'accès 200
7.3 Raffinement du filtre de contrôle d'accès 204
7.3.1 Implémentation du filtre de contrôle d'accès : *AC_Filter* 204
7.3.2 Implémentation du filtre dynamique : *Dynamic_Filter* 205
7.4 Vers une traduction BPEL 210
7.5 Conclusion 211

Conclusion **213**

Annexes 221

A Étude de cas : Traduction d'un ASTD en Event-B **223**
A.1 Contexte commun aux ASTD 224
A.2 Contexte 226
A.3 Machine 227

B Étude de cas : Traduction d'un ASTD en B **235**
B.1 SI de gestion d'une bibliothèques 235
B.1.1 Traduction 236
B.1.2 Statistiques 241

C Règles d'inférence des ASTD **243**

Bibliographie **249**

Table des figures

1 Plan de travail de la thèse 7

1.1 Unités de travail du projet EB^{3}SEC 13
1.2 Unités de travail du projet Selkis 15
1.3 Une expression de processus EB^3 spécifiant le SI gérant une bibliothèque . . 20
1.4 Exemple d'un ASTD 24
1.5 Les actions élémentaires EB^3 et EB^{3}SEC 25
1.6 Exemple de règle de sécurité EB^{3}SEC 25
1.7 LTS correspondant au processus CSP décrivant une machine à thé et à café 33

2.1 Traduction d'expressions de processus EB^{3}SEC vers ASTD 49
2.2 Expression de processus **member** 50
2.3 AST de l'expression de processus **member** 50
2.4 Exemple d'un ASTD de type automate 52
2.5 ASTD résultant de la traduction naïve de l'expression de processus **member** 56
2.6 En haut, la version non optimisée de la traduction de $\sigma(x)^*$. En bas, la version optimisée 63
2.7 En haut, la version non optimisée de la traduction de $g \Rightarrow \sigma(x)$. En bas, la version optimisée 64
2.8 Traduction du patron PMC 65
2.9 Traduction de l'itération de séquences 65
2.10 Comparaison d'actions élémentaires EB^3 et EB^{3}SEC 66
2.11 ASTD résultant de la traduction optimisée de l'expression de processus **member** 67

3.1 Traduction de spécification ASTD en Event-B 71
3.2 An ASTD specification describing a complaint management system 78
3.3 The architecture resulting from the translation process 80

4.1 Traduction de spécification ASTD en B et preuve de la traduction 95
4.2 Un exemple d'ASTD de type automate 105
4.3 Un ASTD de type automate avec tous les types de transitions possibles . . . 109
4.4 Un exemple d'ASTD de type séquence 116
4.5 Un exemple d'ASTD de type choix 122
4.6 Un exemple d'ASTD de type séquence contenant deux ASTD de type fermeture de Kleene 129
4.7 Un ASTD de type synchronisation incluant deux ASTD de type automate . . 135
4.8 Un exemple d'ASTD de type synchronisation quantifiée 142
4.9 Preuve de la traduction en utilisant la fonction η 146

5.1 Modèle formel B de politiques de contrôle d'accès 155
5.2 Functional model 160
5.3 Access control rules for medical records 161
5.4 ASTD model of rule R4: If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read access to his medical record 163
5.5 ASTD model of rule R5: Any modification of a patient's medical record must be eventually validated. Several modifications can be validated by a single validation 164
5.6 Basic B structures related to medical records 167
5.7 Operation *MedicalRecord__GetData* 168
5.8 Operation *MedicalRecord__SetData* 168
5.9 Operation *secure_MedicalRecord__SetData* 171
5.10 Operation *secure_MedicalRecord__GetData* 172
5.11 Operation *Dynamic_MedicalRecord__GetData* 173
5.12 Abstract operations *Rollback* and *Filter_MedicalRecord__GetData* 175
5.13 Overall view of the access control filter 176
5.14 Refinement of *Filter_MedicalRecord__GetData* 180

6.1 Méta-modèle B_{sec} . . . 183
6.2 Main part of the metamodel for AC modeling in B . . . 189
6.3 Details of the static machine of the metamodel for AC modeling in B . . . 191
6.4 Details of the dynamic machine of the metamodel for AC modeling in B . . . 192

7.1 Raffinement guidé vers l'implémentation . . . 197
7.2 Diagramme de séquence : exécution de l'action $E(bp)$ par l'utilisateur . . . 201
7.3 Diagramme de séquence : prise de décision en consultant les filtres . . . 202
7.4 Rapprochement du méta-modèle MM_{imp} et du méta-modèle MMB_{sec} : partie dynamique . . . 203
7.5 Rapprochement des acteurs du diagramme de séquence de Embe Jiague *et al.* et du méta-modèle MMB_{sec} . . . 203
7.6 Exemple d'ASTD de type automate . . . 205
7.7 Un ASTD de type synchronisation quantifiée . . . 208
c.1 Contributions de la thèse . . . 216

A.1 ASTD décrivant un système de gestion de plaintes de clients . . . 224

Liste des tableaux

3.1 Event-B representation of ASTD states 81
3.2 Automata ASTD to Event-B translation rules 83
3.3 Sequence ASTD to Event-B translation rule 84
3.4 Choice ASTD to Event-B translation rule 85
3.5 Kleene closure ASTD to Event-B translation rule 85
3.6 Synchronization ASTD to Event-B translation rule 86
3.7 Quantified interleaving ASTD to Event-B translation rule 87
3.8 Quantified choice ASTD to Event-B translation rule 88
3.9 Guard ASTD to Event-B translation rule 88

4.1 Résumé des notations relatives aux ASTD 98
4.2 Résumé des notations relatives à B 98
4.3 Prédicats et substitutions pour chaque type de transitions 106
4.4 Lien entre les substitutions et les règles d’inférence 108
4.5 Lien entre les prédicats et les règles d’inférence 108

7.1 Implémentation des ensembles spécifiques aux types d’ASTD 206

Introduction

Motivations

Un système d'information (SI), comme de nombreux autres éléments logiciels ou matériels du monde informatique, comporte des exigences de sécurité dont les sources sont multiples.

Les systèmes d'information

Les SI sont des logiciels permettant l'acquisition, le stockage, la structuration, la gestion, le déplacement, le contrôle, l'affichage et l'échange de données. Un exemple de SI classique est une application de gestion des commandes et des clients d'une entreprise. Dans le monde de l'entreprise, un SI constitue un élément extrêmement sensible qui peut influencer le bon fonctionnement et les résultats d'une organisation selon la gestion qui en est faite. De par leur nature et leurs fonctions, les SI ont accès en lecture et en écriture à des données capitales pour le fonctionnement de l'entreprise comme des bases de données de clients, de fournisseurs ou ses données comptables. Un SI correctement développé et adapté à une entreprise peut être source de nombreux avantages. Inversement, un SI mal adapté en taille ou en fonctionnalités peut conduire à un ralentissement des processus métier et à une diminution de la fiabilité de l'entreprise. Un processus métier est une succession de tâches reliées par un objectif commun, comme par exemple la gestion d'une commande d'un client. En cas de ralentissement de ces processus, les conséquences sur une entreprise sont multiples et négatives. Du fait de leur importance, l'entreprise doit donc pouvoir compter sur les SI qu'elle utilise.

Le contrôle d'accès des SI

Du fait des données manipulées, la question de la sécurité des SI est capitale. Plusieurs domaines de l'activité économique et sociale sont maintenant sujets à des lois et des normes très strictes au niveau de la sécurité et de la confidentialité des données. Par exemple, nous pouvons citer la loi Sarbane-Oxley [7] aux États-Unis et la loi PIPEDA au Canada. En effet, il est difficilement envisageable que certaines informations d'une entreprise soient accessibles en lecture ou en écriture par des entités extérieures à l'entreprise. De même, certains employés ne devraient pas avoir accès à certaines données de leur entreprise. Il est dangereux de donner un accès en écriture à tous les employés sur les données relatives aux salaires. Plusieurs niveaux de sécurité sont donc nécessaires au sein des SI. L'assurance que les données ne sont pas accessibles depuis l'extérieur du SI se construit en utilisant des technologies comme les pare-feu logiciels ou physiques. Ces techniques conduisent à la création d'une zone dite démilitarisée, ou zone de confiance, au sein de laquelle les communications sont réputées sécurisées. La propriété que les données consultées le sont par une personne autorisée et dans des conditions correctes peut être assurée par des processus d'authentification et de contrôle d'accès. L'authentification doit vérifier qu'une personne est bien celle qu'elle prétend être. Le contrôle d'accès doit garantir qu'une personne utilise des fonctions qu'elle est habilitée à manipuler. Enfin d'autres règles de contrôle d'accès aux données, comme les droits de lecture ou d'écriture, la durée de validité des données, la révocation de privilèges, peuvent également garantir la sécurité des données manipulées par un SI.

D'autres éléments comme la sûreté, la fiabilité, la redondance, la qualité de service peuvent définir d'autres exigences de sécurité de la part des entreprises envers leurs SI. Ces éléments ne seront pas détaillés ici car ils ne rentrent pas dans le cadre de cette thèse.

Dans le domaine bancaire, le contrôle d'accès est un des services vendus aux clients. En théorie, il n'est pas concevable que les données d'un client soient consultables par tous, ou que le compte d'un client lui soit accessible en écriture directement. Il n'est également pas acceptable que des transactions bancaires soient refusées lors d'opérations de maintenance des SI. En pratique, sécuriser les SI dans le milieu bancaire est d'une complexité élevée et d'un coût non négligeable. Du fait de la présence de multiples architectures, systèmes, langages, base de données et d'acquisitions, restructurations, changements de technologies au cours du cycle de vie d'un SI, il n'est en aucun cas trivial de déployer, d'administrer et

de maintenir une politique de sécurité. Cependant, ces tâches sont indispensables au fonctionnement d'une institution bancaire. Les enjeux de la sécurité dans le domaine bancaire sont multiples. La sécurité dans ce domaine conditionne le fonctionnement non seulement de l'économie mais également de tous les systèmes politiques et sociaux. De plus, dans le contexte de la crise financière internationale, des dérives similaires à celle de l'affaire Jérôme Kerviel [16] en France ne sont pas acceptables, menaçant les institutions, leurs clients et les autres entreprises dans lesquelles elles investissent.

Les hôpitaux abandonnent progressivement les dossiers médicaux au format papier au profit de dossiers médicaux électroniques. Cette modification du système rend les données contenues dans ces dossiers plus simples d'accès pour les personnels hospitaliers et interopérables d'une institution à une autre. Cependant, les dossiers médicaux contiennent des données sensibles souvent protégées par les lois nationales, que ce soit sous leur forme papier ou électronique. La France a par exemple défini par voie légale des moyens à mettre en œuvre pour la protection des données médicales.

FRANCE - CODE DE LA SANTÉ PUBLIQUE [54]

> Toute personne prise en charge par un professionnel, un établissement, un réseau de santé ou tout autre organisme participant à la prévention et aux soins a droit au respect de sa vie privée et du secret des informations la concernant.
>
> ...
>
> Afin de garantir la confidentialité des informations médicales mentionnées aux alinéas précédents, leur conservation sur support informatique, comme leur transmission par voie électronique entre professionnels, sont soumises à des règles définies par décret en Conseil d'Etat.

De même, le Canada, par le biais d'une initiative fédérale, propose une architecture sécurisée [9] pour le développement d'un Dossier de Santé Électronique (DSE). Ces recommandations définissent l'implémentation des dossiers médicaux électroniques et ont été transposées dans le droit provincial. Du fait de ces lois et initiatives, il devient nécessaire de contrôler l'accès aux données disponibles dans les SI utilisés par les établissements et professionnels de la santé. Cependant, ces politiques de sécurité doivent prendre en compte

les situations d'urgences. Un dossier médical ne doit pas pouvoir être consulté par des personnes extérieures à un établissement et non autorisées, dérogation faite des cas d'urgence médicale où un transfert d'autorisation peut être nécessaire. De nombreux autres critères peuvent être utilisés pour autoriser ou non l'accès à certaines données des SI médicaux.

Règles et politiques de contrôle d'accès des SI

Les règles de contrôle d'accès définissent qui, au sein d'un SI, peut exécuter quelles actions et sous quelles contraintes. Il existe différentes catégories de règles de contrôle d'accès comme l'autorisation, l'interdiction, l'obligation et la séparation des devoirs. L'autorisation donne le droit d'exécution d'une action à un utilisateur selon certaines conditions, alors que l'interdiction retire ce droit. L'obligation introduit une notion de sanction en cas de non-respect de la règle. La séparation des devoirs permet d'introduire des séquences d'actions qui devront être exécutées par des personnes différentes. On définit la politique de contrôle d'accès d'un SI comme l'ensemble des règles de contrôle d'accès qui s'y applique. Lors de l'étude de l'architecture des SI, il apparaît généralement que les éléments concernant le contrôle d'accès sont intégrés en différents points du code et au sein des éléments fonctionnels, c'est-à-dire à l'intérieur même des procédures implémentant les fonctionnalités du système. Si cette approche peut sembler plus évidente lors des phases de codage, la maintenance des politiques de contrôle d'accès n'en est pas simplifiée. En cas de changement de politiques, il faut localiser l'élément à modifier dans le code et s'assurer que cet élément n'est pas réutilisé ailleurs. Il n'existe généralement pas de zone dédiée à la description et à l'implémentation des politiques de contrôle d'accès, ces éléments sont souvent littéralement dispersés dans le code.

Vers un processus formel d'intégration de politiques de contrôle d'accès dans les SI

Cette thèse s'intègre dans le cadre de plusieurs initiatives de recherche. Notre approche propose de générer formellement une implémentation de filtre de contrôle d'accès à partir d'une spécification formelle de politique de contrôle d'accès. Deux méthodes de spécifications sont prises en charge. La politique de contrôle d'accès peut être exprimée en utilisant :

(i) le langage EB^3SEC [53], qui permet d'exprimer sous forme d'expressions de processus les règles de contrôle d'accès ; (ii) une approche graphique combinant SecureUML [57] et ASTD [32] pour spécifier les règles selon leur type. Notre approche permet ensuite de traduire la spécification en utilisant des règles de traduction prouvées qui garantissent ainsi une conservation de la sémantique de la politique de contrôle d'accès. Le modèle ainsi traduit est exprimé en utilisant la méthode B [3] ou Event-B. Ces langages permettent d'effectuer des vérifications formelles et de valider la politique par le biais de son animation. La méthode B définit également des étapes de raffinement qui permettent d'obtenir un modèle implémentable du modèle initial. Nous avons donc défini une stratégie de raffinement qui conserve les notions importantes de la politique de contrôle d'accès dans le modèle final. Ce modèle final respecte les contraintes de l'architecture du filtre de contrôle d'accès définie en collaboration avec nos partenaires industriels.

Contributions et plan de la thèse

À des fins d'illustration, la figure 1 détaille le cheminement entre les différentes étapes de transformations des modèles. Les principales contributions de cette thèse sont les suivantes :

1. **Un état de l'art des traductions entre langages formels.** Dans les premières étapes de notre approche, des traductions entre EB^3 et ASTD dans un premier temps, puis d'ASTD vers B dans un second temps sont nécessaires. Plusieurs travaux proposent des traductions outillées ou non entre langages formels. L'étude de ces travaux nous a permis de prendre en compte les spécificités des langages employés dans nos traductions.
2. **Règles de traduction de EB^3 vers ASTD.** Cette traduction nous permet de faire le lien entre les deux types de spécifications initiales possibles dans notre approche. Du fait de la proximité entre EB^3SEC et EB^3, la traduction a été développée pour prendre en compte les deux langages. Cette contribution correspond à la flèche 2 de la figure 1. Les spécifications EB^3SEC ont été développées à partir d'une phase d'analyse des besoins qui n'entre pas dans le cadre de cette thèse.

3. **Règles de traduction de ASTD vers Event-B.** Dans un premier temps, nous avons souhaité traduire les spécifications ASTD en Event-B du fait de la proximité de la sémantique d'exécution des ASTD et des machines Event-B. Cependant cette traduction s'est avérée moins pertinente que prévue et nous avons abandonné l'idée d'utiliser Event-B dans nos modèles. Cette contribution correspond à la flèche 3 de la figure 1.
4. **Règles de traduction de ASTD vers B et leur preuve.** En se basant sur la traduction vers Event-B, des règles de traduction vers B on été développées. Elles sont décrites de manière plus formelle et ont été prouvées, de sorte que des propriétés vérifiées sur une spécification B le soient sur la spécification ASTD d'origine. Cette contribution correspond à la flèche 4 de la figure 1.
5. **Une architecture formelle exprimée en B pour l'application de politiques de contrôle d'accès sur les SI.** Afin de combiner différents aspects de la politique de contrôle d'accès avec la description formelle de l'aspect fonctionnel de l'approche, nous avons défini un méta-modèle B nommé B_{sec} qui intègre les notions des politiques de contrôle d'accès dans la spécification B d'un filtre. Cette contribution correspond aux cases 5 et 6 de la figure 1. Les spécifications SecureUML/B et ASTD utilisées à cette étape découlent d'une phase d'analyse des besoins qui n'entre pas dans le cadre de cette thèse.
6. **Une stratégie de raffinement B pour obtenir une implémentation d'un filtre de contrôle d'accès.** Une fois le modèle respectant le méta-modèle B_{sec} établi, une stratégie de raffinement est suivie pour obtenir une implémentation B respectant l'architecture désirée pour le filtre de contrôle d'accès. Cette contribution correspond à la flèche 7 de la figure 1.

Cette thèse est composée de sept chapitres. Nous étudions le contexte et présentons l'état de l'art dans le chapitre 1. La thèse se décompose ensuite en deux parties. Dans la première partie nous présentons les différentes approches de traductions. Le chapitre 2 présente la traduction de EB^3SEC et EB^3 vers ASTD. Puis le chapitre 3 détaille la traduction des ASTD en Event-B. Enfin le chapitre 4 introduit les règles de traduction des ASTD vers B et prouve que cette traduction conserve la sémantique des ASTD. La seconde partie introduit les notations pour l'expression de la politique de contrôle d'accès en B et son implémentation. Le chapitre 5 présente l'architecture formelle d'un filtre de contrôle d'accès exprimé

en B. Puis le chapitre 6 introduit un méta-modèle basé sur B pour la définition d'un filtre de contrôle d'accès. Enfin le chapitre 7 détaille le processus de raffinement d'une spécification de politique de contrôle d'accès vers une implémentation.

La figure 1 illustre ce cheminement. Elle est présentée en début de chaque chapitre pour indiquer les points développés et les limites des travaux présentés.

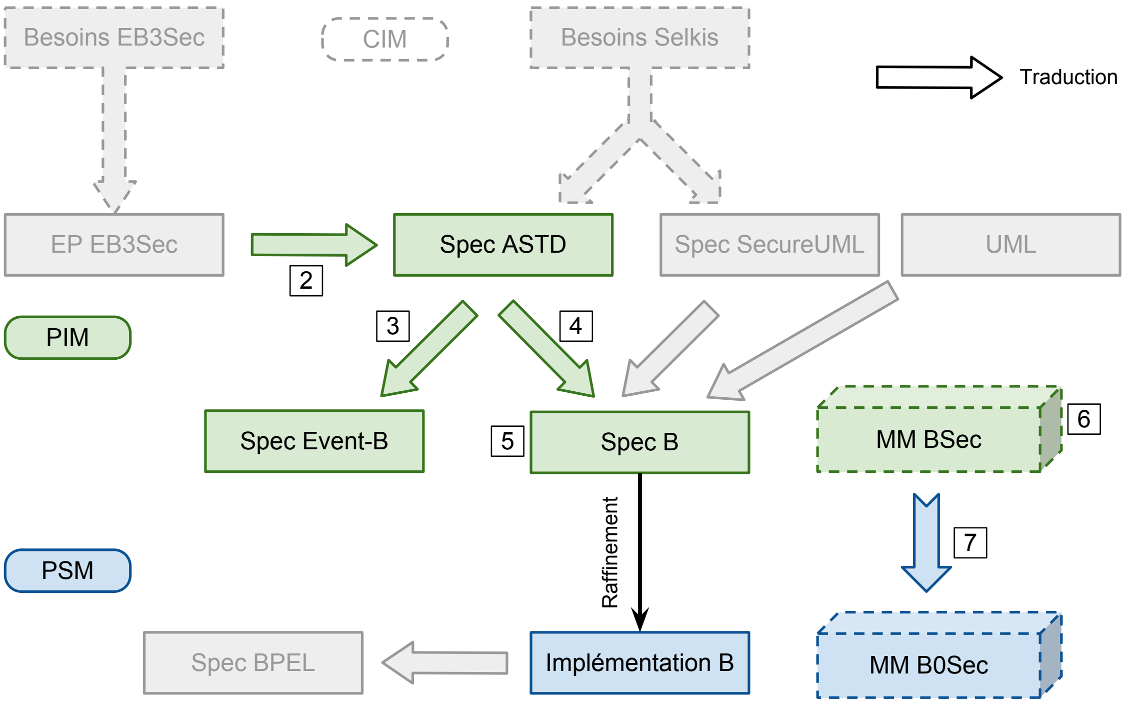
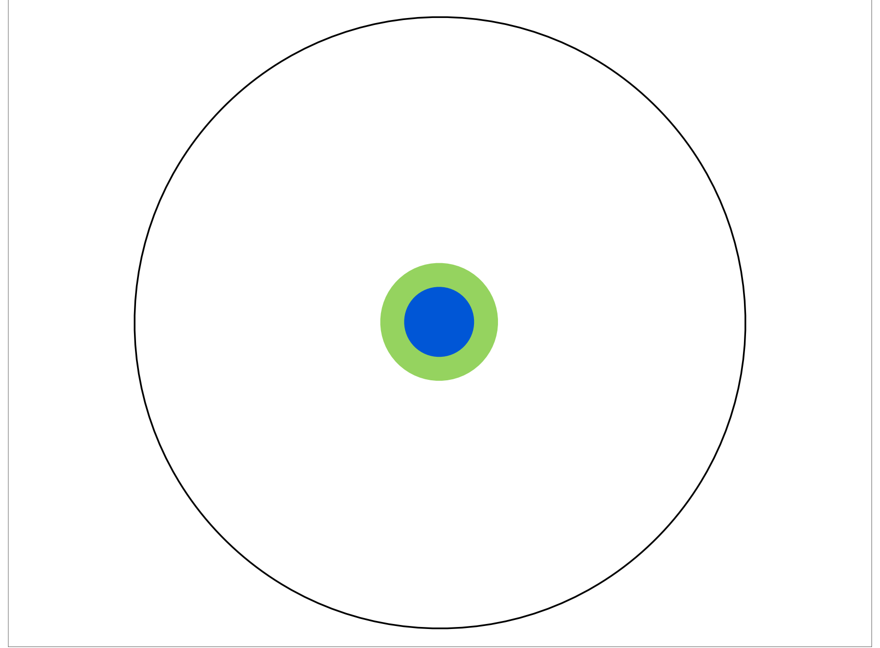


FIGURE 1 – Plan de travail de la thèse

Chapitre 1

Contexte et état de l'art



[65] Ph.D. étape 1 – *En guise d'exergue à chacun des chapitres de cette thèse, je vous propose une petite histoire sur la représentation d'une thèse par rapport à l'ensemble des connaissances humaines. Cette série d'images a été créée par Matt Might, chargé de cours à l'Université de l'Utah.* Imaginons un cercle qui représente l'ensemble des connaissances humaines. Le disque intérieur représente les connaissances acquises à l'école primaire. L'anneau concentrique suivant représente les connaissances acquises au collège.

Les sections suivantes détaillent le contexte de cette thèse et décrivent l'état de l'art dans le domaine des traductions de langages formels et l'implémentation de politiques de contrôle d'accès. La section 1.1 introduit tout d'abord les projets de recherche EB^{3}SEC et Selkis. Nous présentons les langages formels utilisés dans notre approche dans la section 1.2. Nous décrivons ensuite les approches de traduction de langages formels utilisant des structures similaires à EB^{3}, EB^{3}SEC et ASTD dans la section 1.3. Enfin, nous étudions les travaux sur l'implémentation de règles de contrôle d'accès dans la section 1.4. Une conclusion récapitulant les éléments importants de l'état de l'art est présentée en section 1.5.

1.1 Les projets EB^{3}SEC et Selkis

Les projets EB^{3}SEC [1] et Selkis [2], initiatives de recherche respectivement canadienne, via le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG) et l'Université de Sherbrooke, et française via l'Agence Nationale de la Recherche (ANR), visent à proposer une méthode formelle d'expression de politiques de contrôle d'accès des SI et un processus conduisant à l'implémentation d'un noyau gérant le contrôle d'accès au sein d'un SI à partir de sa spécification formelle. La méthodologie des projets EB^{3}SEC et Selkis permet de définir une politique de contrôle d'accès formelle en suivant une approche ingénierie dirigée par les modèles, *Model Driven Engineering* (MDE). L'approche MDE est une approche permettant, par le biais d'outils et de langages formels ou non, de créer et de transformer des modèles. Ces modèles représentent de façon plus ou moins abstraite le système développé. Dans le cadre de nos projets de recherche, un modèle formel de la politique de contrôle d'accès est dérivé en d'autres modèles jusqu'à obtenir une modèle « utilisable », c'est-à-dire l'implémentation de la politique de contrôle d'accès dans le SI.

L'approche MDE se compose de trois étapes de modélisation successives. Ces étapes peuvent se décomposer en d'autres sous-étapes. Elles obéissent cependant à des règles précises. La première étape est la conception d'un modèle indépendant de l'informatique, *Computation Independent Model* (CIM), qui doit être indépendant de toute représentation et technique informatique. Un exemple de modélisation CIM est la description d'une politique de contrôle d'accès avec un langage naturel comme « Cet employé n'a pas le droit

1. http://pages.usherbrooke.ca/eb3sec/
2. http://lacl.fr/selkis/

de consulter ce document ». La seconde modélisation produit un modèle indépendant de la plateforme, *Platform Independent Model* (PIM) qui ne doit pas contenir d'éléments spécifiques à la plateforme sur laquelle est déployé le logiciel. Un exemple de modèle PIM est l'expression de la règle de sécurité décrite précédemment en utilisant un langage de modélisation comme *Unified Modeling Language* (UML), sans pour autant implémenter la règle. Enfin, le modèle spécifique à la plateforme, *Platform Specific Model* (PSM) apporte les détails de l'implémentation spécifiques à l'architecture. Un exemple d'implémentation PSM est le code Java découlant de la règle de sécurité décrite précédemment.

Dans le cadre du projet EB^{3}SEC, l'approche qui a été retenue est la spécification d'un PIM de politiques de contrôle d'accès exprimées en utilisant une algèbre de processus associée à une sémantique formelle [52] similaire à EB^3 [29, 38]. Pour le projet Selkis, la spécification de la politique de sécurité se fait en utilisant des notations graphiques comme SecureUML [57] et les ASTD [32, 33]. Les modèles PSM attendus pour le filtre de contrôle d'accès, constituant l'implémentation d'un politique de contrôle d'accès au sein du SI, sont similaires dans les deux approches.

Cette thèse s'inscrit dans le cadre des unités de travail *WP2* du projet EB^{3}SEC présenté dans la figure 1.1 et *WP3*/*WP5* du projet Selkis présenté dans la figure 1.2.

1.1.1 Le projet EB^{3}SEC

Le projet EB^{3}SEC s'adresse à un aspect de la sécurité dit fonctionnel. Cet aspect comprend les règles de sécurité qui doivent être spécifiées au niveau des processus d'affaires de l'entreprise. Par exemple, dans un système de gestion des dossiers patients d'un hôpital, les données d'un patient ne peuvent être consultées que par le médecin traitant. En cas d'urgence, un spécialiste peut avoir un accès restreint aux données particulières pour une période de temps limitée. Il y a plusieurs niveaux de spécification des règles de sécurité : les attributs d'une entité, les services atomiques et les processus d'affaires (composés de services atomiques).

Le solution apportée par le projet doit proposer :

- Une méthode de spécification des politiques de contrôle d'accès pour chacun des niveaux de spécification.

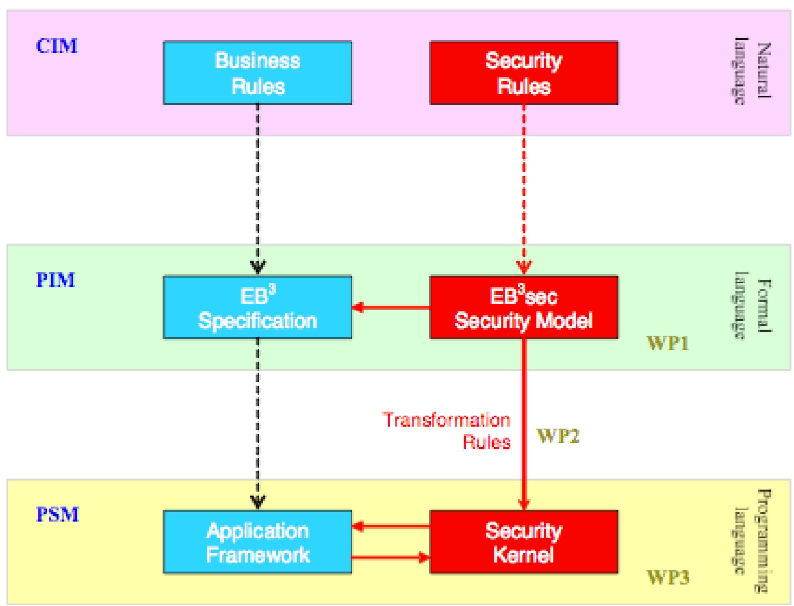


FIGURE 1.1 – Détails des unités de travail (*Work Packages*) du projet EB^{3}SEC

– Des algorithmes de synthèse automatique de filtre de contrôle d'accès qui traduisent une politique de sécurité en un programme exécutable, dans le contexte d'une architecture orientée services Web, *Service Oriented Architecture* (SOA).

La spécification d'une politique de sécurité fonctionnelle doit être distincte de la spécification du comportement des services, afin de faciliter la maintenance en temps réel des politiques de sécurité fonctionnelle d'une application sans avoir à modifier l'application. Notre approche est fondée sur les méthodes formelles de développement, qui permettent de vérifier la cohérence et la validité d'une politique.

Au sein du projet, cette thèse se concentre sur la transformation d'une politique de contrôle d'accès exprimée avec le langage EB^{3}SEC en un noyau de sécurité, que nous appelons filtre de contrôle d'accès, indépendant du système.

1.1.2 Le projet Selkis

Le projet Selkis a pour objectif de développer une méthode d'analyse et conception de SI sécurisés qui aborde les aspects fonctionnels et sécuritaires dès les premiers niveaux

d'abstraction du développement et combine les mécanismes sécuritaires disponibles au niveau implantation dans les logiciels de gestion de l'information. La méthode proposée doit prendre en compte les propriétés de sécurité suivantes : disponibilité, intégrité, confidentialité et traçabilité qui sont cruciales dans ce type de SI. La méthode est fondée sur une approche MDE qui permet de décrire un SI à trois niveaux d'abstraction. Le premier niveau consiste à créer un modèle du système global CIM, indépendant de tout aspect informatique, en considérant différents types de besoins, dont les besoins de sécurité, recueillis à partir du système à construire et de son environnement. Le méthode Kaos [15, 17] a été choisie pour exprimer une partie de ces besoins [5]. Le second niveau décrit un modèle du système à construire indépendant de toute plateforme d'implémentation (PIM). Les langages choisis pour cette modélisation sont SecureUML et ASTD. Une traduction en B depuis SecureUML et ASTD a également été développée. Le dernier niveau décrit un modèle qui caractérise l'architecture d'implémentation retenue pour le SI (PSM). Ces implémentations doivent se faire au sein d'une architecture SOA. Le langage *Business Process Execution Language* (BPEL) a été choisi pour une de ces implémentations [23].

L'objectif du projet Selkis est triple : (i) spécifier de manière séparée et abstraite les besoins fonctionnels et de sécurité ; (ii) implémenter des mécanismes de sécurité indépendamment du code de l'application ; (iii) définir de manière explicite les liens entre l'implémentation et la spécification.

Au sein du projet Selkis, cette thèse se concentre sur les différents modèles de la politique de contrôle d'accès et leur transformation : (i) Au niveau CIM, les règles de la politique de contrôle d'accès sont définies en analysant les besoins de sécurité de l'application. Puis un modèle formel de la politique de contrôle d'accès est défini au niveau PIM à partir de l'analyse des besoins de sécurité. Ces étapes n'entrent pas dans le cadre de cette thèse. (ii) Au niveau PIM, un filtre abstrait de contrôle d'accès intègre les différents aspects de la politique de contrôle d'accès avec l'aspect fonctionnel du système. Ce filtre interroge la politique de contrôle d'accès pour déterminer si l'exécution d'une action est autorisée ou non. Selon le résultat, le filtre peut déclencher ou non l'exécution de l'action au niveau fonctionnel. L'ensemble constitué par le filtre, la politique de contrôle d'accès et l'aspect fonctionnel du système peut être vérifié et prouvé afin de garantir des propriétés de sécurité sur l'ensemble du système. (iii) Au niveau PSM, le filtre agit comme une composant interceptant les requêtes de l'utilisateur et communiquant avec d'autres modules constituant la

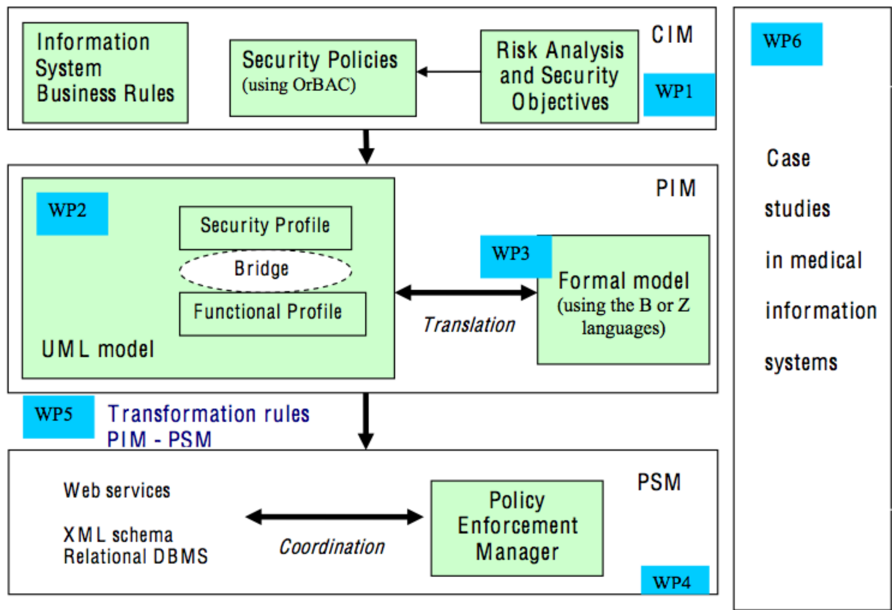


FIGURE 1.2 – Détails des unités de travail (*Work Packages*) du projet Selkis

politique de contrôle d'accès, les données du système et la partie fonctionnelle du système. Le filtre doit être raffiné à partir de sa définition formelle selon une approche précise afin qu'il conserve les propriétés de sécurité prouvées ou vérifiées au niveau PIM.

1.1.3 Comparaison des projets

En rapprochant la figure 1.1 et la figure 1.2 décrivant les projets Selkis et EB^{3}SEC, on constate que le principe d'un langage formel décrivant une politique de sécurité est similaire dans les deux approches. De même, le filtre de contrôle d'accès constitue une implémentation similaire nommée *Security Kernel* ou *Policy Enforcement Manager*. Enfin, les deux projets souhaitent réaliser la traduction de modèles en implémentations par le biais de règles de transformation ou de raffinement.

Dans le cadre du projet EB^{3}SEC, la spécification de la politique de sécurité prend la forme d'une expression de processus. Pour le projet Selkis, la spécification de la politique de sécurité est décrite en utilisant des notations graphiques. Cependant, le filtre de contrôle d'accès est similaire à celui du projet EB^{3}SEC.

L'article [21] présente une méthode de conception des SI basée sur la séparation des spécifications et des implémentations des aspects fonctionnels et contrôle d'accès. Cette séparation entre les aspects fonctionnels et de sécurité est au centre des projets Selkis et EB^{3}SEC. L'article introduit également des propriétés à vérifier sur chacun des aspects mais également sur l'ensemble de la spécification comme les propriétés dites de vivacité (*liveness*).

1.2 Les langages formels de spécification

Dans les sections qui suivent, nous détaillons les langages utilisés dans le cadre de cette thèse.

1.2.1 EB^3

Le projet APIS (*automatic production of information systems*) [31, 30] a pour objectif de générer automatiquement un SI décrit par un PIM formel. Issue du projet APIS, EB^3 [29, 38] (*Entity-Based Black Box*) est une méthode de développement de SI qui permet de spécifier différentes parties d'un système et de générer une implémentation à l'aide de différents outils.

L'algèbre de processus EB^3

L'algèbre de processus de la méthode EB^3 permet d'écrire des expressions de processus qui servent d'entrées pour la génération automatique d'un SI. Ces spécifications se basent sur des notations orientées évènement comme les langages de spécification CSP [44] et CCS [66].

Dans le cadre de cette thèse, l'algèbre de processus, et plus particulièrement sa sémantique opérationnelle occupent une place importante. Nous détaillons ici les différents opérateurs. Les expressions de processus sont créées à l'aide d'expressions de processus élémentaires et des opérateurs de l'algèbre de processus EB^3.

Expression élémentaire Une expression de processus élémentaire peut être λ, une action interne au système, semblable à ε dans la théorie des automates ; ou $\mathsf{action}(x_1, \ldots, x_n)$

est une action élémentaire du système avec ses n paramètres x_1 à x_n. Les expressions de processus peuvent être représentées sous forme d'arbres de syntaxe abstraite. Les feuilles correspondent alors à des expression élémentaires.

Séquence La séquence de deux expressions de processus est notée $E_1 \mathbin{.} E_2$. La séquence permet d'exprimer le fait que le processus E_1 doit terminer son exécution avant que l'exécution de E_2 ne débute. L'opérateur de séquence EB[3] est semblable à l'opérateur . de la théorie des expressions rationnelles.

Choix Le choix entre deux expressions de processus est noté $E_1 \mid E_2$. Le choix permet d'exprimer le fait que seul l'un des deux processus E_1 ou E_2 est exécuté. La détermination du processus exécuté est laissée à l'utilisateur. L'opérateur de choix EB[3] est semblable à l'opérateur $\mid$ ou $+$ de la théorie des expressions rationnelles.

Fermeture de Kleene La fermeture de Kleene d'une expression de processus est notée E^*. Elle permet d'exprimer le fait que l'exécution de E peut avoir lieu zéro, une ou plusieurs fois. Le nombre d'itérations est laissé au choix de l'utilisateur. La fermeture de Kleene est similaire à l'opérateur de même nom de la théorie des expressions rationnelles.

Synchronisation La synchronisation sur un ensemble d'étiquettes d'actions Δ est notée $E_1|[\Delta]|E_2$. Elle exprime le fait que les expressions E_1 et E_2 s'exécutent de manière entrelacée sauf lors de l'exécution d'une action dont l'étiquette appartient à Δ où les deux expressions doivent se synchroniser. Dans ce cas, les deux expressions doivent pouvoir exécuter la même action en même temps. Les synchronisations peuvent engendrer des situations de blocage de l'expression de processus, aucune action ne peut alors être exécutée.

Choix quantifié La version quantifiée du choix d'une expression de processus est notée $\mid x : \mathrm{T} : E$. Elle exprime le fait qu'une valeur $v \in \mathrm{T}$ doit être choisie et que les occurrences de la variable x dans E devront être remplacées par v. Un choix quantifié est donc équivalent au choix entre $card(\mathrm{T})$ processus différents.

Synchronisation quantifiée La version quantifiée de la synchronisation d'une expression de processus est notée $|[\Delta]|x : \mathrm{T} : E$. Elle exprime le fait que $card(\mathrm{T})$ processus entrelacés sont lancés avec une valeur $v \in \mathrm{T}$ différente choisie pour remplacer la variable x dans E. Les expressions doivent cependant se synchroniser pour l'exécution des actions dont l'étiquette appartient à Δ.

Garde La garde d'une expression de processus par un prédicat p est notée $p \Longrightarrow E$. Elle permet d'empêcher le début de l'exécution de E tant que le prédicat p n'est pas vrai. Le prédicat de la garde est exprimé en utilisant la logique du premier ordre et peut faire appel à des variables quantifiées, des attributs du système ou des constantes.

Appel de processus L'appel de processus permet de faire référence à une expression de processus en lui passant éventuellement des paramètres.

Synchronisation faible Suite à des cas d'étude dans le domaine bancaire, certaines formes d'expressions sont apparues de manière récurrente. Pour simplifier ces expressions de processus, un nouvel opérateur à été défini. Il se nomme *synchronisation faible* et s'applique sur deux expressions de processus. La synchronisation faible de deux expressions de processus est notée $E_1 \pitchfork E_2$. Soit σ une action. La sémantique de l'opérateur $\pitchfork$ est définie par les trois règles d'inférence opérationnelles suivantes :

$$\frac{E_1 \xrightarrow{(\sigma,\Gamma)} E_1' \quad E_2 \dagger^{(\sigma,\Gamma)}}{E_1 \pitchfork E_2 \xrightarrow{(\sigma,\Gamma)} E_1' \pitchfork E_2}$$

Cette règle d'inférence décrit le résultat de l'exécution de σ dans l'environnement Γ par $E_1 \pitchfork E_2$ sachant que E_1 peut exécuter σ dans l'environnement Γ et que E_2 ne peut pas exécuter σ dans l'environnement Γ.

$$\frac{E_1 \dagger^{(\sigma,\Gamma)} \quad E_2 \xrightarrow{(\sigma,\Gamma)} E_2'}{E_1 \pitchfork E_2 \xrightarrow{(\sigma,\Gamma)} E_1 \pitchfork E_2'}$$

Cette règle d'inférence décrit le résultat de l'exécution de σ dans l'environnement Γ par $E_1 \pitchfork E_2$ sachant que E_1 ne peut pas exécuter σ dans l'environnement Γ et que E_2 peut

exécuter σ dans l'environnement Γ.

$$\frac{E_1 \xrightarrow{(\sigma,\Gamma)} E_1' \quad E_2 \xrightarrow{(\sigma,\Gamma)} E_2'}{E_1 \pitchfork E_2 \xrightarrow{(\sigma,\Gamma)} E_1' \pitchfork E_2'}$$

Cette règle d'inférence décrit le résultat de l'exécution de σ dans l'environnement Γ par $E_1 \pitchfork E_2$ sachant que E_1 et E_2 peuvent exécuter σ dans l'environnement Γ. Intuitivement, cet opérateur permet d'exécuter l'action σ sur le maximum d'opérandes qui peuvent l'exécuter.

Opérateurs supplémentaires Du fait de la fréquence de leur apparition dans les expressions de processus, les opérateurs suivant sont également définis : l'entrelacement noté $E_1 \,|||\, E_2$ et défini par $E_1|[\varnothing]|E_2$; l'exécution parallèle notée $E_1 \parallel E_2$ et définie par $E_1|[\Delta]|E_2$, où Δ est l'ensemble des étiquettes des actions communes à E_1 et E_2. Enfin, consécutivement à l'exécution d'actions dans une expression de processus, peuvent apparaître des environnements notés $\Gamma = ([x_1 := y_1, \ldots, x_n := y_n])$. Les environnements représentent des substitutions à venir, dans le sens de l'évaluation paresseuse, de chaque terme x_i par le terme y_i.

Exemple d'expression de processus EB[3]

La figure 1.3 présente une spécification d'un SI de gestion d'une bibliothèque. Le SI suit le cycle de vie des entités de livres (*book* de clé *bId*) et de membres (*member* de clé *mId*) qui peuvent participer aux associations emprunt (*loan*) ou réservation (*reservation*). Un livre doit d'abord être acheté (Acquire(*bId*)) avant d'éventuellement participer à un ou plusieurs emprunts. La fermeture de Kleene exprime cette itération. Cependant l'emprunt ne se fait qu'avec un seul membre simultanément grâce à l'utilisation de l'opérateur du choix quantifié. Une fois tous les emprunts terminés, le livre peut être revendu en exécutant Discard(*bId*), une action qui termine son cycle de vie. Concernant le membre, le processus est similaire. Un membre s'inscrit, puis peut faire des emprunts. On remarquera que le membre peut emprunter plusieurs livres simultanément comme le modélise l'opérateur de synchronisation quantifiée. Une fois tous ses emprunts terminés, le membre peut quitter la bibliothèque.

$$
\begin{aligned}
&\textbf{main} = \ (\ |||\, bId \in \text{BOOKID} : \textbf{book}(bId)^*\ ) \\
&\quad \|\ (\ |||\, mId \in \text{MEMBERID} : \textbf{member}(mId)^*\ ) \\
&\quad \|\ \mathsf{DisplayBorrowerByCategory}(\,)^*
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{book}(bId : \text{BOOKID}) = \mathsf{Acquire}(bId, _) \\
&\quad .\ (\ \ (|\, mId \in \text{MEMBERID} : \textbf{loan}(mId, bId))^* \\
&\quad\quad \|\ (\ |||\, mId \in \text{MEMBERID} : \textbf{reservation}(mId, bId)^*) \\
&\quad\quad \|\ \mathsf{DisplayCurrentBorrower}(bId)^* \\
&\quad\ ) \\
&\quad .\ \mathsf{Discard}(bId)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{member}(mId : \text{MEMBERID}) = \mathsf{Join}(mId) \\
&\quad .\ (\ \ (\ |||\, bId \in \text{BOOKID} : \textbf{loan}(mId, bId)^*\ ) \\
&\quad\quad \|\ (\ |||\, bId \in \text{BOOKID} : \textbf{reservation}(mId, bId)^*) \\
&\quad\quad \|\ \mathsf{DisplayNumberOfLoans}(mId)^* \\
&\quad\ ) \\
&\quad .\ \mathsf{Leave}(mId)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{loan}(mId : \text{MEMBERID}, bId : \text{BOOKID}) = \\
&\quad (\ \mathsf{Lend}(mId, bId)\ |\ \mathsf{Take}(mId, bId)\ )\ .\ \mathsf{Renew}(mId, bId)^*\ .\ \mathsf{Return}(mId, bId)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{reservation}(mId : \text{MEMBERID}, bId : \text{BOOKID}) = \\
&\quad \mathsf{Reserve}(mId, bId)\ .\ (\ isFirst(trace, mId, bId) \Longrightarrow \mathsf{Take}(mId, bId) \\
&\quad\quad |\ \mathsf{Cancel}(mId, bId)\ )
\end{aligned}
$$

FIGURE 1.3 – Une expression de processus EB^3 spécifiant le SI gérant une bibliothèque

L'interpréteur d'expressions de processus EB^{3}PAI [26] prend en entrée une spécification de système ainsi qu'une trace d'actions à exécuter. Pour chaque action, il calcule si l'expression de processus peut accepter ou non l'exécution de l'action. Dans le cas ou l'action est autorisée, il calcule une nouvelle expression de processus modélisant le nouvel état du système résultant de l'exécution. Dans le cas contraire, il retourne un message d'erreur pertinent, déduit de l'expression de processus [62, 67]. Les règles d'exécution des actions sont calculées par un ensemble de règles d'inférence définissant le comportement de chaque opérateur du langage. Ces règles sont déterministes.

Un des inconvénient de l'algèbre de processus EB^3 est son manque d'outils de vérification. Il n'existe pas de *model checker* pour EB^3 [28].

Les autres composantes d'APIS

Dans le cadre de la génération complète d'un système en utilisant l'approche APIS, d'autres spécifications sont nécessaires.

Du diagramme entités-associations au schéma de la base de données Le diagramme entités-associations de la méthode APIS s'exprime via la notation UML [77]. Il permet de définir le schéma de la base de données à générer et est également utilisé pour la génération des sorties en fonction des entrées.

De la spécification de l'interface graphique au site web La spécification de l'interface graphique permet de déterminer comment sont agencés les éléments issus du SI ainsi que leur intégration dans une interface de type Internet. Cela comprend les formulaires nécessaires à l'entrée de données et de paramètres pour les actions, les zones de retour pour afficher les résultats ainsi que la possibilité de signaler une erreur à l'utilisateur.

De l'évaluation des attributs en fonction de la trace aux requêtes de mise à jour de la base de données Sous la forme de fonctions récursives sur la trace, sont définies les valeurs que peuvent prendre les attributs. Chaque fonction est associée à un attribut et réciproquement. Ces fonctions détaillent les conséquences qu'a chaque action du système sur l'attribut auquel elle se rapporte. Ces fonctions sont ensuite associées, à un niveau moins abstrait, à des requêtes de mise à jour des attributs dans la base de données.

Des règles d'entrées-sorties aux exécutions d'actions La signature des actions permet de définir le nombre et le type des arguments d'une action, si elle en a. Ces signatures sont utilisées pour vérifier que le format d'entrée d'une action est respecté par l'utilisateur lors de ses requêtes, mais aussi pour s'assurer que l'expression de processus est correcte dans le sens où elle doit respecter ces signatures. Une sortie est également associée à chaque entrée valide du système afin de relier les modifications des attributs ou l'affichage de données à des actions valides de l'utilisateur.

1.2.2 ASTD

Partant du constat que les notations graphiques comme UML sont plus appréciées et plus compréhensibles par les analystes développeurs et que les méthodes formelles permettent l'expression de spécifications vérifiables et validables, Frappier, Gervais, Laleau, Fraikin et St-Denis ont proposé la notation ASTD (*Algebraic State Transition Diagrams*) [32]. Cette notation combine formellement les automates, les Statecharts de Harel [41] ainsi que l'algèbre de processus EB^3. Un ASTD est composé de deux parties : une structure et un état. Selon le type d'ASTD, la structure et l'état varient. La structure représente la partie graphique ainsi que la sémantique d'un ASTD tandis que l'état modélise son évolution durant l'exécution d'actions. De la structure d'un ASTD se calcule un unique état initial et un ensemble d'états finaux. Tout comme avec EB^3, des règles d'inférence définissent les exécutions possibles d'un ASTD en fonction de son type. Ces règles sont disponibles en annexe C. Les différents types d'ASTD sont :

Le type élémentaire Il est noté elem et est similaire aux places d'un automate. Ce type d'ASTD ne peut apparaître que dans des ASTD de type automate.

Le type automate Les places d'un ASTD automate sont chacune un ASTD de n'importe quel type, et les transitions peuvent être gardées. Les transitions peuvent également se faire depuis ou vers une place d'un ASTD de type automate de niveau immédiatement inférieur, à la manière des Statecharts de Harel. De plus, des états historiques peuvent être utilisés.

Le type séquence Il est associé à deux opérandes eux mêmes des ASTD de n'importe quel type. Ce type autorise l'exécution du second ASTD quand le premier est dans un état final. Un ASTD de type séquence est similaire à l'opérateur EB^3 de séquence.

Le type choix Il autorise l'exécution de l'un ou l'autre des ASTD opérandes, au choix de l'utilisateur. Ce type d'ASTD est similaire à l'opérateur EB^3 de choix.

Le type synchronisation sur Δ Il permet l'entrelacement de deux ASTD opérandes, avec synchronisation sur les actions dont les étiquettes appartiennent à Δ. Ce type d'ASTD est similaire à l'opérateur EB^3 de synchronisation.

Le type garde Il permet de définir un prédicat qui doit être vérifié avant l'exécution de son opérande. Ce type d'ASTD est similaire à l'opérateur EB^3 de garde.

Le type fermeture de Kleene Il permet l'itération de zéro, une ou plusieurs occurrences de son opérande. Ce type d'ASTD est similaire à l'opérateur EB^3 de fermeture de Kleene.

Le type appel d'ASTD Il permet de référencer un autre ASTD en lui passant d'éventuels paramètres. Ce type d'ASTD est similaire à l'opérateur EB^3 d'appel de processus.

Le type choix quantifié Il est similaire au choix quantifié EB^3.

Le type synchronisation quantifiée Il est similaire à la synchronisation quantifiée EB^3.

La figure 1.4 présente la spécification d'un SI gérant une bibliothèque. Cette spécification est similaire à celle de la figure 1.3 décrite en utilisant EB^3 hormis l'absence de la gestion de réservations pour la version ASTD.

Les ASTD apportent également des avantages liés à leur représentation graphique et leurs états. Il est par exemple possible d'associer l'exécution d'un ASTD à un animateur. On pourrait alors imaginer un animateur qui déplacerait des jetons dans les différents états d'un ASTD pour représenter l'évolution de l'exécution. Ce genre d'approche permet la validation de la spécification de manière plus aisée qu'avec EB^3. En effet, la structure d'un

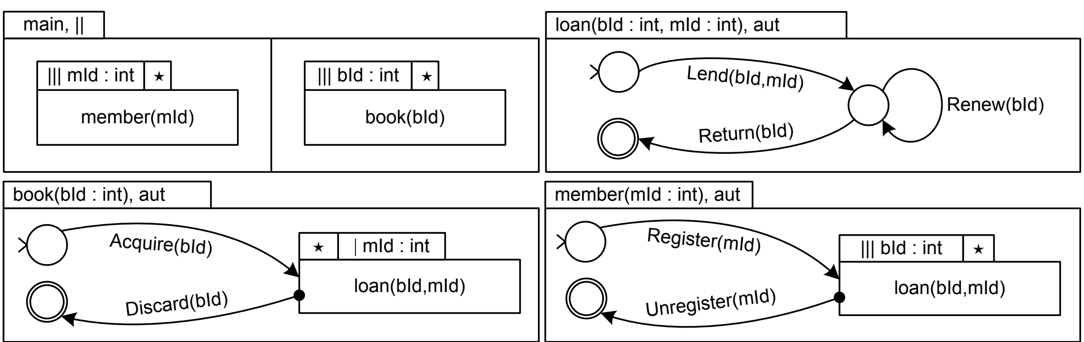


FIGURE 1.4 – Exemple d'un ASTD modélisant le SI pour une bibliothèque

ASTD est constante au cours de l'exécution d'actions, alors que l'expression de processus EB^3 d'un système évolue et peut être considérablement plus longue et complexe que la spécification initiale. De ce fait, il est toujours possible de déduire d'un ASTD quel était son état initial, chose très complexe pour les spécifications interprétées par EB^3PAI.

1.2.3 EB^3SEC et ASTDsec

EB^3SEC [51, 52] est une extension de la méthode EB^3. Alors que EB^3 permet de spécifier le comportement attendu d'un SI à l'aide d'une algèbre de processus, EB^3SEC permet de définir une politique de contrôle d'accès à appliquer sur un SI, qu'il ait été conçu avec la méthode EB^3 ou non.

L'approche se base sur l'expression de règles de contrôle d'accès modélisant l'enchaînement autorisé ou interdit d'actions en fonction de paramètres de sécurité comme l'identifiant de l'utilisateur, son rôle et son organisation par exemple. Le nombre et le type des paramètres n'est cependant pas limité à ce choix. Les seules contraintes sur les paramètres de sécurité sont : (i) leur nombre et leur ordre sont constants au sein d'une même spécification ; (ii) ils doivent pouvoir être fournis ou calculés à partir de la requête de l'utilisateur. Au niveau syntaxique, une différence apparaît si l'on compare une action EB^3 avec une action EB^3SEC. Les paramètres de sécurité sont séparés des paramètres fonctionnels comme le montre la figure 1.5. Les paramètres u et r, représentant respectivement l'utilisateur qui exécute l'action et son rôle, ne sont pas présents dans l'action EB^3.

$$\mathsf{Lend}(bId, mId)$$

$$< u, r, \mathsf{Lend}(bId, mId) >$$

FIGURE 1.5 – Les actions élémentaires EB^3 et EB^{3}SEC

$$\begin{array}{l} \text{Rule} \triangleq \\ |||\, mId \in \text{MEMBERID} : \\ \quad |\, uId \in \text{USERID} : \\ \quad\quad userOf(mId) \neq uId \implies \; < uId, _ , _ , \mathsf{lend}(_ , mId) > \end{array}$$

FIGURE 1.6 – Exemple de règle de sécurité EB^{3}SEC

La figure 1.6 décrit une règle de contrôle d'accès qui s'applique sur un SI gérant une bibliothèque. Elle exprime que pour pouvoir emprunter un livre, la personne enregistrant la transaction dans le système doit être différente du membre qui emprunte le livre. Ce genre de règles s'appelle une séparation des devoirs (SoD, *Separation of Duty*). Elle permet d'éviter des situations où un utilisateur du système pourrait tirer avantage de son accès au SI en nécessitant la confirmation d'un autre utilisateur au moment de l'exécution.

D'une manière similaire à EB^3 et EB^{3}SEC, les ASTD sont étendus en ASTDsec. Les actions élémentaires d'un ASTDsec sont alors similaires à celles de EB^{3}SEC et incluent des paramètres de sécurité.

1.2.4 La méthode B

La méthode B [3] est une méthode de spécification formelle permettant de décrire un système de son analyse à son implémentation. Son formalisme s'appuie sur des notations mathématiques utilisant la théorie des ensembles et la logique du premier ordre pour spécifier à la fois le système et ses propriétés. La méthode B se base sur le concept de machine abstraite représentant le système. Cette machine abstraite décrivant le comportement général est détaillée au fur et à mesure de la modélisation pour devenir plus concrète et plus proche de l'implémentation finale du système. Ce processus est appelé raffinement. Il joue

un rôle central dans la méthode B. Les étapes de raffinement sont prouvées afin de garantir que le modèles successifs modélisent le même système et qu'aucune erreur n'est introduite lors de la modélisation. De nombreux outils comme l'atelier B [12], les prouveurs B ou ProB [56] appuient le développement de systèmes avec la méthode B.

Machines

Une machine est composée d'une partie statique (codant les constantes et les propriétés du système) et d'une partie dynamique (des variables et des opérations qui les modifient). Les variables sont typées par des invariants et les opérations ne peuvent pas modifier le typage. Les invariants peuvent également être utilisés pour définir des propriétés sur les variables qui doivent rester vraies durant toute l'exécution de la machine. Les machines peuvent également inclure ou voir d'autres machines et accéder ainsi en lecture ou en écriture à certains éléments de la spécification. C'est le cas par exemple des machines raffinement. Si une machine R raffine une machine M alors R aura par exemple accès aux ensembles et aux constantes de M.

Opérations et substitutions

Les opérations sont exprimées sous la forme de substitutions sur les variables qui peuvent être non déterministes. La théorie des substitutions généralisées introduit une sémantique formelle pour chaque type de substitution. Cette sémantique se définit par l'application d'une substitution S sur un prédicat P et est notée $[S]P$. La liste complète des substitutions et de leur sémantique est présentée dans [3]. Une substitution particulière dite « précondition » est cependant plus importante que les autres. Elle est utilisée par des outils comme ProB [55] pour animer une machine B. Avec ProB, les opérations peuvent être exécutées si et seulement si leur précondition est vraie. Ce comportement est différent de la sémantique de B qui veut que si on exécute une opération sans que sa précondition ne soit vraie, alors le comportement résultant n'est pas connu. Selon le degré d'abstraction de la machine, certaines substitutions peuvent être illégales, et ne doivent pas apparaître dans les opérations. Par exemple, à chaque étape de raffinement, les opérations tendent à devenir de plus en plus déterministes jusqu'à la machine la plus concrète, appelée implémentation, qui ne doit plus contenir de substitutions non déterministes.

Obligations de preuve

Les obligations de preuves sont des formules logiques permettant de garantir la correction des composants d'une spécification B. Elles peuvent être générées par l'outil de conception atelier B [12] et doivent être prouvées soit automatiquement par un prouveur logiciel, soit manuellement par le développeur. Les obligations de preuves "préservation de l'invariant" permettent par exemple de prouver que si l'on exécute une opération sur des variables qui respectent les invariants de la machine, alors les nouvelles valeurs des variables respectent toujours les invariants. L'obligation de preuves "initialisation" permet de vérifier que l'initialisation de la machine établit les invariants. D'autres obligations de preuves sont générées lors des étapes de raffinement. Le calcul des obligations de preuves se fait à l'aide d'un prédicat *R* et d'une substitution *S*. La théorie des substitutions généralisées définit une sémantique pour chaque substitution appliquée sur *R*. On note *Q*, le prédicat résultant de l'application de la substitution *S* sur le prédicat *R* de la manière suivante : $Q = [S]\ R$.

En fonction des choix fait lors de la modélisation des opérations à l'aide de substitutions, différentes obligations de preuve peuvent être générées. Trois types de substitutions sont généralement utilisées pour décrire des opérations : la substitution précondition ; la substitution sélection ; la substitution conditionnelle.

Substitution précondition PRE La substitution précondition se note **PRE** *P* **THEN** *T* **END** où *P* est un prédicat et *T* une substitution. Elle exprime le fait si le prédicat *P* est vrai, alors la substitution *T* est appliquée. Si *P* est faux, alors le comportement n'est pas connu, et l'état de la machine peut être modifié de manière quelconque. La sémantique de cette substitution dans les obligations de preuves est :

$$[\textbf{PRE}\ P\ \textbf{THEN}\ T\ \textbf{END}]\ R = P \wedge [T]\ R$$

Ainsi, lors de l'utilisation de la substitution précondition dans une opération, le prédicat *P* doit être vrai pour pouvoir prouver les obligations de preuves associées à cette opération.

Substitution sélection SELECT La substitution sélection se note **SELECT** *P1* **THEN** *T1* **WHEN** *P2* **THEN** *T2* **END**. Elle signifie que si *P1* est vrai, alors *T1* est appliquée, si *P2* est vrai, alors *S2* est appliquée. Si *P1* et *P2* sont vrais simultanément, alors *T1* ou

T2 est appliquée, le choix étant fait de manière non déterministe. La sémantique de cette substitution dans les obligations de preuves est :

[**SELECT** *P1* **THEN** *T1* **WHEN** *P2* **THEN** *T2* **END**] $R = (P1 \Rightarrow [T1]\, R) \wedge (P2 \Rightarrow [T2]\, R)$

Ainsi, lors de l'utilisation de la substitution sélection dans une opération, si *P1* et *P2* sont faux, la sémantique du **SELECT** retourne vrai et ne bloque pas la preuve associée à cette opération.

Substitution conditionnelle IF La substitution conditionnelle se note **IF** *P1* **THEN** *T1* **ELSIF** *P2* **THEN** *T2* **END**. Elle signifie que si *P1* est vrai, alors *T1* est appliquée, si *P1* est faux et que *P2* est vrai, alors *S2* est appliquée. La sémantique de cette substitution dans les obligations de preuves est :

[**IF** *P1* **THEN** *T1* **ELSIF** *P2* **THEN** *T2* **END**] $R = (P1 \Rightarrow [T1]\, R) \wedge ((\neg P1 \wedge P2) \Rightarrow [T2]\, R)$

Ainsi, lors de l'utilisation de la substitution conditionnelle dans une opération, si *P1* et *P2* sont faux, la sémantique du **IF** retourne vrai et ne bloque pas la preuve associée à cette opération, tout comme pour la substitution **SELECT**.

1.2.5 La méthode Event-B

Event-B [4] est une méthode basée sur la méthode B [3] avec un aspect évènementiel. Elle permet de spécifier le comportement d'un système appelé « système d'évènements » de manière à inclure ses propriétés critiques dans la conception. Dans la méthode Event-B, le modèle initial est successivement raffiné plusieurs fois pour atteindre une modélisation correcte par construction. Les propriétés introduites sont également prouvées et l'utilisation d'une notation mathématique permet une modélisation indépendante de toute considération informatique. La communauté Event-B étant très active, et du fait de l'utilisation industrielle de la méthode, des outils performants sont proposés. Des outils de preuve et des vérificateurs de modèles (*model checkers*) utilisant les logiques temporelles [76], comme LTL et CTL, permettent de vérifier de propriétés temporelles sur les modèles développés avec Event-B.

Machines et contextes

Une machine est une spécification élémentaire en Event-B. Les machines peuvent inclure des contextes qui définissent des éléments statiques comme des constantes et leurs propriétés. Les machines modélisent le comportement dynamique. Elles permettent de définir des variables ainsi que des invariants, *i.e.* des propriétés que doivent toujours vérifier les variables. Les machines sont également composées d'évènements qui modifient les variables.

Évènements

En Event-B, les évènements remplacent les opérations B. Un évènement d'initialisation permet de définir les valeurs initiales des variables. Ces valeurs doivent établir les invariants. Les autres évènements ont des gardes qui garantissent des propriétés. Les gardes sont définies dans la clause **WHEN** et sont mises en conjonction. Si l'une des gardes d'un évènement est fausse, l'évènement ne peut pas être déclenché. Ce comportement est donc différent de B où les opérations dont la précondition est fausse peuvent être exécutées sans que l'on puisse garantir le résultat. Les évènements sont également composés d'actions qui modifient les variables de la machine. Les actions sont définies dans la clause **THEN** de l'évènement. Il n'existe pas en Event-B de structure similaire aux substitutions **IF** ou **WHILE** de B. Les actions d'un évènement sont exécutées de manière concurrente, et il n'est pas possible de modifier la même variable dans deux actions d'un même évènement. Les actions peuvent utiliser les paramètres de l'évènement définis par une clause **ANY**. Des obligations de preuves sont générées pour garantir que les actions ne permettent pas d'atteindre un état de la machine dans lequel les invariants sont violés. D'autres obligations de preuves comme l'absence d'interblocage garantissent des propriétés de la machine.

Raffinement

La notion de raffinement s'applique à la fois aux machines mais aussi aux évènements. On peut ajouter de nouveaux évènements lors d'un raffinement en Event-B. On considère que ces nouveaux évènements raffinent *skip*, un évènement qui n'a pas de garde et ne modifie pas les variables de la machine. Un évènement abstrait peut être raffiné par plusieurs évènements concrets. Plusieurs évènements abstraits peuvent être raffinés par un évène-

ment concret. Le raffinement d'évènements génère des obligations de preuves, dont une est de vérifier que les gardes des évènements concrets sont plus fortes que les gardes des évènements abstraits. Il n'y a pas de notion d'implémentation en Event-B.

1.3 Traductions de langages formels

Plusieurs méthodes de traduction de langages formels existent. Les plus proches de notre approche sont détaillées dans les sections suivantes.

1.3.1 Traduction EB^3 vers B – Attributs

La traduction de EB^3 vers B de Gervais et al. [40] permet de générer une machine B décrivant le fonctionnement des actions d'un SI à partir d'une spécification EB^3.

La méthode se décompose en plusieurs étapes :

1. Générer les ensembles et les invariants de la machine B à partir du diagramme entités-associations de EB^3. Cette génération se base sur [70] qui propose une méthode de traduction de diagrammes objets en B.
2. Définir l'état initial de la machine B en générant les substitutions de l'opération d'initialisation à partir des points fixes des définitions récursives des attributs.
3. Générer les opérations B et les pré-conditions de typage des paramètres B à partir des signatures des actions EB^3.
4. Générer les substitutions mettant à jour les attributs dans les opérations correspondantes à partir des définitions récursives des attributs EB^3.

Cette approche de traduction ne peut cependant pas être réutilisée dans le cadre d'une traduction de spécifications de politiques de contrôle d'accès ASTD en B. En effet, l'approche présentée ne tient pas du tout compte de l'expression de processus dans le cadre de la traduction. Si les principes utilisés pour la traduction, comme la génération de la partie statique puis de la partie dynamique, pourraient être réutilisés dans le cadre d'une traduction des ASTD en B/Event-B, les autres éléments sont spécifiques aux traductions de fonctions récursives sur les attributs.

1.3.2 Traduction EB^3 vers B – Expressions de processus

Dans [36], Laleau et Frappier proposent une approche permettant de prouver des propriétés d'ordonnancement sur un spécification B d'un SI. Pour cela, les propriétés sont exprimées en utilisant EB^3 et une machine B traduisant cette expression de processus est construite. Une autre machine B représente le SI sous forme de spécification « états transitions » dans laquelle des variables codent les états de la spécification, et les opérations modifient les variables d'état. Les auteurs mettent alors en relation la machine B du SI et la machine issue de la traduction de la propriété d'ordonnancement. Si on peut prouver que la machine du SI est un raffinement de la machine traduite, alors la spécification du SI respecte bien la propriété d'ordonnancement.

La traduction de la spécification EB^3, notée E, en B se déroule en plusieurs étapes :

- Plusieurs ensembles B sont créés : l'ensemble des évènements de E, l'ensemble des traces acceptées par E noté $T(E)$.
- Une variable t est créée dans la machine B. Cette variable représente la trace des évènements acceptés par la spécification. Le typage de cette variable est $t \in T(E)$.
- À l'état initial, la trace t est vide.
- Une opération par évènement est créée. Elle vérifie si la trace composée de t à laquelle on ajoute l'évènement est une trace valide. Si oui, la trace t est mise à jour, sinon, elle reste inchangée.

La spécification du SI est modélisée en B par une machine « états transitions ». Afin de prouver la relation de raffinement entre les deux machines, il faut écrire un invariant de collage pour montrer comment les valeurs des variables d'état sont liées à t, la variable abstraite stockant la trace acceptée. Les obligations de preuves générées par le raffinement conduisent également à la génération d'un ELTS représentant l'expression de processus E. Un ELTS (*Extended Labelled Transition Diagram*) est un diagramme états-transitions étendu qui permet notamment de modéliser les états des LTS classiques. Les ELTS seront alors définis de manière formelle et deviendront les ASTD. À partir de cet ELTS généré, la preuve de certaines équivalences requises par les obligations de preuves sont plus simples.

Plusieurs éléments de cette approche nous intéressent. En effet, dans le cadre de cette thèse, nous devons transformer des modèles exprimés à l'aide de EB^3. La traduction des expressions de processus EB^3 en ELTS, bien que peu détaillée, est une première étape di-

rectement utilisable dans nos travaux. La traduction de EB^3 vers B en se basant sur les traces semble difficile d'utilisation. En effet, dans le cadre de la définition de propriétés d'ordonnancement, peu d'évènements et d'opérateurs sont utilisés. L'exemple le plus complexe présenté dans [36] utilise trois évènements et quatre opérateurs. Dans le cadre de nos travaux, ces nombres peuvent être bien plus élevés. Or pour que cette traduction soit possible, il faut pouvoir être capable de calculer $T(E)$, l'ensemble des traces acceptées par E, qui peut être infini du fait de l'opérateur de fermeture de Kleene par exemple. Ce calcul est loin d'être trivial, mais résulte en une traduction simple et identique pour tous les évènements de la spécification.

1.3.3 csp2B

csp2B [11] est une méthode permettant de combiner des spécifications écrites en CSP [44] et des machines B. L'expression de processus CSP agit alors comme contrôleur d'une machine B en définissant des contraintes d'ordonnancement sur les exécutions d'opérations B. Pour cela, une traduction outillée d'un sous-ensemble de CSP en B a été construite. La traduction d'une spécification CSP se base sur sa structure syntaxique. Les appels à des processus sont dit gardés en CSP, ce qui signifie qu'avant l'appel d'un processus, une action élémentaire doit être exécutée. En se basant sur cette structure syntaxique, l'outil de traduction génère un ensemble d'états et des opérations. Les opérations correspondent aux actions qui gardent les processus et modifient l'état courant de la machine, alors qu'un état modélise le processus en cours de la spécification CSP. L'état courant est stocké dans des variables B dites de contrôle. La composition parallèle des processus est traduite en utilisant la composition parallèle des substitutions B notée $\|$. Les canaux de communication CSP sont également pris en compte avec l'utilisation d'une variable tampon entre les évènements, et les entrelacements quantifiés sont traduits en utilisant des fonctions B.

Si l'on résume l'approche de traduction, elle correspond à générer une machine B similaire à un système à transitions étiquetées (*LTS, Labelled Transition System*). L'approche est similaire à celle présentée pour les SI dans la sous-section 1.3.2. Les états du LTS sont alors codés dans les variables de contrôle et chaque transition qui code une action CSP correspond à une opération. Par exemple, soit une spécification CSP codant le fonctionnement d'une machine distribuant du thé ou du café sous réserve que l'on s'acquitte d'un paiement

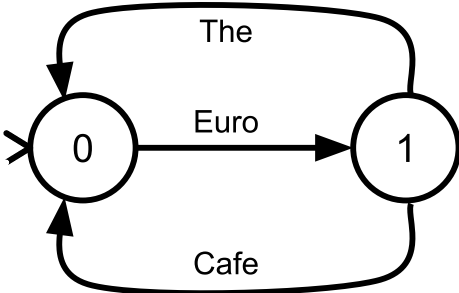


FIGURE 1.7 – LTS correspondant au processus CSP décrivant une machine à thé et à café

d'un euro :

$$
\begin{aligned}
Machine &= AttendreEuro \\
AttendreEuro &= Euro \rightarrow Servir \\
Servir &= The \rightarrow AttendreEuro \; [] \; Cafe \rightarrow AttendreEuro
\end{aligned}
$$

Cette spécification CSP conduit à la génération du LTS présenté dans la figure 1.7. Une machine B est ensuite créée pour coder en B le LTS ainsi généré. Cette traduction est obtenue en créant un ensemble d'états $STATES = \{s0, s1\}$ et trois opérations *the*, *cafe* et *euro* codant les transitions du LTS par des substitutions sur la variable stockant l'état courant.

Cette méthode de traduction est similaire et comparable à celle recherchée dans notre approche. Cependant de fortes restrictions s'appliquent sur la traduction des synchronisations. La spécification doit respecter des contraintes de formes précises. Dans le cadre d'un opérateur de synchronisation, celui-ci ne doit pas apparaître en profondeur dans la spécification. L'auteur assure cependant que ces restrictions n'empêchent pas l'utilisation de son approche sur des systèmes de grande taille [42].

De ce fait dans le cadre de la traduction des ASTD en B/Event-B, cette approche ne peut servir que dans le cas des opérateurs choix, séquence et entrelacement. Il est nécessaire d'envisager d'autres algorithmes de traduction pour les autres opérateurs.

1.3.4 UML-B

Il existe plusieurs traductions d'UML vers B. Elles ne correspondent pas toutes à la traduction que nous recherchons pour les ASTD. Certaines traduisent les diagrammes de classes en machines B, d'autres se basent sur les diagrammes états-transitions. Nous nous intéressons ici à l'approche UML-B qui traduit le plus grand nombre de fonctionnalités des diagrammes états-transitions UML en B.

UML-B de Snook et Butler [80] est une approche outillée [79] visant à intégrer certains diagrammes UML dans B. Cette approche vise, selon les auteurs, à corriger le manque de sémantique formelle reproché à UML et à rendre la méthode B plus facilement utilisable. À partir de packages, diagrammes de classes et diagrammes états-transitions UML et de stéréotypes définis dans un profil UML-B, l'outil U2B génère des machines B. Une version plus récente [81] propose également de travailler avec Event-B. Dans la suite, nous étudions la version UML-B pour Event-B.

Dans UML-B, chaque diagramme a un but précis :

1. Le **package** représente le projet dans son ensemble. Il définit les liens entre les machines et les contextes.
2. Les **diagrammes de classes** permettent de définir les contextes. Ces diagrammes sont appelés diagrammes de contextes. Ils ne contiennent que des constantes et des propriétés de typage. Les propriétés des constantes (les axiomes en Event-B) et les théorèmes sont attachés aux classes. Les diagrammes de classes permettent également de définir les machines.
3. Les **diagrammes états-transitions** sont associés aux diagrammes de classes des machines pour décrire les modifications induites par l'exécution d'une transition sur la spécification. Les transitions correspondent à des évènements Event-B. Des invariants Event-B peuvent également être associés aux états.

   L'approche permet ainsi de spécifier à la fois en UML et Event-B. Certaines tâches complexes comme l'ordonnancement des évènements ou la définition d'invariants locaux sont grandement simplifiées par l'utilisation de diagrammes états-transitions. Cette notation permet de spécifier graphiquement les contraintes d'ordonnancement des évènements et s'assurer que le code Event-B généré correspond bien à la sémantique des diagrammes états-transitions. De même, la visualisation des dépendances

entre machines et contextes est plus simple avec les packages UML. Enfin, les traductions des diagrammes de classes en Event-B permettent une représentation normalisée des classes UML en Event-B. L'ensemble agit comme un outil permettant d'améliorer la cohérence de la spécification avec les modèles UML et d'augmenter la lisibilité d'une spécification formelle Event-B.

Dans le cadre de nos travaux, UML-B ne peut pas être réutilisé tel quel. La traduction des diagrammes états-transitions en B/Event-B n'est possible avec UML-B que lorsque le diagramme états-transitions est défini en lien avec un diagramme de classes. Les diagrammes états-transitions sont considérés comme dépendant des diagrammes de classes. Dans le cadre des ASTD, une spécification peut se faire sans que nécessairement un diagramme de classes ne soit disponible. De plus, les diagrammes états-transitions considérés ne peuvent pas utiliser la notion de parallélisme, notion indispensable dans le cadres des SI et disponible dans les ASTD. De même, les états OR et AND ne sont pas pris en compte. Seuls les diagrammes états-transitions en tant qu'automates hiérarchiques sont acceptables dans une spécification UML-B.

1.3.5 Statecharts vers B

Les Statecharts au sens de Harel [41] sont une notation graphique formelle de modélisation de processus. Sekerinski *et al.* proposent dans [87] une méthode de traduction des Statecharts en B. L'approche de traduction se fait par la définition d'un méta-modèle des Statecharts exprimé à l'aide de diagrammes de classes et suit les algorithmes de traduction de spécifications UML en B de [60]. L'article décrit également une approche de normalisation des Statecharts afin de réduire le nombre de patrons de traduction.

Le processus de traduction s'effectue en plusieurs étapes. Tout d'abord, le Statechart à traduire est normalisé. Cette normalisation s'effectue suivant trois conditions : (i) les transitions pointant vers des états AND doivent être des transitions pointant vers tous les sous-états de l'état AND ; (ii) si une transition pointe vers un sous-état d'un état AND, alors elle doit aussi pointer vers les autres sous-états de cet état AND ; (iii) les noms des états sont uniques et si nécessaire l'algorithme les nomme de façon à respecter cette contrainte. Une fois le Statechart normalisé, neuf critères sont vérifiés pour qu'il soit qualifié de valide.

1. Les transitions ne sont pas entre des états concurrents.

2. Les transitions ne peuvent se séparer, *i.e.* avoir plusieurs états destinations, que sur des états concurrents.
3. Les transitions ne peuvent se réunir, *i.e.* avoir plusieurs états sources, que depuis des états concurrents.
4. Il ne doit pas y avoir de cycles dans les transitions internes.
5. Tous les états initiaux doivent permettre d'atteindre au moins un état final.
6. Les transitions *init* doivent aller vers un sous-état.
7. Les états initiaux ne peuvent pas être la cible de transitions.
8. Les transitions *init* ne doivent pas comporter d'évènements ou de gardes.
9. Les transmissions *broadcast* ne génèrent pas le même évènement deux fois.

Une fois ces vérifications effectuées, des variables codant les états sont générées. Puis la traduction des transitions est effectuée en fonction de patrons. Une fois cette traduction effectuée, des optimisations sont faites sur les substitutions obtenues. Ces optimisations évitent d'avoir des substitutions de type **IF** *C* **THEN** *T* **ELSE** *skip*, ou des substitutions **CASE** avec des *skip*. Enfin, les transitions internes sont traduites de sorte que si elles peuvent être exécutées dans un état, alors elles le sont.

Cette approche peut être intéressante pour définir une traduction des ASTD en B. En effet, les Statecharts de type OR sont similaires aux ASTD de type choix, les Statecharts de type AND sont similaires aux ASTD de type synchronisation et les ASTD ont également la notion d'état historique ainsi que d'états imbriqués. Cependant, les ASTD ont également d'autres opérateurs qui n'existent pas dans les Statecharts, comme les quantifications de choix ou de synchronisation qui nécessitent le développement d'une approche de traduction spécifique. On pourra cependant réutiliser l'approche de traduction des états par des ensembles énumérés et l'état courant modélisé par un ensemble de variables.

1.3.6 Coder une algèbre de processus en Event-B

Dans [1], Ait-Ameur *et al.* définissent une méthode de traduction générique d'algèbres de processus en Event-B. Ils proposent d'associer aux expressions de processus pouvant être décrites par une algèbre de processus définie formellement par une grammaire BNF une représentation sous forme d'une hiérarchie de machines Event-B. Chaque règle de la

BNF de la forme $T ::= E\ op\ F$ est alors traduite en deux machines Event-B. La première modélise T et ne contient qu'un évènement nommé *eventT*. La seconde machine est un raffinement de la première et correspond à $E\ op\ F$. Elle contient deux nouveaux évènements *eventE* et *eventF* modélisant le comportement respectivement de E et de F ainsi que de l'opérateur op. On déclenche alors l'exécution de *eventE* et *eventF* puis celle de *eventT* quand *eventE* et *eventF* ont terminé leur exécution. Pour cela, un variant est introduit et les évènement sont gardés en utilisant ce variant. Ce variant permet d'assurer que l'exécution des évènements *eventE* et *eventF* terminera, condition nécessaire pour garantir que l'exécution de *eventT* ait lieu. Un variant B est une variable entière positive qui décroît à chaque exécution d'un évènement. Les auteurs détaillent ensuite le processus de traduction en l'appliquant sur l'algèbre de processus CTT [74]. Chaque opérateur est alors détaillé soit en donnant sa traduction, soit en le redéfinissant à l'aide d'autres opérateurs.

Cette approche est directement applicable à l'algèbre de processus EB^3 car EB^3 respecte les critères fournis par les auteurs. En effet, l'algèbre de processus EB^3 peut être formellement décrite à l'aide d'une grammaire BNF. On peut donc obtenir une machine Event-B codant le comportement d'une expression de processus. Cette machine est le résultat de n opérations de raffinement, n étant le nombre d'opérateurs de l'expression de processus initiale. Cependant, des évènements annexes, n'apparaissant pas dans la spécification initiale apparaîtront dans la machine Event-B finale. Ces évènements sont nécessaires pour coder par exemple la réinitialisation de certaines variables induites par une itération de la fermeture de Kleene d'une expression de processus. Ce genre d'évènements artefacts fait que la machine Event-B peut être dans un état qui ne correspond pas à celui de la spécification EB^3. De ce fait, certaines preuves de propriétés sur le modèle Event-B peuvent ne pas être transposables sur l'expression de processus.

Une des limites de cette approche est que, du fait de sa généricité, elle nécessite de proposer une traduction de chaque opérateur. Si cette approche peut être simple et est bien décrite par les auteurs pour les opérateurs usuels comme la séquence ou le choix, les opérateurs plus exotiques comme $\pitchfork$ ne sont pas traduisibles immédiatement. La synchronisation faible en EB^3, notée $\pitchfork$, a un comportement similaire à un entrelacement, sauf si les deux opérandes peuvent exécuter au même moment l'action, dans ce cas l'opérateur se comporte comme une synchronisation. Ce genre d'opérateur n'est pas traité dans [1]. Finalement, les ASTD ne peuvent pas bénéficier de cette approche puisqu'ils ne peuvent pas être formali-

sés en utilisant une grammaire BNF du fait de la présence d'automates dans les types des ASTD. Le type automate est indispensable aux ASTD, puisque tout ASTD feuille est de type élémentaire, c'est-à-dire une place d'un automate.

1.3.7 Synthèse

Les approches de traduction de langages formels en B ou Event-B existantes ne peuvent pas être appliquées telles quelles aux ASTD ou à EB^3 car elle ne couvrent que des sous-ensemble des notations que nous utilisons, ou ne sont tout simplement pas applicables dans certains cas. Cependant, certains de ces sous-ensembles peuvent être réutilisés dans nos travaux. On peut distinguer deux approches de traduction. Une première approche est la traduction s'appuyant sur les méta-modèles. Cette approche est similaire aux transformations de modèles de l'approche MDE. Elle permet un outillage de la traduction grâce à des langages comme ATL [48] permettant d'exprimer les liens entre des méta-modèles de deux langages et d'ainsi effectuer la traduction de modèles exprimés dans un langage dans un autre. La seconde approche est de donner des patrons de traduction après avoir défini une forme normale de la structure à traduire. Ainsi, les patrons définissent une liste exhaustive des structures et peuvent éventuellement s'appliquer de manière récursive.

1.4 De la spécification de la politique de contrôle d'accès à l'implémentation

Selon [85], une spécification de politique de contrôle d'accès est dite implémentée si chaque action autorisée par l'implémentation l'est aussi par la spécification de la politique. En considérant cette définition, l'implémentation d'une politique de contrôle d'accès peut être beaucoup moins permissive que sa spécification. En effet, la politique de contrôle d'accès qui refuse toutes les exécutions est donc une implémentation possible de toute politique de contrôle d'accès. Dans les sections qui suivent, nous étudions des implémentations de politiques de contrôle d'accès.

1.4.1 Algèbre de politiques de contrôle d'accès et génération de politique XACML

Dans [78], les auteurs proposent une algèbre de politiques de contrôle d'accès ainsi que la traduction de cette algèbre en règles XACML [72]. Les auteurs introduisent pour cela une notation formelle des règles de contrôle d'accès, exprimées comme des quadruplés contenant le rôle de l'utilisateur, l'action qu'il veut effectuer, l'heure à laquelle il veut effectuer son action et la décision prise par la politique (*granted* ou *denied*). La politique reçoit des triplets {rôle, action, heure} et répond soit *granted* ou *denied* quand une règle couvre l'ensemble du triplet reçu, soit *N/A* si aucune règle de la politique ne correspond. Les auteurs introduisent ensuite des opérateurs de combinaison des politiques définis sous la forme de tables de vérité sur une logique ternaire (*granted* correspond à *true*, *denied* à *false* et *N/A* à *unknown*). Les opérateurs proposés sont similaires aux opérateurs s'appliquant sur un treillis dont la relation d'ordre partiel est $granted > N/A$, $denied > N/A$. L'opérateur $+$ correspond au *max*, l'opérateur & au *min* du treillis. D'autres opérateurs, dont la projection et la négation, sont introduits. Les auteurs montrent que dans le cadre de règles écrites sous la forme de quadruplés sans condition, toutes les politiques peuvent être exprimées avec leur algèbre, et ils déterminent l'ensemble minimal d'opérateurs pour le faire.

Afin de proposer une traduction en XACML, chaque politique de contrôle d'accès est traduite en un MTBDD (*Multi-Terminal Binary Decision Diagram*) [37]. Un MTBDD est un diagramme de décision binaire, *i.e.* une représentation graphique de fonctions booléennes, qui peut avoir plus de deux états terminaux. Les BDD ont pour états terminaux *true* et *false*. Ici, dans le cadre de MTBDD, les auteurs donnent comme état terminaux *granted*, *denied* et *N/A*. Les variables booléennes du MTBDD correspondent aux conditions atomiques des règles. Par exemple, si une règle autorise l'action *update* aux utilisateurs de rôle *manager* et la refuse aux utilisateurs de rôle *staff* alors on peut créer des variables booléennes x_0, x_1 et x_2 telles que :

$$x_0 \triangleq action = update \quad x_1 \triangleq role = manager \quad x_2 \triangleq role = staff$$

On remarque que x_1 et x_2 sont mutuellement exclusives. Puis on peut définir une fonction booléenne $f(x_0, x_1, x_2)$ qui retourne *granted*, *denied* ou *N/A* selon les valeurs des variables

booléennes.

$$\begin{aligned} f(x_0,x_1,x_2) = granted &\Leftrightarrow x_0 \wedge x_1 \\ f(x_0,x_1,x_2) = denied &\Leftrightarrow x_0 \wedge x_2 \\ f(x_0,x_1,x_2) = N/A &\Leftrightarrow \neg(x_0 \wedge x_1) \wedge \neg(x_0 \wedge x_2) \end{aligned}$$

Le MTBDD associé à la politique constituée par la règle est la représentation MTBDD de la fonction f.

Une fois toutes les politiques traduites, les auteurs fournissent les algorithmes de combinaison des MTBDD correspondant à chacun des opérateurs de leur algèbre de politiques de contrôle d'accès. On obtient ainsi un MTBDD représentant la politique complète. Une fonction de traduction en XACML est donnée pour traduire les chemins menant à un état terminal d'un MTBDD en une règle XACML. Les auteurs ont cependant remarqué que le nombre de règles générées était exponentiel par rapport au nombre de noeuds du MTBDD. Cependant des techniques de réduction logiques classiques de la théorie des BDD [24] ont conduit à une réduction de 75% à 99% du nombre de règles XACML générées. Ainsi, l'implémentation selon cette approche se définit comme suit :

1. Si l'implémentation autorise une exécution, la spécification doit l'autoriser.
2. Si l'implémentation refuse une exécution, la spécification doit la refuser.
3. Si l'implémentation ne peut pas calculer une réponse à donner pour une exécution, la spécification non plus.

Cette définition est beaucoup plus forte que celle de [85] qui ne comprend que le premier point. Cependant, les auteurs de [78] n'ont pas montré que ces propriétés sont vraies pour leur implémentation.

1.4.2 Modélisation Event-B de politiques de sécurité et implémentation par raffinement

Dans [10], l'auteur propose une formalisation des politiques de contrôle d'accès en utilisant Event-B. Cette formalisation est appliquée sur les politiques modélisées avec le

modèle RBAC [83]. La modélisation fait appel aux concepts d'utilisateurs, de permission et de politique de contrôle d'accès. Ainsi un utilisateur *user* a la permission *perm* dans la politique *policy* si et seulement si $(user \mapsto perm) \in PERMISSIONS_OF(policy)$. La fonction $PERMISSIONS_OF$ représente l'ensemble des couples utilisateur/permission autorisés par la politique de contrôle d'accès. Dans cette approche, l'auteur considère uniquement les permissions de la politique, *i.e.* la politique ne définit qu'un ensemble de permission, et non un ensemble d'interdiction. La politique est donc appliquée avec le principe de « tout ce qui n'est pas autorisé explicitement est interdit ». L'auteur détaille également la formalisation de la politique d'administration de la politique de sécurité. Cette politique détaille la façon dont la politique de contrôle d'accès peut être modifiée en cours d'utilisation.

La vérification de l'implémentation d'une politique de contrôle d'accès modélisée en Event-B est détaillée. Cette vérification se fait par raffinement. Une politique P_2 implémente P_1 si la machine Event-B de P_2 raffine celle de P_1. L'auteur propose une obligation de preuves supplémentaire à vérifier afin de garantir l'implémentation. Il s'agit d'un théorème qui, exprimé en langage naturel, dit « chaque action autorisée par l'implémentation l'est aussi par la politique ».

Enfin, l'auteur reprend les opérateurs de l'algèbre de processus de [78] et exprime comment les politiques composées à l'aide de cette algèbre sont composés en Event-B. Il effectue également les preuves montrant que la composition de deux implémentations de deux politiques et l'implémentation de la composition de deux politiques sont équivalentes au sens des actions autorisées par la politique.

1.4.3 Implémentation WS-BPEL de politiques de sécurité exprimées avec la notation ASTD

Dans [19], les auteurs proposent une implémentation WS-BPEL [73] dans une architecture orientée service (SOA) d'une politique de contrôle d'accès exprimée à l'aide de la notation ASTD. Après une description de l'architecture SOA, les auteurs détaillent le processus de transformation d'une spécification ASTD de politique de contrôle d'accès en processus BPEL. Cette politique est exprimée en utilisant des patrons correspondant aux règles de permission, d'interdiction, d'obligation et de séparation des devoirs. Chaque règle de la politique est l'instanciation d'un de ces patrons. Les règles de transformation d'ASTD vers

BPEL sont définies sur ces patrons. L'implémentation de ces règles de transformation peut, selon les auteurs, s'effectuer par la définition de règles de transformation des méta-modèles en utilisant le langage ATL [49]. Cette approche nécessite un méta-modèle des ASTD ainsi qu'un méta-modèle BPEL, déjà disponible. L'approche nécessite également la génération d'un document WSDL (*Web Service Description Language*) afin d'intégrer l'implémentation dans une architecture SOA. Ce document est également généré depuis la spécification ASTD de la politique de contrôle d'accès. Il définit les interfaces du service, les messages qu'il peut recevoir ou émettre ainsi que le typage des paramètres des messages.

L'implémentation obtenue est un filtre de contrôle d'accès nommé PDP (*Policy Decision Point*) dans l'architecture SOA. Ce filtre respecte une architecture définie à l'aide d'un méta-modèle et de diagrammes de séquences. Cette architecture est l'architecture cible du projet Selkis. Les différentes implémentations qui sont proposées dans le cadre du projet doivent respecter cette architecture.

La politique définie par ce filtre est appliquée au système existant par l'utilisation de PEP (*Policy Enforcement Point*) qui interceptent les requêtes avant leur exécution par un service et les transmettent au PDP. Selon la réponse du PDP, la requête est alors, en cas d'acceptation, transmise au service initial ou retournée à l'utilisateur avec un message d'erreur. L'un des avantages de l'approche, c'est qu'elle ne nécessite pas de modification du système que l'on souhaite sécuriser.

1.4.4 Synthèse

Concernant la définition formelle d'un filtre de contrôle d'accès puis son implémentation, peu de travaux ont été conduits. Les travaux étudiés suivent deux approches pour générer une implémentation. La première est de raffiner le modèle initial de la politique de contrôle d'accès. Une preuve de ce raffinement et l'ajout de théorèmes à prouver permettent alors de garantir que l'implémentation respecte bien certaines propriétés. La seconde approche consiste à définir des règles de traduction entre le langage de spécification et le langage d'implémentation. Cette approche est moins formelle si les règles de traduction ne sont pas prouvées et ne garantissent pas la conservation de la sémantique initiale de la politique de contrôle d'accès.

1.5 Conclusion

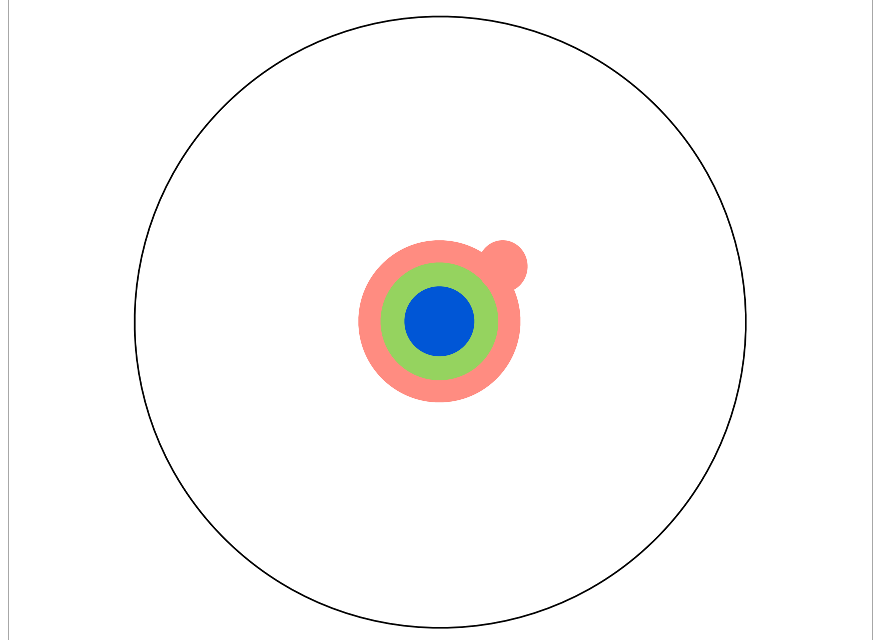
Dans le cadre de cette thèse, et du fait de la nature critique des politiques de contrôle d'accès dans un SI, nous devons garantir que les transformations de modèles n'introduisent pas d'erreurs et ne modifient pas la spécification initiale de la politique. Pour cette raison, les traductions doivent être prouvées, et l'implémentation doit suivre une approche par raffinement formel. De plus, les transformations de modèles effectuées ne doivent pas empêcher la validation, la vérification ou la preuve des modèles générés. Les preuves effectuées sur un modèle traduit doivent aussi pouvoir s'appliquer au modèle initial.

Première partie

Traductions de modèles

Chapitre 2

Traduction d’expressions de processus EB^3 en ASTD



[65] Ph.D. étape 2 – Puis vient le temps du lycée et de notre première spécialisation dans un domaine particulier.

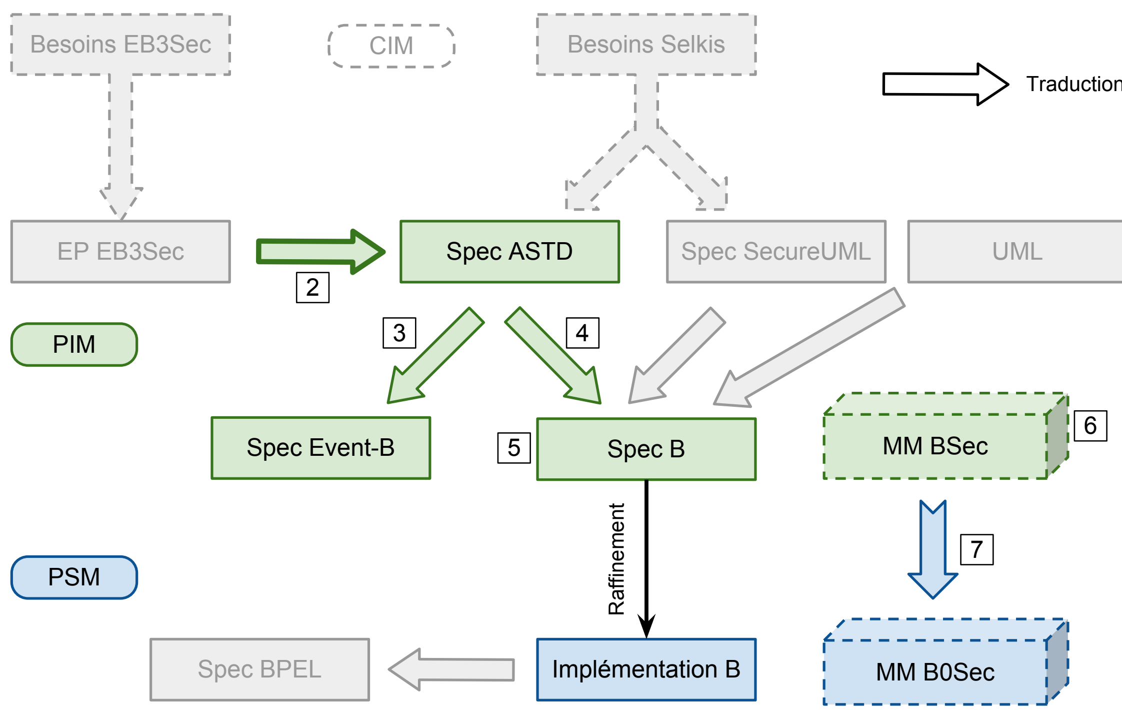


FIGURE 2.1 – Traduction d'expressions de processus EB^{3}SEC vers ASTD

La figure 2.1 indique que nous allons nous intéresser tout d'abord à la traduction d'expressions de processus exprimées avec EB^3 en spécifications ASTD. Afin de pouvoir utiliser la même approche dans le cadre des projets EB^{3}SEC et Selkis, nous avons décidé d'utiliser les ASTD comme langage pivot. Ainsi les étapes suivantes de transformation du modèle de politiques de contrôle d'accès seront communes aux deux approches. De plus, traduire les politiques exprimées à l'aide de EB^{3}SEC permettra de vérifier et de valider la spécification, puisqu'aucun outil ne permet de faire de preuves ou de vérifications de modèle sur EB^{3}SEC.

2.1 Introduction

La traduction d'expressions de processus EB^3 en ASTD se base sur la proximité des sémantiques opérationnelles des deux approches. En effet, les opérateurs des ASTD sont inspirés des opérateurs de EB^3. Dans la section 2.2 nous présentons un algorithme de traduction de EB^3 vers ASTD. Puis des optimisations de cette traduction sont proposées dans

$$
\begin{aligned}
\textbf{member}(mId : \text{MEMBERID}) = \mathsf{Join}(mId)\\
.\,(\quad & (\mid\mid\mid bId \in \text{BOOKID} : \textbf{loan}(mId, bId)^*) \\
& \mid\mid (\mid\mid\mid bId \in \text{BOOKID} : \textbf{reservation}(mId, bId)^*) \\
& \mid\mid \mathsf{DisplayNumberOfLoans}(mId)^* \\
&) \\
.\, & \mathsf{Leave}(mId)
\end{aligned}
$$

FIGURE 2.2 – Expression de processus **member** extraite d'un SI gérant des bibliothèques

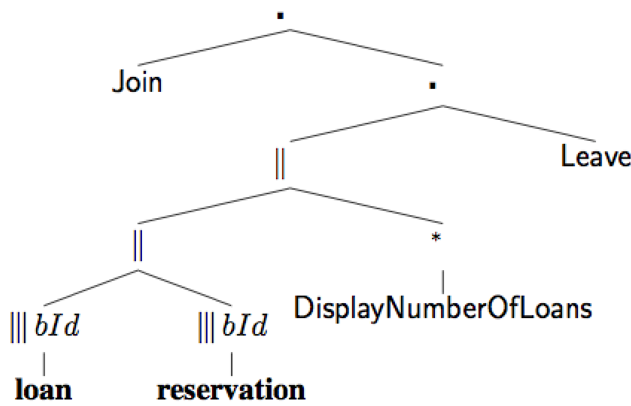


FIGURE 2.3 – AST de l'expression de processus **member**

la section 2.3. L'explication des différences entre une expression de processus EB^{3}SEC et une expression de processus EB^3 est présentée dans la sous-section 2.4.1 afin d'obtenir la traduction d'une expression de processus EB^{3}SEC en ASTDsec.

2.1.1 Notations pour EB^3

Dans les algorithmes qui suivent, nous considérons que les expressions de processus EB^3 sont représentées sous une forme d'arbre de syntaxe abstraite, *Abstract Syntax Tree* (AST). La figure 2.3 représente, sous forme d'AST, l'expression de processus présentée dans la figure 2.2 qui modélise les membres dans le SI gérant une bibliothèque.

De plus, nous utilisons une structure de données pour représenter les expressions de processus EB^3. Cette structure de données est décrite ci-dessous. Les types `Label`, `Params`,

`Variable`, `LabelSet`, `ConstantSet`, `Environment` et `Predicate` ont un nom suffisamment transparent pour en comprendre le sens et ne sont pas détaillés ici par soucis de concision.

```
expressionEB3 =
    Box
  | Lambda
  | Action (Label , Params)
  | Sequence (expressionEB3 , expressionEB3)
  | Kleene (expressionEB3)
  | Choice (expressionEB3 , expressionEB3)
  | Qchoice (Variable , ConstantSet , expressionEB3)
  | Synchro (LabelSet , expressionEB3 , expressionEB3)
  | Qsynchro (LabelSet , Variable
                , ConstantSet , expressionEB3)
  | ProcessCall (Name , Params)
  | Closure (Environment , expressionEB3)
  | Guard (Predicate , expressionEB3)
```

2.1.2 Notations pour les ASTD et les automates

Les ASTD sont formalisés par la notation mathématique introduite dans [33]. En particulier les ASTD de type automate sont hiérarchiques et à transitions gardées. Ils sont modélisés par $< \Sigma, N, \nu, \delta, F, n_0 >$ avec :

- Σ représente l'alphabet des étiquettes des actions du système.
- N est l'ensemble des places de l'automate.
- ν est une fonction indiquant si une place est élémentaire ($elem$) ou hiérarchique.
- δ est la fonction de transition indiquant si chaque transition est locale (*loc*), originaire (*fromSub*) ou à destination (*toSub*) d'une place d'un sous-ASTD automate, et si elle est gardée.
- F est l'ensemble des places finales.
- n_0 est la place initiale.

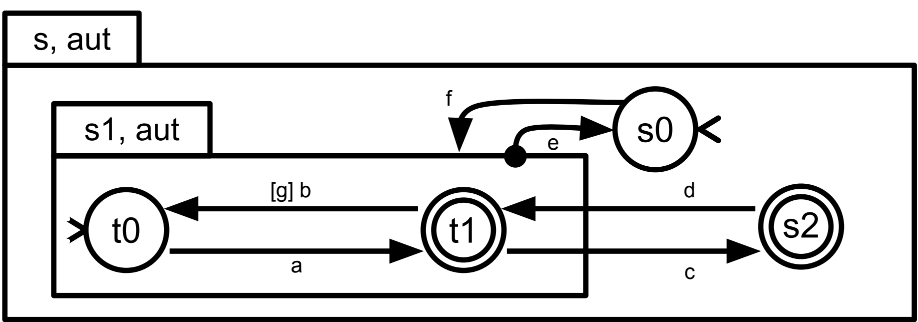


FIGURE 2.4 – Exemple d'un ASTD de type automate

Ainsi, l'ASTD de type automate de la figure 2.4 est représenté par :

$autS =< \Sigma, N, \nu, \delta, F, n_0 >$

$autS.\Sigma = \{c, d, e, f\}$

$autS.N = \{s0, s1, s2\}$

$autS.\nu = \{s0 \mapsto elem, s1 \mapsto autS1, s2 \mapsto elem\}$

$autS.\delta = \{<< loc, s0, s1 >, f, \text{TRUE}, \text{FALSE} >\} \cup$

$\{<< loc, s1, s0 >, e, \text{TRUE}, \text{TRUE} >\} \cup$

$\{<< tosub, s2, s1, t1 >, d, \text{TRUE}, \text{FALSE} >\} \cup$

$\{<< fromsub, s1, t1, s2 >, c, \text{TRUE}, \text{FALSE} >\}$

$autS.F = \{s2\}$

$autS.n_0 = s0$

$autS1 =< \Sigma, N, \nu, \delta, F, n_0 >$

$autS1.\Sigma = \{a, b\}$

$autS1.N = \{t0, t2\}$

$autS1.\nu = \{t0 \mapsto elem, t1 \mapsto elem\}$

$autS1.\delta = \{<< loc, t0, t1 >, a, \text{TRUE}, \text{FALSE} >\} \cup \{<< loc, t1, t0 >, b, g, \text{FALSE} >\}$

$autS1.F = \{t1\}$

$autS1.n_0 = t0$

2.1.3 Notations pour les algorithmes

Les algorithmes présentés utilisent le filtrage par motif (*pattern matching*) comme dans OCaml. Le filtrage est introduit par les mots clé **match** . . . **with**. Le mot clé **match** est suivi de l'expression à analyser ; le mot clé **with** indique l'approche de filtrage qui est appliquée sur l'expression analysée.

2.2 Traduction naïve

Intuitivement, on peut traduire chaque action EB^3 en un ASTD de type automate avec un état initial, un état final et une transition étiquetée par l'action. La sémantique des expressions de processus élémentaires composées uniquement d'une action est alors préservée. On peut ensuite construire par récurrence l'ASTD correspondant à une expression de processus EB^3 puisque pour chaque opérateur de EB^3, il existe un type d'ASTD équivalent. On obtient alors un ASTD de profondeur comparable à la profondeur de l'expression de processus traduite.

2.2.1 Algorithme

L'algorithme 1 présente la traduction naïve d'une expression de processus EB^3 en ASTD. Chaque opérateur EB^3 est traduit par l'ASTD du type correspondant.

Algorithme 1 $operatorTranslation(Exp)$

Description : *Algorithme de traduction d'expressions de processus* EB^3 *en* ASTD.

input *Exp* **:** *Une expression de processus* EB^3.

output : *Une spécification* ASTD.

$operatorTranslation(Exp) =$
begin
match *Exp* ***with***
  $Kleene(e1) \rightarrow a$
    such as
      $a.type = \star$

$a.body = b$

and such as

$b = operatorTranslation(e1)$

$Sequence(e1,e2) \rightarrow a$

such as

$a.type = \mapsto$

$a.fst = b$

$a.snd = c$

and such as

$b = operatorTranslation(e1)$

$c = operatorTranslation(e2)$

$Guard(g,e1) \rightarrow a$

such as

$a.type = \Rightarrow$

$a.gard = g$

$a.body = b$

and such as

$b = operatorTranslation(e1)$

$Choice(e1,e2) \rightarrow a$

such as

$a.type = |$

$a.left = b$

$a.right = c$

and such as

$b = operatorTranslation(e1)$

$c = operatorTranslation(e2)$

$Synchro(D,e1,e2) \rightarrow a$

such as

$a.type = |[]|$
$a.\Delta = D$
$a.left = b$
$a.right = c$
and such as
$b = operatorTranslation(e1)$
$c = operatorTranslation(e2)$

$Qchoice(v, T, e1) \to a$
such as
$a.type = |:$
$a.x = v$
$a.T = T$
$a.body = b$
and such as
$b = operatorTranslation(e1)$

$Qsynchro(D, v, T, e1) \to a$
such as
$a.type = |[]| :$
$a.\Delta = D$
$a.x = v$
$a.T = T$
$a.body = b$
and such as
$b = operatorTranslation(e1)$

$ProcessCall(name, params) \to a$
such as
$a.type = call$
$a.call = name(params)$
end

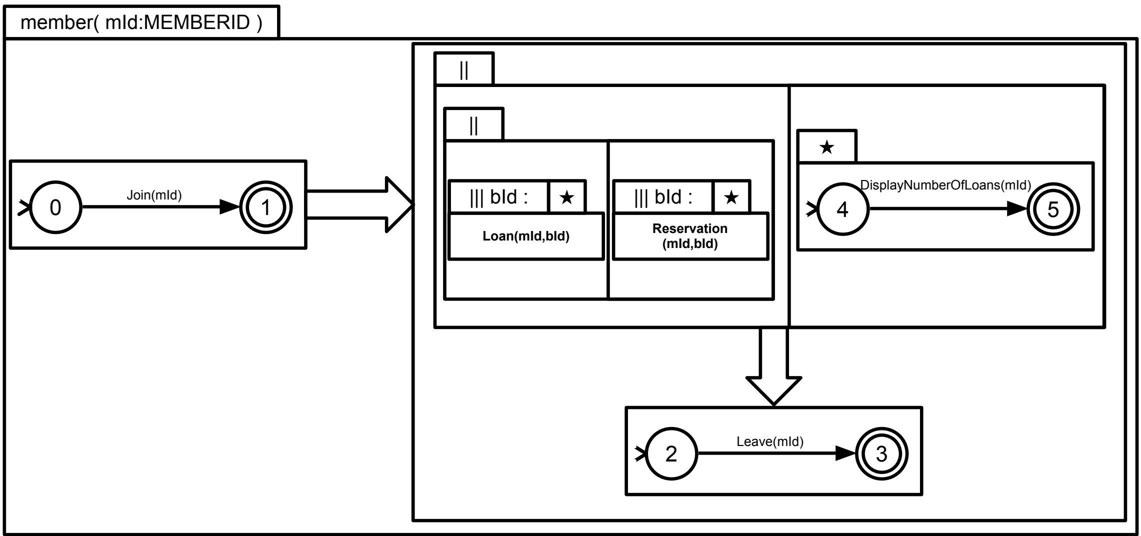


FIGURE 2.5 – ASTD résultant de la traduction naïve de l'expression de processus **member**

2.2.2 Exemple

La traduction en ASTD de l'expression de processus EB^3 de la figure 2.2 utilisant la méthode naïve est présentée dans la figure 2.5. On remarque notamment les ASTD de type séquence depuis et vers des automates qui pourraient être remplacés par des transitions afin de réduire la complexité des ASTD.

2.2.3 Critique

Si ce genre de traduction est acceptable dans le sens où elle conserve la sémantique de l'expression de processus EB^3, elle peut générer des ASTD comportant de nombreuses boîtes imbriquées et des automates à deux états et une transition. Ces ASTD n'utilisent pas toute la puissance d'expression de la notation ASTD. De ce fait, on peut améliorer la traduction en utilisant les ASTD de type automate de manière plus pertinente.

2.3 Optimisations

Nous détaillons dans les sections suivantes quelques-unes des optimisations possibles pour la traduction de EB^3 en ASTD.

2.3.1 Choix et séquences

Les expressions de processus uniquement composées d'actions, de choix ($|$) et de séquences (**.**) peuvent se traduire en ASTD de type automate en suivant les algorithmes classiques de la théorie des automates. On définit le prédicat *isChoicesAndSequences* qui va calculer si une expression de processus n'est composée que de choix, de séquences et d'actions élémentaires. L'algorithme 2 présente ce calcul.

Algorithme 2 $isChoicesAndSequences(Exp)$

Description : *Calcule le prédicat détectant si une expression de processus est composée uniquement de choix, de séquences et d'actions élémentaires.*

input *Exp* **:** *Une expression de processus* EB^3.

output : *Booléen :* TRUE *si l'expression de processus est composée uniquement de choix, de séquences et d'actions élémentaires,* FALSE *sinon.*

$isChoicesAndSequences(Exp) =$
begin
match *Exp* ***with***
Action(_,_) $\rightarrow$ TRUE
Sequence(e1,e2) $\rightarrow$ *isChoicesAndSequences(e1)* $\wedge$ *isChoicesAndSequences(e2)*
Choice(e1,e2) $\rightarrow$ *isChoicesAndSequences(e1)* $\wedge$ *isChoicesAndSequences(e2)*
_ $\rightarrow$ FALSE
end

$eb32astd$, présenté dans l'algorithme 3, teste si l'expression de processus est uniquement composée de choix et de séquences et adapte la traduction de l'expression de processus en conséquence. C'est l'algorithme principal de la traduction.

Algorithme 3 $eb32astd(Exp)$

Description : *Algorithme de traduction d'expressions de processus* EB^3 *en spécification* ASTD.

input *Exp* **:** *Une expression de processus* EB^3.

output : *Une spécification* ASTD.

$eb32astd(Exp) =$
begin
  if *isChoicesAndSequences*(Exp)
  then *choicesAndSequences*$2astd(Exp)$
  else $operatorTranslation(Exp)$
end

Une fois que l'on est certain qu'une expression de processus n'est composée que de choix et de séquences, on peut effectuer la traduction présentée dans l'algorithme 4. Il utilise deux autres algorithmes : (i) $prefixStateNames(a, prefixe)$ qui va préfixer les noms de tous les états de l'automate par le préfixe indiqué. Cela permet d'obtenir des états dont le nom est unique. (ii) $\varepsilon free(a)$ retire toutes les transitions ε sans changer le langage accepté (à l'exception notable de ε lui-même). Cet algorithme est décrit dans la section ε-*transition automata* de [43]. L'application de cet algorithme est nécessaire puisque les transitions ε ne sont pas autorisées dans les ASTD.

Algorithme 4 *choicesAndSequences*$2astd(Exp)$

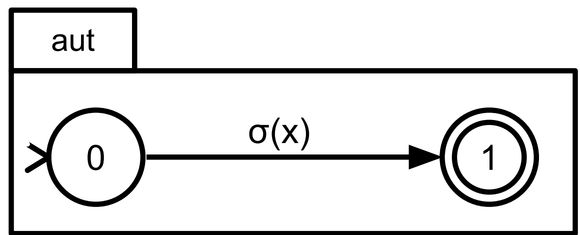
Description : *Traduit une expression de processus* EB^3 *en un automate.*

input *Exp* **:** *Une expression de processus* EB^3 *composée de choix de séquences et actions élémentaires.*

output : *Automate résultant de la traduction.*

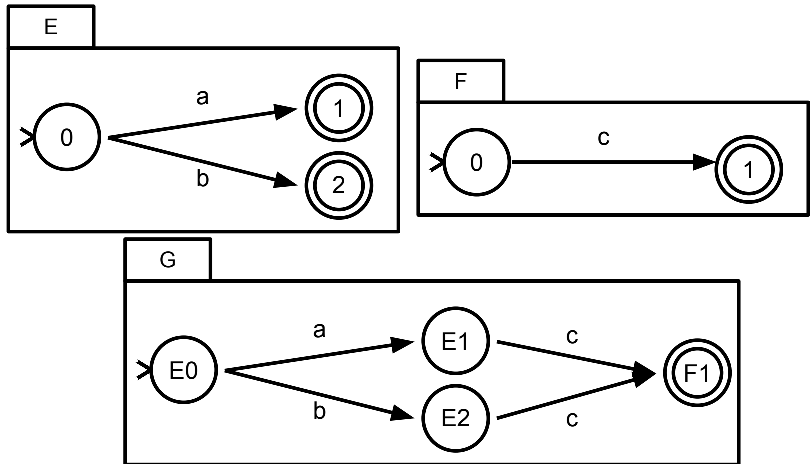
choicesAndSequences$2astd(Exp) =$
begin
match *Exp* ***with***
 Action(σ*,x)* $\rightarrow$ *a*
  such as

$a.\Sigma = \{\sigma\}$
$a.N = \{0, 1\}$
$a.\nu = \{0 \mapsto elem, 1 \mapsto elem\}$
$a.\delta = \{<< loc, 0, 1 >, \sigma(x), \text{TRUE}, \text{FALSE} >\}$
$a.F = \{1\}$
$a.n_0 = 0$



/ Traduction d'une action en un automate avec un état initial (0),*
un état final (1) et une transition étiquetée par l'action $\sigma(x)$ *de 0 à 1. */*

Sequence(e1,e2) $\rightarrow \varepsilon free(c)$
such as
$c.\Sigma = a.\Sigma \cup b.\Sigma \cup \{\varepsilon\}$
$c.N = a.N \cup b.N$
$c.\nu = a.\nu \cup b.\nu$
$c.\delta = a.\delta \cup b.\delta \cup (\bigcup_{s \in a.F} << loc, s, b.n_0 >, \varepsilon, \text{TRUE}, \text{FALSE} >)$
$c.F = b.F$
$c.n_0 = a.n_0$
and such as
$a = prefixStateNames(choicesAndSequences2astd(e1), "a")$
$b = prefixStateNames(choicesAndSequences2astd(e2), "b")$



/ Traduction d'une séquence entre **E**= a | b et **F**= c*
*par un automate **G**. L'automate a pour état initial celui de **E** ,*
*pour états finaux ceux de **F**, une fonction de transition combinant celle de **E** et **F***
*adjointe de transitions ε entre les états finaux de **E** et l'état initial de **F**.*
*L'automate **G** est obtenu en appliquant l'algorithme $\varepsilon free()$ sur le résultat. */*

Choice(e1,e2) $\rightarrow \varepsilon free(c)$

such as

$c.\Sigma = a.\Sigma \cup b.\Sigma \cup \{\varepsilon\}$

$c.N = a.N \cup b.N \cup \{0\}$

$c.\nu = a.\nu \cup b.\nu \cup \{0 \mapsto elem\}$

$c.\delta = a.\delta \cup b.\delta \cup << loc, 0, a.n_0 >, \varepsilon, \text{TRUE}, \text{FALSE} >$
$\cup << loc, 0, b.n_0 >, \varepsilon, \text{TRUE}, \text{FALSE} >$

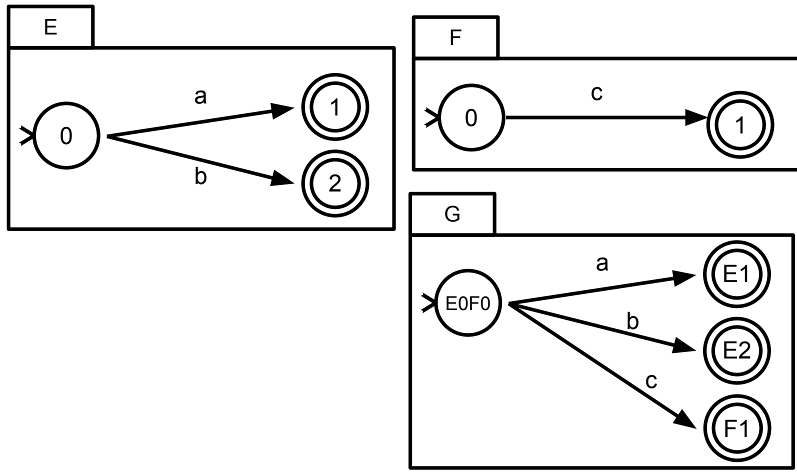
$c.F = a.F \cup b.F$

$c.n_0 = 0$

and such as

$a = prefixStateNames(choicesAndSequences2astd(e1), "a")$

$b = prefixStateNames(choicesAndSequences2astd(e2), "b")$



/ Traduction d'un choix entre **E**= a | b et **F**= c*
*par un automate **G**. L'automate a pour état initial un nouvel état,*
*pour états finaux ceux de **E** et **F**, une fonction de transition combinant celle de **E** et **F***
*adjointe de transitions ε entre le nouvel état et les états initiaux de **E** et de **F**.*
*L'automate **G** est obtenu en appliquant l'algorithme $\varepsilon free()$ sur le résultat. */*
end

Suite à l'étude des traductions générées par ces algorithmes, certaines optimisations sont apparues afin d'utiliser de nouvelles caractéristiques de la notation ASTD. Elles sont détaillées dans les sections suivantes.

2.3.2 Fermeture de Kleene

Il existe des algorithmes de traduction d'expressions régulières composées de choix (|), de séquences (**.**) et de fermetures de Kleene (*) en automate sans ε-transitions [43]. Cependant, de tels algorithmes peuvent considérablement modifier la structure initiale d'un ASTD. C'est pourquoi nous avons choisi de ne pas utiliser ces algorithmes. La traduction obtenue pourrait ne plus être compréhensible aisément par un humain.

Une amélioration de l'algorithme de traduction peut cependant être faite afin d'éviter l'emploi d'un opérateur de Kleene dans certains cas. Quand l'opérateur de Kleene s'applique sur une action, la traduction doit alors donner un automate composé d'un état, à la fois initial et final, et d'une transition cyclique, comme illustré dans l'algorithme 5.

Algorithme 5 *kleeneAction2astd*(Exp)

Description : *Traduit une expression de processus* EB^3 *composée d'une action et d'une fermeture de Kleene en automate.*

input *Exp* **:** *Une expression de processus* EB^3 *composée d'une action et d'une fermeture de Kleene.*

output : *Automate résultant de la traduction.*

$kleeneAction2astd(Exp) =$
begin
match *Exp* ***with***
Kleene(Action(σ*,x))* $\rightarrow a$
such as
$a.\Sigma = \{\sigma\}$
$a.N = \{0\}$
$a.\nu = \{0 \mapsto elem\}$
$a.\delta = \{<< loc, 0, 0 >, \sigma(x), \text{TRUE}, \text{FALSE} >\}$
$a.F = \{0\}$
$a.n_0 = 0$
end

La différence entre la traduction naïve et la traduction optimisée pour une fermeture de Kleene sur une action élémentaire est présentée dans la figure 2.6.

2.3.3 Gardes

Certaines gardes ne s'appliquent que sur une action élémentaire. Afin d'utiliser le fait que les transitions des ASTD peuvent être gardées, ces expressions de processus sont traduites comme illustré dans l'algorithme 6.

Algorithme 6 *guardAction2astd*(Exp)

Description : *Traduit une expression de processus* EB^3 *composée d'une action et d'une garde en automate.*

input *Exp* **:** *Une expression de processus* EB^3 *composée d'une action et d'une garde.*

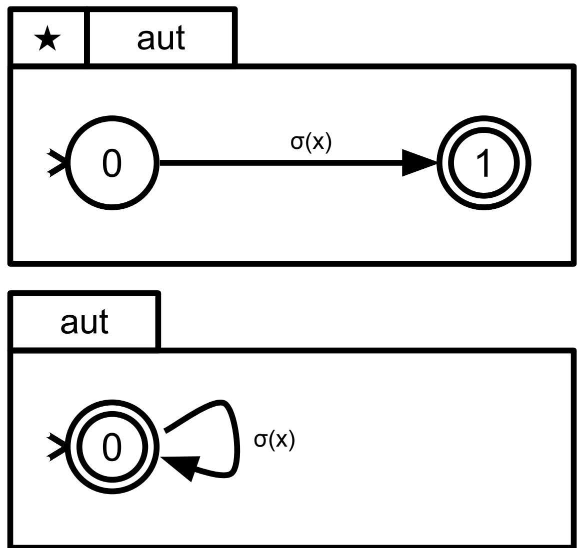


FIGURE 2.6 – En haut, la version non optimisée de la traduction de $\sigma(x)^*$. En bas, la version optimisée

output : *Automate résultant de la traduction.*

$guardAction2astd(Exp) =$

begin

match *Exp* ***with***

Guard(g,Action(σ,x)) $\rightarrow a$

with

$a.\Sigma = \{\sigma\}$

$a.N = \{0,1\}$

$a.\nu = \{0 \mapsto elem, 1 \mapsto elem\}$

$a.\delta = \{<< loc, 0, 1 >, \sigma(x), g, \text{FALSE} >\}$

$a.F = \{1\}$

$a.n_0 = 0$

end

La différence entre la traduction naïve et la traduction optimisée pour une garde sur une action élémentaire est présentée dans la figure 2.7.

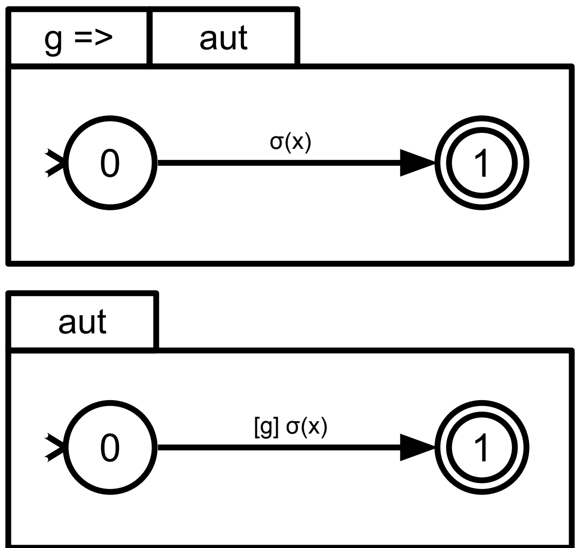


FIGURE 2.7 – En haut, la version non optimisée de la traduction de $g \Rightarrow \sigma(x)$. En bas, la version optimisée

2.3.4 Patron PMC

Le patron PMC est un patron important en EB³. PMC signifie Producteur, Modificateur, Consommateur. Il s'exprime sous la forme $\mathsf{p}(\,)\,.\,\mathbf{M}(\,)^*\,.\,\mathsf{c}(\,)$ où $\mathsf{p}(\,)$ et $\mathsf{c}(\,)$ sont deux actions élémentaires et $\mathbf{M}(\,)$ une expression de processus quelconque. Ce patron est détaillé dans [38]. L'ASTD résultant de la traduction est présenté en figure 2.8. Il utilise une transition finale, représentée par une transition avec un point à son origine. Cela signifie que la transition ne peut s'activer que quand l'état de l'ASTD source est final.

2.3.5 Itération de séquences

Lorsqu'une fermeture de Kleene s'applique sur une séquence d'actions, une traduction plus simple que celle proposée par l'algorithme peut s'appliquer. En effet, on peut créer un automate dont l'état initial est également final, et faire en sorte que la dernière transition pointe vers cet état. Par exemple, la figure 2.9 présente l'ASTD correspondant à l'expression de processus $(\mathsf{x}_1\,.\,\ldots\,.\,\mathsf{x}_n)^*$.

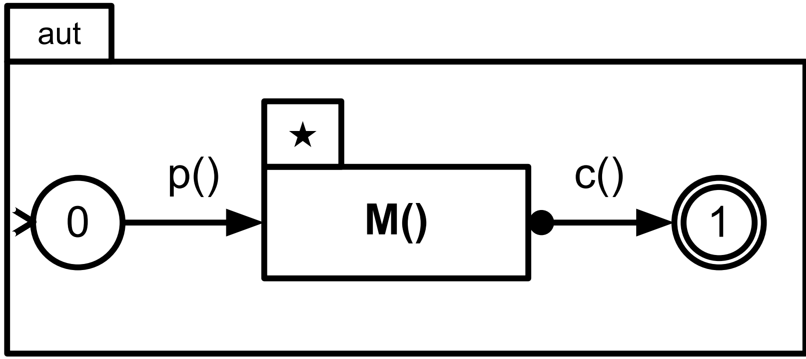


FIGURE 2.8 – Traduction du patron PMC

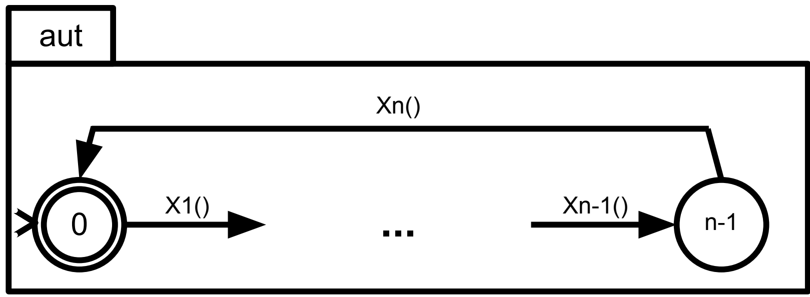


FIGURE 2.9 – Traduction de l'itération de séquences

2.4 Conclusion

2.4.1 Traduction de EB^{3}SEC vers ASTDsec

La principale différence entre une action élémentaire d'une expression de processus EB^3 ou d'un ASTD et celle d'une expression de processus EB^{3}SEC ou d'un ASTDsec se situe au niveau des paramètres. En effet, dans le cadre de EB^{3}SEC, certains paramètres dits de sécurité jouent un rôle particulier et sont donc séparés des paramètres dits fonctionnels que l'on trouve dans les actions EB^3 et ASTD.

Deux actions ayant la même étiquette σ sont présentées dans la figure 2.10. La première est une action élémentaire EB^3, la seconde une action sécurisée EB^{3}SEC avec les même paramètres fonctionnels x et y mais également des paramètres dits de sécurité u et r correspondant à l'utilisateur et à son rôle lors de l'exécution de l'action.

$$\sigma(x,y)$$

$$< u, r, \sigma(x,y) >$$

FIGURE 2.10 – Comparaison d'actions élémentaires EB^3 et EB^{3}SEC

Afin de profiter de la traduction de EB^3 vers ASTD pour les expressions EB^{3}SEC, une petite modification des algorithmes doit être effectuée. La seule différence apparaît dans la traduction des actions élémentaires. Cette traduction doit prendre en compte les paramètres de sécurité. On obtient ainsi directement une spécification ASTDsec depuis une expression de processus EB^{3}SEC.

2.4.2 Bilan

Nous avons présenté une traduction de EB^3 vers ASTD qui peut être utilisée pour traduire une expression EB^{3}SEC en ASTDsec. Cette traduction est optimisée pour des patrons fréquents dans les SI. D'autres améliorations pourraient être apportées afin d'utiliser plus de fonctionnalités des ASTD comme les transitions *tosub* ou *fromsub*, les états historiques ou les états hiérarchiques.

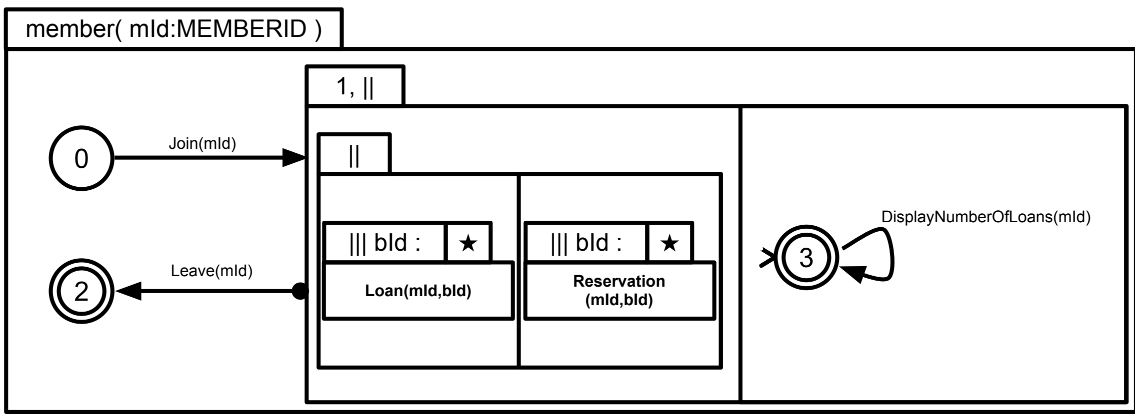


FIGURE 2.11 – ASTD résultant de la traduction optimisée de l'expression de processus **member**

La traduction en ASTD de l'expression de processus EB^3 de la figure 2.2 utilisant la méthode optimisée est présentée dans la figure 2.11.

2.4.3 Travaux futurs

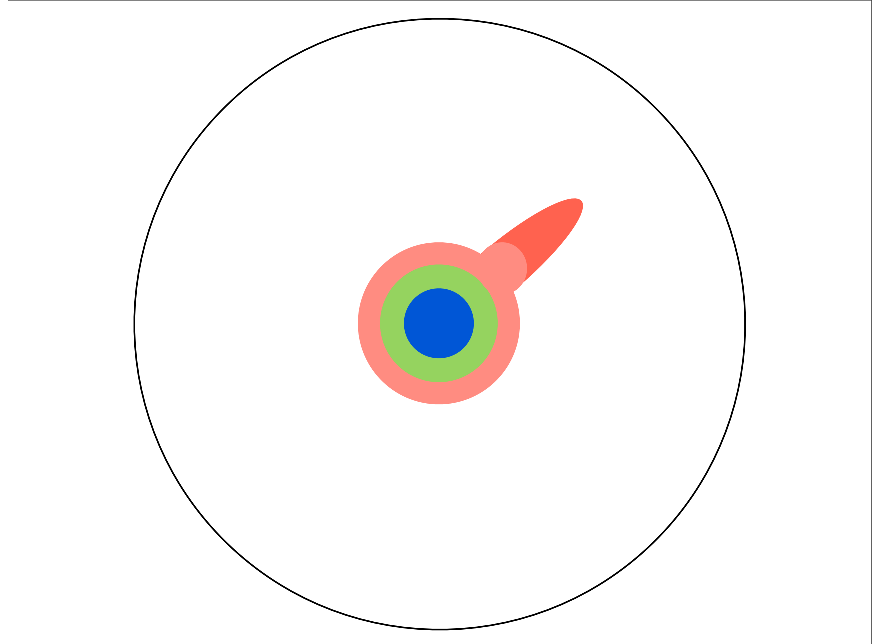
Une implémentation de ces algorithmes de traduction peut être développée en se basant sur *i*ASTD [86] et OCamlPAI, respectivement un interpréteur pour les ASTD et un interpréteur EB^3. Ils ont été tous deux développés en utilisant le langage OCaml. L'outillage des algorithmes de traduction présentés ici permettrait d'obtenir une spécification ASTD plus simplement qu'actuellement, car la notation ASCII utilisée par *i*ASTD est complexe et peu lisible. Lors d'une phase de spécification, on pourrait alors écrire le comportement attendu d'un système en utilisant EB^3 puis on traduirait automatiquement la spécification qui serait alors directement interprétable par *i*ASTD.

De plus, il faut prouver que la traduction ASTD respecte bien la sémantique de la spécification EB^3. Pour cela, on pourra se baser sur les règles d'inférence définissant la sémantique d'exécution de EB^3 et celle d'ASTD. Ces règles étant similaires, il est fort probable que les preuves de traduction soient aisées sinon triviales.

Une autre perspective est la traduction d'ASTD vers EB^3. Cette traduction ne peut cependant pas être complète puisque certaines fonctionnalités des ASTD n'ont pas d'équivalent en EB^3, comme les états historiques.

Chapitre 3

Règles de traduction systématique des ASTD en Event-B



[65] Ph.D. étape 3 – On entre ensuite à l'université pour une licence puis un master où l'on se spécialise encore plus et où l'on accède à des connaissances bien plus pointues qu'auparavant.

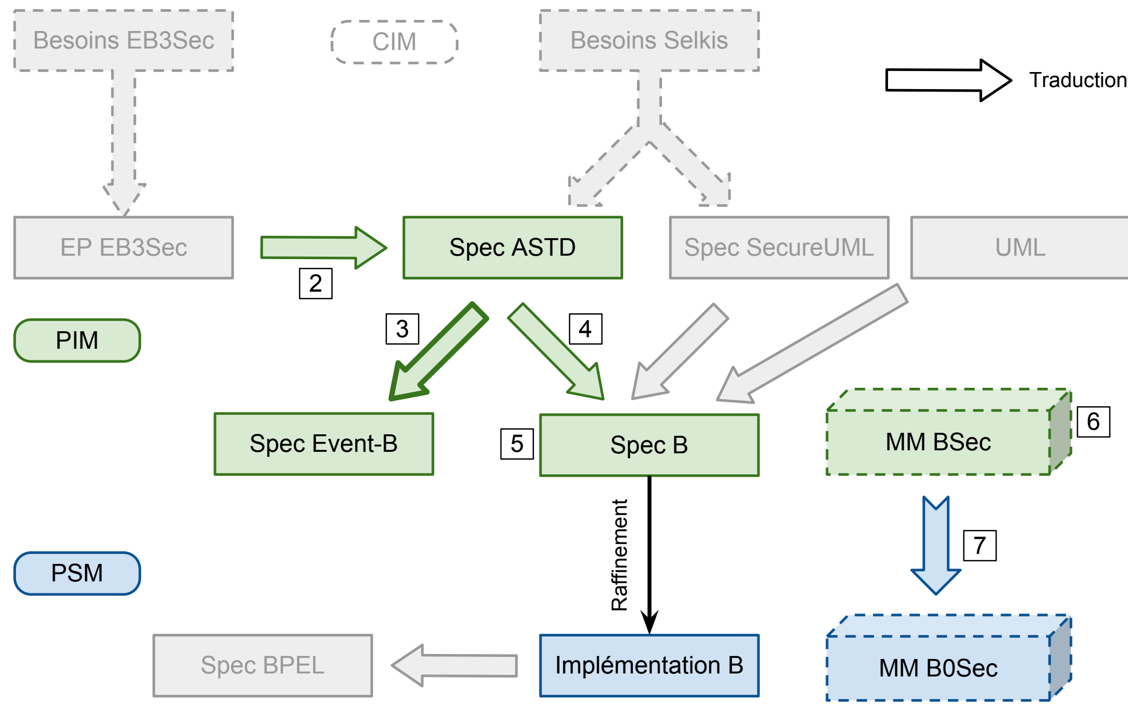


FIGURE 3.1 – Traduction de spécification ASTD en Event-B

Nous nous intéressons maintenant, comme indiqué au point 3 de la figure 3.1, à la traduction de spécifications ASTD en Event-B afin d'accéder aux outils de Event-B pour vérifier et valider les spécifications de politiques de contrôle d'accès. Chronologiquement, ces règles ont été développées avant celles de la traduction ASTD vers B présentées dans le chapitre suivant. Cette traduction devait initialement être prouvée. Cependant, la traduction a généré des évènements surnuméraires qui ne correspondent à aucune transition de l'ASTD initial. Cette particularité est due à l'absence de structure du type **IF...THEN...ELSE** en Event-B. De ce fait, bien qu'intuitivement correcte, cette traduction n'a jamais été prouvée, et les travaux subséquent se sont portés uniquement sur la traduction vers B. Néanmoins, la traduction a sa place dans cette thèse car elle a permis de mieux cerner les problématiques de la traduction des ASTD vers B, et la nécessité de bien formaliser les règles de traduction. Une application de ces règles de traduction sur une étude de cas est disponible en annexe A.

Résumé

Cet article présente un ensemble de règles de traduction pour générer des spécifications Event-B à partir de langages de spécification basés sur les algèbres de processus comme ASTD. Illustré par une étude de cas, il détaille les règles et le processus de traduction. L'objectif de cette traduction est de pouvoir utiliser l'outil de développement et de preuves de spécifications Event-B, Rodin pour animer, valider et vérifier la spécification traduite.

Commentaires

Ma contribution au sein de cet article consiste dans le développement des règles de traduction des ASTD vers Event-B ; de l'étude de cas ; des tests de la traduction ; de l'animation de l'étude de cas, ainsi que de la rédaction majoritaire du texte. Cet article a été soumis et accepté à la huitième édition de la conférence internationale *integrated Formal Methods* (iFM 2010) ayant eu lieu à Nancy en France du 11 au 14 octobre 2010. Il a fait l'objet d'une publication dans la série *Lecture Notes of Computer Science* de l'éditeur Springer Berlin / Heidelberg dans le volume 6396.

Systematic translation rules from ASTD to Event-B

Jérémy Milhau [1,2], Marc Frappier [1], Frédéric Gervais [2], Régine Laleau [2]

1 : GRIL, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke QC, Canada
{Jeremy.Milhau,Marc.Frappier}@usherbrooke.ca

2 : LACL, Université Paris-Est, IUT Sénart Fontainebleau, Département Informatique Route Forestière Hurtault, 77300 Fontainebleau, France {Frederic.Gervais,Laleau}@u-pec.fr

Abstract This article presents a set of translation rules to generate Event-B machines from process-algebra based specification languages such as ASTD. Illustrated by a case study, it details the rules and the process of the translation. The ultimate goal of this systematic translation is to take advantage of Rodin, the Event-B platform to perform proofs, animation and model-checking over the translated specification.

Keywords Process algebra, Translation rules, Systematic translation, ASTD, Event-B.

3.1 Introduction

Information Systems (IS) are taking an increasingly important place in today's organizations. As computer programs connected to databases and other systems, they induce increasing costs for their development. Indeed, with the importance of the Internet and their high computer market penetration, IS have become the de-facto standard for managing most of the aspects of a company strategy. In the context of IS, formal methods can help improving the reliability, security and coherence of the system and its specification. The APIS (Automated Production of Information system) project [31] offers a way to specify and generate code, graphical user interfaces, databases, transactions and error messages of such systems, by using a process algebra-based specification language. However, process algebra, despite their formal aspect, are not as easily understandable as semi-formal graphical notations, such as UML [77]. In order to address this issue, a formal notation combining graphical elements and process algebra was introduced: Algebraic State Transition Diagrams (ASTD) [32]. Using ASTD, one can specify the behavior of an IS. The

interpreter *i*ASTD [86] can efficiently execute ASTD specifications. However, there is no tool allowing proof of invariants or property check over an ASTD specification. This paper aims to define systematic translation rules from an ASTD specification to Event-B [4] in order to model check or prove properties using tools of the RODIN platform [2]. Moreover, translation results will allow to bridge other process algebras (like EB^3 [38] or CSP [44]) with Event-B as they share a similar semantics with ASTD. Event-B is first introduced to the reader in Section 3.2. An overview of ASTD and a case study will be then presented. This case study will help readers unfamiliar with ASTD to discover the formalism in Section 3.3. The Event-B machine resulting from translation rules applied to this case study will be described as well as rules and relevant steps of translation in Section 3.4. Finally, future work and evolution perspectives will be presented.

3.2 Event-B Background

Event-B [4] is an evolution of the B method [3] allowing to model discrete systems using a formal mathematical notation. The modeling process usually follows several refinement steps, starting from an abstract model to a more concrete one in the next step. Event-B specifications are built using two elements: *context* and *machine*. A *context* describes the static part of an Event-B specification. It consists of declarations of *constants* and *sets*. *Axioms*, which describe types and properties of constants and sets, are also included in the context. A *machine* is the dynamic part of an Event-B specification. It has a state consisting of several *variables* that are first initialized. Then *events* can be executed to modify the state. An event can be executed if it is enabled, *i.e.* all the conditions prior to its execution hold. Theses conditions are named *guards*. Among all enabled events, only one is executed. In this case, substitutions, called *actions*, are applied over variables. All actions are applied simultaneously, meaning that an event is atomic. The state resulting from the execution of the event is the new state of the machine, enabling and disabling events. Alongside the execution of events, *invariants* must hold. An invariant is a property of the system written using a first-order predicate on the state variables. In order to ensure that invariants hold, *proofs* are performed over the specification.

3.3 ASTD Background

ASTD is a graphical notation linked to a formal semantics allowing to specify systems such as IS. An ASTD defines a set of traces of actions accepted by the system. ASTD actions correspond to events in Event-B. Event-B actions and substitutions, as they modify the state of an Event-B machine, can be binded to the change of state in ASTD. The ASTD notation is based on operators from the EB^3 [38] method and was introduced as an extension of Harel's Statecharts [41]. An ASTD is built from transitions, denoting action labels and parameters, and states that can be elementary (as in automata) or ASTD themselves. Each ASTD has a type associated to a formal semantics. This type can be automata, sequence, choice, Kleene closure, synchronization over a set of action labels, choice or interleaving quantification, guard and ASTD call. One of ASTD most important features is to allow parametrized instances and quantifications, aspects missing from original Statecharts. An ASTD can also refer to attributes, which are defined as recursive functions on traces accepted by the ASTD, as in the EB^3 method. Such a recursive function compares the last action of the trace and maps each possible action to a value of the attribute it is defining. Computing this value may imply to call the function again on the remaining of the trace.

3.3.1 ASTD Operators

Several operators, or ASTD types, are used to specify an IS. We detail them in the following paragraphs. Operators will be further illustrated in Section 3.3.2 with the introduction of a case study.

Automata In an ASTD specification, one can describe a system using hierarchical states automata with guarded transitions. Each automata state is either elementary or another ASTD of whichever type. Transitions can be on states of the same depth, or go up or down of one level of depth. A transition decorated by a bullet (•) is called a final transition. A final transition is enabled when the source state is final. As in Statecharts, an history state allows the current state of an automata ASTD to be saved before leaving it in order to reuse it later.

Sequence A sequence is applied to two ASTD. It implies that the left hand side ASTD will be executed and will reach a final state before the right hand side ASTD can start. There is no immediate equivalent of this operator in Harel's Statecharts, but its behavior can be reproduced with guards and final transitions. A sequence ASTD is noted with a double arrow $\Rightarrow$.

Choice A choice, noted $|$, allows the execution of only one of its operands, like a choice in regular expressions or in process algebras. The choice of the ASTD to execute is made on the first action executed. After the execution of the first action, the chosen ASTD is kept until it terminates its execution. If both operands of a choice ASTD can execute the first action, then a nondeterministic choice is made between the two ASTD. The behavior of a choice ASTD can be modeled in Statecharts using internal transition from an initial state, in a similar way to automata theory with ε transitions.

Kleene Closure As in regular expressions, a Kleene closure ASTD noted $*$, allows its operand to be executed zero, one or several times. When the state of its operand is final, a new iteration can start. There is no similar operator in Statecharts, but the same behavior can be reproduced with guards and transitions.

Synchronization Over a Set of Action Labels As the name suggests, this operator allows the definition of a set of actions that both operands must execute at the same time. It is similar to Roscoe's CSP parallel operator $\underset{X}{\|}$. There are some similarities with AND states of Statecharts and synchronization ASTD. A synchronization over the set of actions Δ is noted $|[\Delta]|$. We derive two often used operators from synchronization : interleaving, noted $|||$, is the synchronization over an empty set ; parallel, noted $\|$, synchronizes ASTD over the set of common actions of its operands, like Hoare's CSP $\|$.

Quantified Interleaving A quantified interleaving models the behavior of a set of concurrent ASTD. It sets up a quantification set that will define the number of instances that can be executed and a variable that can take a value inside the quantification set. Each instance of the quantification is linked to a single value, two different instances have two different values. This feature lacks in Statecharts, as we have to express distinctly each instance

behavior, but was proposed as an extension and named “parametrized-and" state by Harel. A quantified interleaving of variable x over the set T is noted $|||\ x : \mathrm{T}$.

Quantified Choice A quantified choice, noted $|\ x : \mathrm{T}$, lets model that only one instance inside a set will be executed. Once the choice is made, no more instances can be executed. As in quantified synchronization, the instance is linked to one value of a variable in the quantification set. An extension of Statecharts named a similar feature “parametrized-or" state.

Guard Usually, guards are applied to transitions. With the guard ASTD, one can forbid the execution of an entire ASTD until a condition holds. The predicate of a guard can use variables from quantifications and attributes. A predicate $P(x)$ guarding an ASTD is noted $\Longrightarrow P(x)$

ASTD call An ASTD call simply links to other parts of the specification using the name of another ASTD. The same ASTD can be called several times, in different locations of the specification. It allows the designer to reuse ASTD in the same specification and helps synchronize processes. An ASTD call is made by writing the name of the ASTD called and its parameters (if any).

3.3.2 An ASTD Case Study

In order to present features and expressiveness of the ASTD notation to the reader, Fig. 3.2 introduces the case study that will be used throughout this paper. This ASTD models an information system designed to manage complaints of customers in a company. In this system, each complaint is issued from a customer relatively to a department. This example is inspired from [89]. The **main** ASTD, whose type is a synchronization over common actions, describes the system as a parallel execution of interleaved customers and departments processes. The IS lifecycle of a given customer is described by the parametrized ASTD **customer**(u), the same applies for the description of the company departments in the ASTD **department**(d). In the initial state, a customer or a department must be created. Then complaints regarding these entities can be issued. This is described using an ASTD call. The final transition means that the event can be executed if the source state is final.

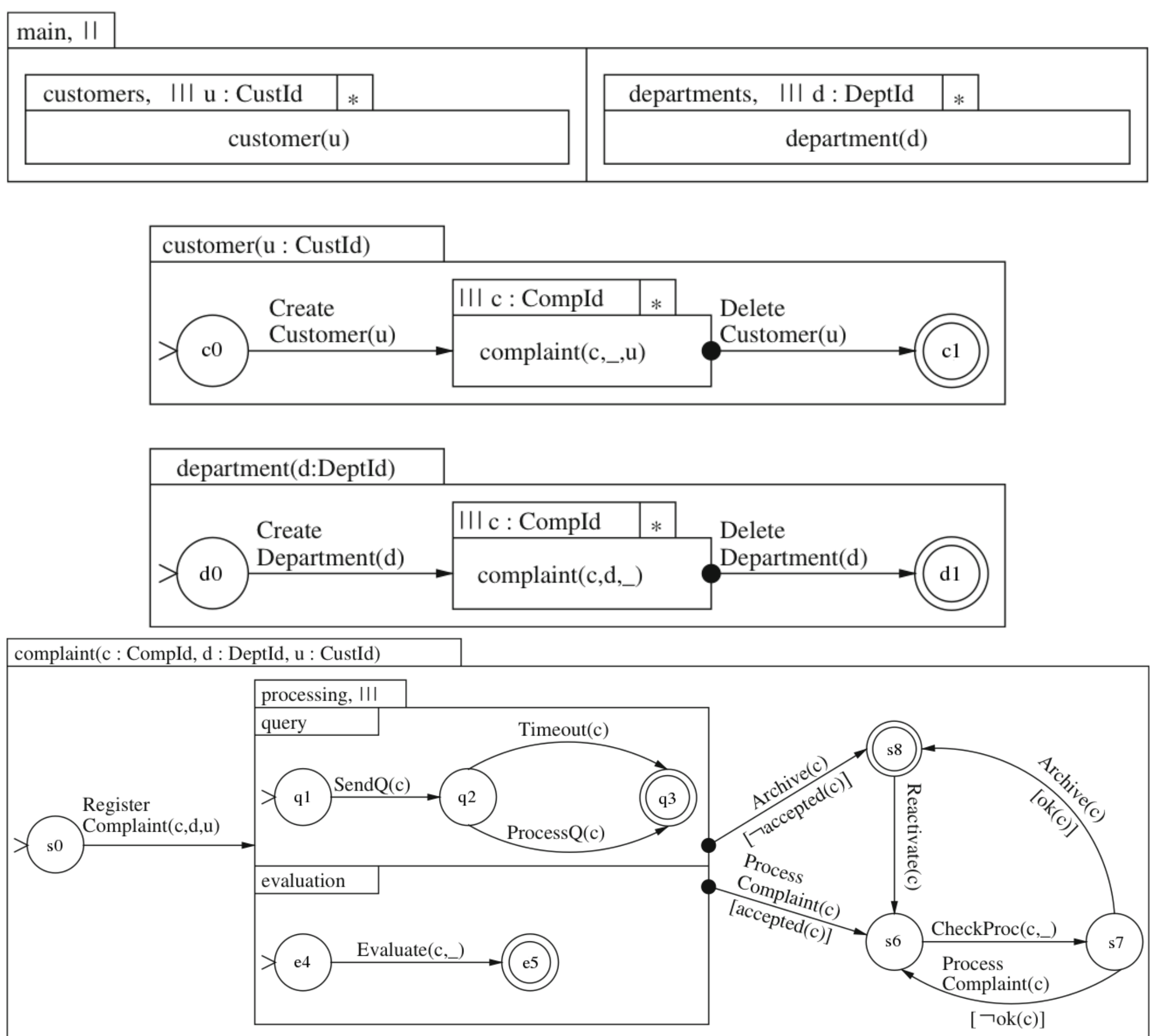


Figure 3.2: An ASTD specification describing a complaint management system

In our case, in order to delete a customer or a department, any related complaints must be closed. Finally, the ASTD describing the checking and processing of a complaint c issued by customer c about department d is given by **complaint**(c, d, u). After registering a complaint in the system, it must be evaluated by the company and a questionnaire is sent to the customer in order to detail his/her complaint. The specification takes into account the possibility that the customer does not answer the questionnaire with the Timeout(c) event. Then, if the complaint is accepted and the questionnaire received or timed out, a check is performed. In case of refusal of the complaint, it is archived, but it can be reactivated later. The only final state is state $s8$, meaning that the complaint was archived (solved or not). In this specification, no attribute modification is performed. An ASTD only describes traces and has no consequences on updates to be performed against IS data, such as attributes that are stored in databases. However, an ASTD can access attribute values to use them in guards, as shown in both Archive(c) actions.

3.3.3 Motivations

ASTD are not the only way to specify IS behavior. The UML-B [80] method introduces a behavior specification in the form of a Statechart. Using Statecharts, it is easy to describe an ordered sequence of actions whereas using B, it is easier to model interleaving events. A systematic translation of Statecharts into B machines is proposed by [87]. Compared to Statecharts, ASTD offer additional operators to combine ASTD in sequence, iteration, choice and synchronisation. When a UML-B specification models a system, it can only describe the life-cycle of a single instance of a class whereas ASTD specification models the behavior of all instances of all classes of the system. A new version of UML-B [82] introduces the possibility to refine class and Statecharts as part of the modeling process, and can translate it into Event-B. The UML-B approach can describe the evolution of entity attributes using B substitutions, a feature that ASTD lacks. csp2B [11] provides better proofs (on the B machine) and model checking (on the CSP side) tools than Statecharts but lacks the visual representation of the specification given by UML Statecharts. It is also limited to a subset of CSP specifications, where the quantified interleaving operator must not be nested. ASTD aims to be a compromise in both visual and synchronization aspects. On the other hand, ASTD lacks proofs and model checking allowed by the B side of UML-

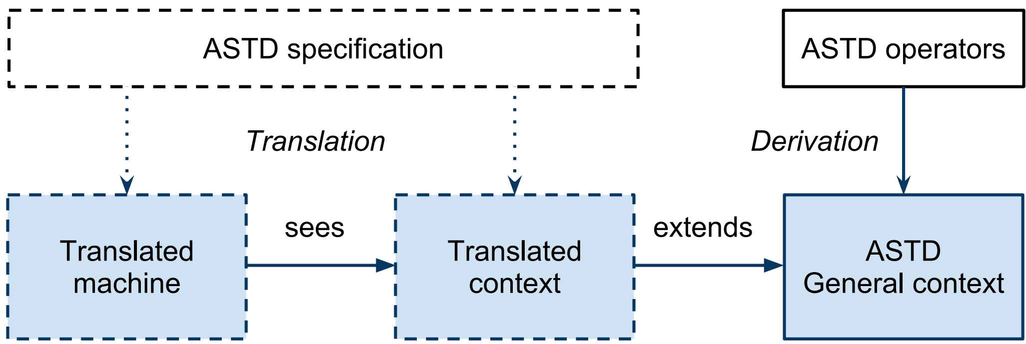


Figure 3.3: The architecture resulting from the translation process

B and csp2B approaches. In order to answer this issue, a systematic translation of ASTD specifications into Event-B is proposed.

The choice between classical B and Event-B was made at an early stage by comparing tools and momentum of both methods. It appears that community efforts and tool development are currently focused on Event-B. Despite the fact that classical B offers some convenient notation such as IF / THEN statements or operation calls, Event-B appeared as a good compromise for our efforts. Classical B translation rules inspired by Event-B rules might be written.

3.4 Translation

Translation from ASTD to Event-B is achieved in several steps. Fig. 3.3 presents the architecture of the translation process. A context derived from ASTD operators introduces constants and sets needed to code their semantics. This context is the same in all translations and is described by Table 3.1. It codes elements from the semantics of all types of ASTD except automata, and is inspired of mathematical definition of ASTD semantics. Constants, sets and axioms defined in this context may be re-used in other part of the Event-B translation, hence this context is extended by a translation specific context. Automata states are translated into such a specific context since automata states depend on the ASTD specification to translate. For each ASTD, a variable and an invariant corresponding to its type are created. The invariant associates the variable to the set of values it can take, as defined in

ASTD state	State domain	Initial State
choice	$State \in \{$ `none`, `first`, `second` $\}$	`none`
sequence	$State \in \{$ `left`, `right` $\}$	`left`
Kleene closure	$State \in \{$ `neverExecuted` , `started` $\}$	`neverExecuted`
synchronization	-	-
quantified choice	$State \in \{$ `notMade`, `made` $\} \nrightarrow$ QUANTIFICATIONSET	`notMade` $\mapsto 0$
quantified synch	$State \in$ QUANTIFICATIONSET $\rightarrow$ STATESET	initial for all
guard	$State \in \{$`checked`, `notChecked` $\}$	`notChecked`
ASTD call	-	-

Table 3.1: Event-B representation of ASTD states

both contexts. In the following sections, we provide translation rules for each ASTD type, generating appropriate contexts and machines.

3.4.1 Automata

The first part of automata translation concerns the static part, the context. Several elements are introduced in the context: states, initial states, final states and transition functions.

States States from automata ASTD are represented as constants and grouped into state sets in order to facilitate later use. Even hierarchical states are represented by a constant.

Initial States Since an ASTD can be reset by the execution of a Kleene closure, initial states are defined as separate constants. They are also useful in the initialisation event of the machine generated in next step of our translation.

Final Predicates A final predicate is a function taking a state as argument and returning TRUE or FALSE depending if the state is final or not. The number of arguments depends on the type of the ASTD. This predicate is useful in the case of final transitions, sequences or Kleene closures, when transitions are activated if, and only if, a state is final. Hence, a final predicate is written for each ASTD type in the context common to all translations.

Transition Functions A transition function for each action label is generated. It takes as argument the current state of an automata ASTD and returns the resulting state. Transition functions are deterministic and partial.

The generated context for our case study defines 40 constants, 5 sets and 29 axioms. It is not presented here for the sake of conciseness. Then, for the dynamic part, for each distinct action label in the translated automata ASTD, a single event will be produced. If the action has a guard, a WHEN clause *i.e.* a guard, is generated. If the ASTD action has arguments (in the case of quantified variable for instance), an ANY clause is built accordingly and a guard specifying a type for the variable is added. Then a guard testing that the execution of the action is allowed *i.e.* the current state is in the domain of the transition function of the event. The modification of the state is applied by generating a THEN substitution.

Translation rules for automata ASTD are presented in Table 3.2. When a transition, an initial state or a final state is found, the first rule applies. In the case of a final transition, the second rule then applies. In the second pattern translation, the guard numbered `g1` of Table 3.2 is added to event e that was generated by applying first rule. In our case study, the second rule is applied for the ProcessComplaint(c) action. The guard added in this case is described by guard `grdAutomata`.

`grdAutomata`: $isFinalProcessing(isFinalQuery(StateQuery(c)) \mapsto$
$isFinalEvaluate(StateEval(c))) = \text{TRUE}$

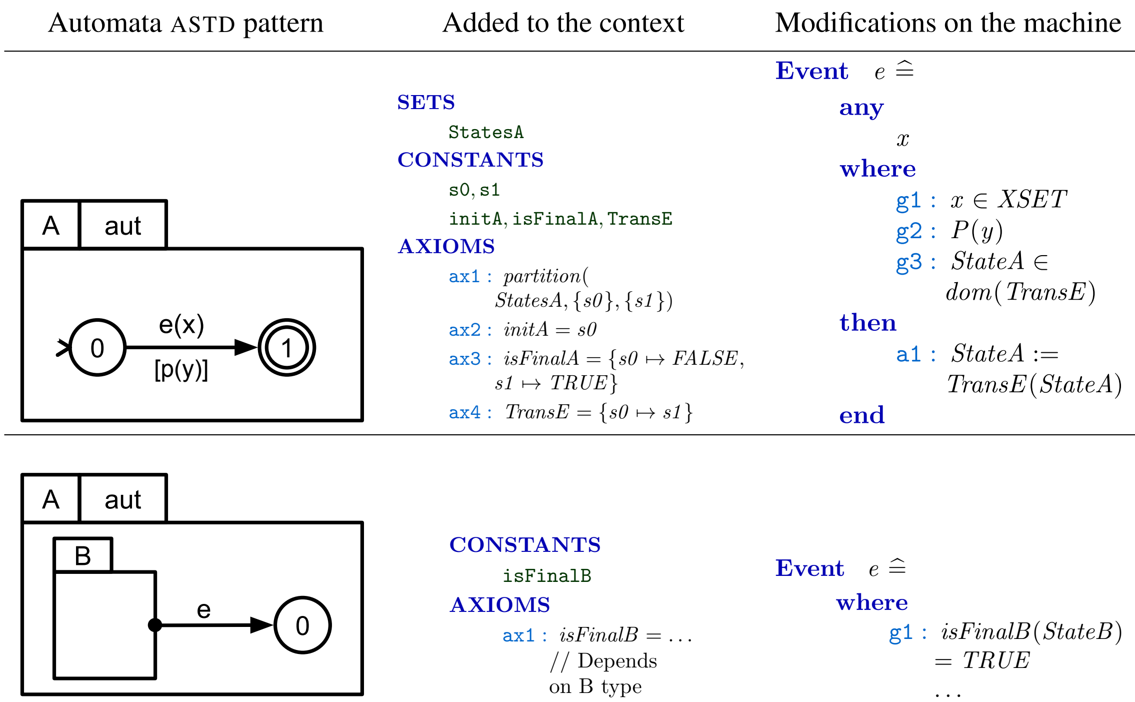
Constants $isFinalX$ and $StateX$ refer to ASTD **X** in Fig. 3.2. An interleaved state is final if, and only if, both of its operand states are final. For this reason, guard `grdAutomata` checks if both states of **Query** and **Evaluation** ASTD are final. A pair (x, y) is noted $x \mapsto y$ in Event-B.

The action CreateCustomer(u) is translated into the event described below. $grd1$ describes the set in which the parameter u can take its value. $grd2$ verifies that a customer is in a state of the domain of transition function $TransCreateCustomer$. $act1$ describes the state update for action CreateCustomer(u): it only modifies the state of customer u according to the transition function $TransCreateCustomer$.

Event $CreateCustomer \mathrel{\widehat{=}}$

any

u



Automata ASTD pattern	Added to the context	Modifications on the machine

A aut; states 0 → 1 on e(x) [p(y)]	**SETS**
	`StatesA`
	CONSTANTS
	`s0, s1`
	`initA, isFinalA, TransE`
	AXIOMS
	ax1 : $partition(StatesA, \{s0\}, \{s1\})$
	ax2 : $initA = s0$
	ax3 : $isFinalA = \{s0 \mapsto FALSE, s1 \mapsto TRUE\}$
	ax4 : $TransE = \{s0 \mapsto s1\}$
		Event $e \mathrel{\widehat{=}}$
		any
		x
		where
		g1 : $x \in XSET$
		g2 : $P(y)$
		g3 : $StateA \in dom(TransE)$
		then
		a1 : $StateA := TransE(StateA)$
		end

A aut; B → 0 on e	**CONSTANTS** `isFinalB` **AXIOMS** ax1 : $isFinalB = \dots$ // Depends on B type	**Event** $e \mathrel{\widehat{=}}$ **where** g1 : $isFinalB(StateB) = TRUE$ ...

Table 3.2: Automata ASTD to Event-B translation rules

where

grd1 : $u \in USERSET$

grd2 : $StateCustomer(u) \in dom(TransCreateCustomer)$

then

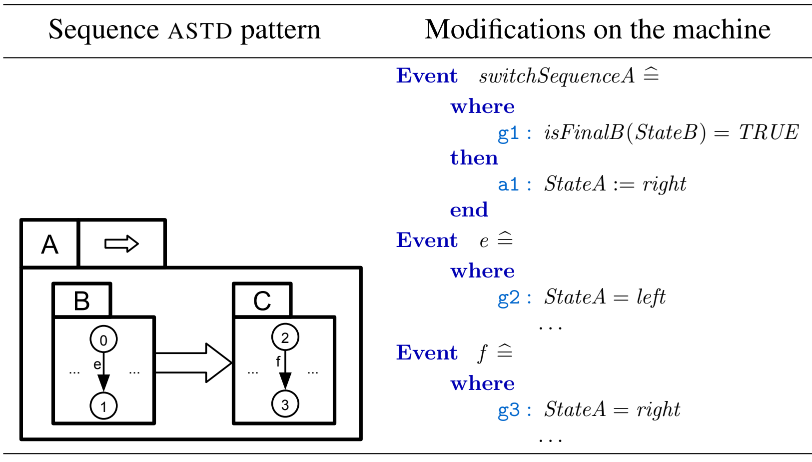
act1 : $StateCustomer(u) := TransCreateCustomer(StateCustomer(u))$

end

3.4.2 Sequence

Because of the number of possibilities to determine whether or not a sequence can switch from `left` state to `right` state, an extra event is introduced. This event is similar to an internal event of the IS and will verify that all the conditions for the switch from left to right side to happen holds and then change the state of the sequence. For example, if an ASTD named **A** is a sequence of ASTD **B** and **C**, the generated event will be called *switchSequenceA*. Then, in order to ensure that the current state allows the execution of

every events of ASTD **B** and **C**, a guard is added to each event of **B** and **C** to check if the state of ASTD **A** is *left* or *right* respectively. As for automata, a final predicate must be generated in the context for ASTD **B** state. Translation rule is described in Table 3.3.



Sequence ASTD pattern	Modifications on the machine
(diagram)	**Event** $switchSequenceA \mathrel{\widehat{=}}$ **where** g1 : $isFinalB(StateB) = TRUE$ **then** a1 : $StateA := right$ **end** **Event** $e \mathrel{\widehat{=}}$ **where** g2 : $StateA = left$ ... **Event** $f \mathrel{\widehat{=}}$ **where** g3 : $StateA = right$ ...

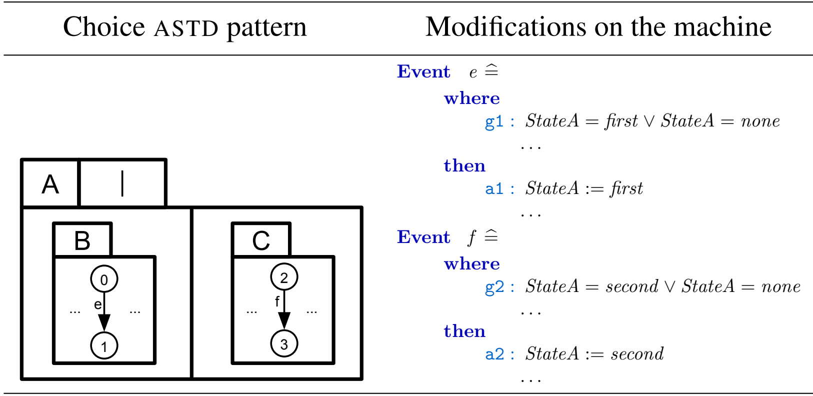
Table 3.3: Sequence ASTD to Event-B translation rule

3.4.3 Choice

A choice ASTD can be in three states as described by the general ASTD context: `none` when the choice is not made yet, `first` or `second` depending of the side chosen. The translation rule for a choice ASTD is presented in Table 3.4. If an ASTD named **A** is a choice between ASTD **B** and **C**, then a guard and an action are added to each event. Events from **B** will receive guard `g1` and action `a1`. A similar transformation of events from ASTD **C** is also needed with guard `g2` and action `a2`.

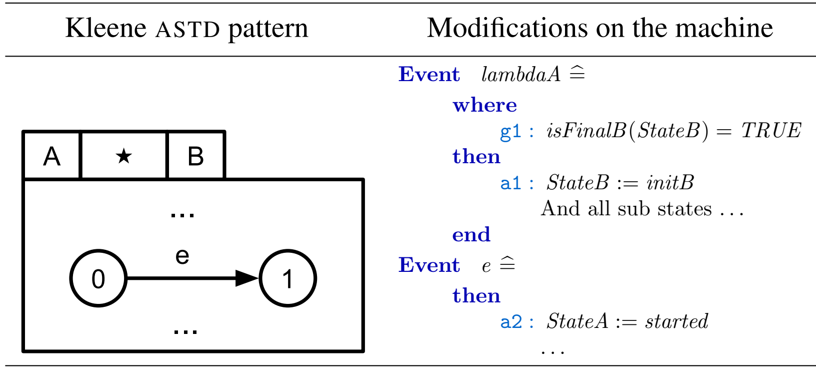
3.4.4 Kleene Closure

When an iteration of a Kleene closure ASTD is completed, its operand must be reset to initial state. For this reason, an additional event is generated. In the IS, this event is internal and hidden, in the ASTD specification, the semantics off Kleene operator handles the process, but in Event-B the reset must be described. This event will be activated when



Choice ASTD pattern	Modifications on the machine
A | B C (diagram: 0 –e→ 1; 2 –f→ 3)	**Event** $e \widehat{=}$ **where** g1 : $StateA = first \vee StateA = none$ ... **then** a1 : $StateA := first$ ... **Event** $f \widehat{=}$ **where** g2 : $StateA = second \vee StateA = none$ ... **then** a2 : $StateA := second$ ...

Table 3.4: Choice ASTD to Event-B translation rule



Kleene ASTD pattern	Modifications on the machine
A ★ B (diagram: 0 –e→ 1)	**Event** $lambdaA \widehat{=}$ **where** g1 : $isFinalB(StateB) = TRUE$ **then** a1 : $StateB := initB$ And all sub states ... **end** **Event** $e \widehat{=}$ **then** a2 : $StateA := started$ ...

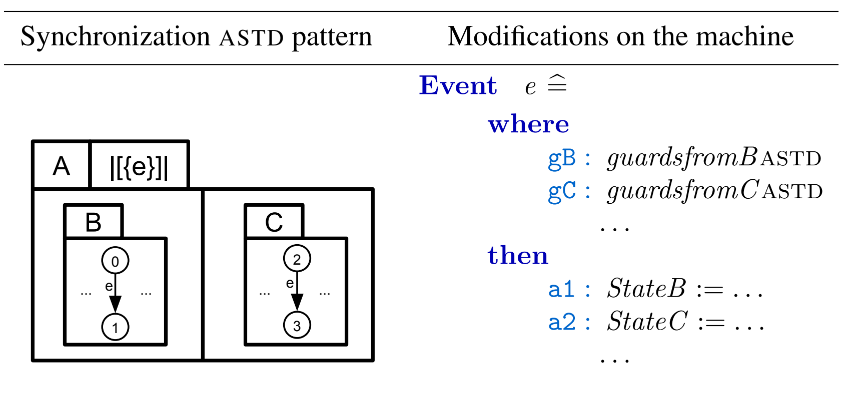
Table 3.5: Kleene closure ASTD to Event-B translation rule

the its operand is final, and will reinitialize all sub-states in the hierarchy. Table 3.5 details the resulting Event-B machine.

As presented for automata and sequence, a final predicate must be generated in the context for ASTD under the Kleene closure operator.

3.4.5 Synchronization Over a Set of Action Labels

For actions that are not synchronized, nothing is introduced or modified by the translation of synchronization ASTD. This is the case for interleaving ASTD and action labels not common to both operands of the parallel operator. In the case of a synchronized action,



Synchronization ASTD pattern	Modifications on the machine
A |[{e}]| B, C	**Event** $e \;\widehat{=}$ **where** gB : *guardsfromB*ASTD gC : *guardsfromC*ASTD ... **then** a1 : $StateB :=$ ... a2 : $StateC :=$ ... ...

Table 3.6: Synchronization ASTD to Event-B translation rule

guards from both operand must be put in conjunction, and substitutions applied conjointly. Table 3.6 details the resulting Event-B machine.

In our case study, the only synchronization ASTD is **main**. Common actions of both sides are only actions appearing in the ASTD **complaint**(c,d,u). For each one of the generated events of **complaint**(c,d,u), the guards `readyInCustomer` and `readyInDepartment` must hold. cc and dc states correspond to states where the customer and the department respectively are in the complaint quantified interleaving ASTD.

`readyInCustomer` : $StateCustomer(AssociationCustomer(c)) = cc$

`readyInDepartment` : $StateDepartment(AssociationDepartment(c)) = dc$

Theses guards check that the customer associated to the complaint c is in the state allowing him to complain *i.e.* created and not deleted, and if the department associated to the complaint c exists in the IS.

3.4.6 Quantified Interleaving

The quantified interleaving does not introduce additional constraints to events. Table 3.7 shows how variables induced by quantified interleaving are handled in events.

Entities and associations patterns are common in EB^3 and ASTD as mentioned in [38]. Such pattern are expressed using interleaving quantifications. In order to code in Event-B the association between several entities, a table variable must register their link. In our case

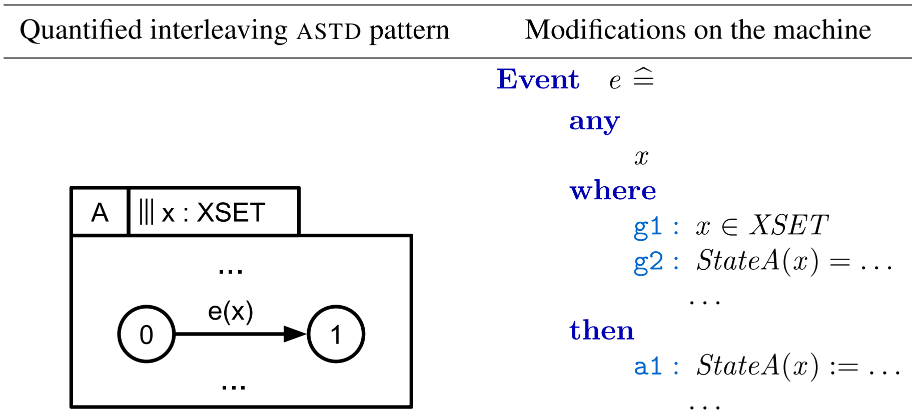
Quantified interleaving ASTD pattern	Modifications on the machine
	**Event** $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $x \in XSET$ g2 : $StateA(x) = \ldots$ $\ldots$ **then** a1 : $StateA(x) := \ldots$ $\ldots$

Table 3.7: Quantified interleaving ASTD to Event-B translation rule

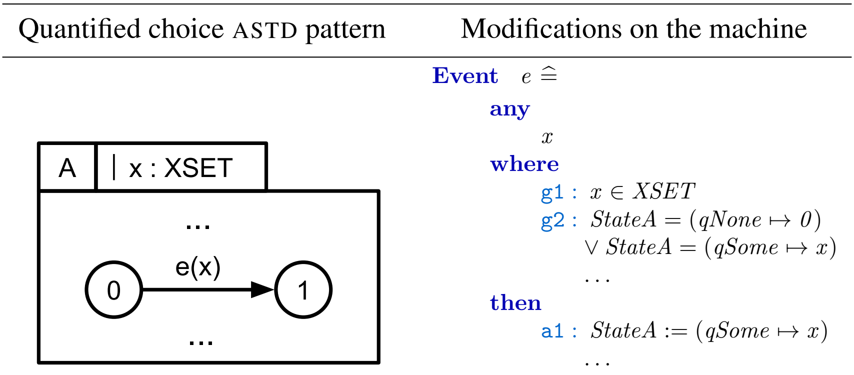
study, we can see that a 1-n association between a customer and a complaint is created. When a complaint is created, a unique customer u is linked to the complaint c. The same applies to the department associated to the complaint. An Event-B variable is created in order to save the link between a complaint and a customer (respectively a department) and is updated whenever a complaint is registered in the system.

3.4.7 Quantified Choice

Similarly to the choice operator, the quantified choice implies that for all events using it, a check is performed about whether the choice was made or not. In the case of an action labeled `e` and taking x as a parameter, where x is the variable of a quantified choice ASTD named **A** then the guard `g2` described in Table 3.8 must hold. The substitution `a1` must also be executed in case this is the first call of an action with this quantified variable. All the events of ASTD **A** will be modified to include this guard and substitution.

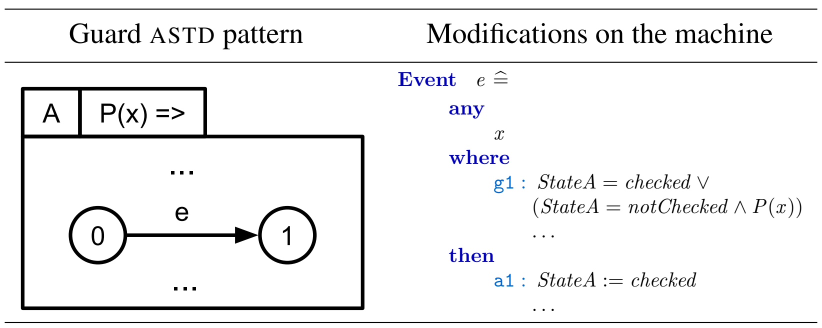
3.4.8 Guard

There are two cases for guard state: the guard was checked and held when we executed an event ; the guard did not hold, and no event was executed. These cases are handled with guard `g1` and substitution `a1` for a guard ASTD named **A** guarded with predicate $P(x)$. All



Quantified choice ASTD pattern	Modifications on the machine
A | x : XSET ... 0 —e(x)→ 1 ...	**Event** $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $x \in XSET$ g2 : $StateA = (qNone \mapsto 0)$ $\vee\ StateA = (qSome \mapsto x)$ ... **then** a1 : $StateA := (qSome \mapsto x)$ ...

Table 3.8: Quantified choice ASTD to Event-B translation rule



Guard ASTD pattern	Modifications on the machine
A | P(x) => ... 0 —e→ 1 ...	**Event** $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $StateA = checked\ \vee$ $(StateA = notChecked \wedge P(x))$ ... **then** a1 : $StateA := checked$ ...

Table 3.9: Guard ASTD to Event-B translation rule

the events of ASTD **A** will be modified to include this guard and substitution. Table 3.9 details the resulting Event-B machine.

3.4.9 Process call

An ASTD that calls other ASTD does not need any constraint over its actions in Event-B. The translation will be achieved as if the entire called ASTD was substituted for the ASTD call. We do not deal with recursive ASTD calls yet.

When the translation process is completed, we can now access all the tools offered by Rodin to animate, model check and prove elements of the translated ASTD specification.

3.5 Animation and Model Checking of the Case Study

The final generated system, a context and a machine, translated from our case study represents 270 lines of Event-B, including 40 constants, 5 sets and 29 axioms for the static part and 7 variants, 7 invariants, 17 events (one for initialization, 13 representing ASTD actions and 3 internal events for Kleene Closure induced resets) representing 57 guards and 33 actions for the dynamic part. During the construction of translation rules, animation helped to correct rules, to improve the quality of translation rules and to factor contexts in order to separate static elements from machine. It was chosen to limit the size of quantification sets to three elements each. Only three departments, customers and complaints can be registered inside the system at any time. The screen capture was taken after the execution of 150 events and shows the state of variables of the machine. In order to informally verify the consistency of the Event-B machine with the initial ASTD specification, we generated a set of traces of events executed via the ProB animator. Then, for each trace, we removed the internal events introduced by the translation process such as `lambdaComplaint(c)`. Then we interpreted the initial ASTD specification with *i*ASTD and executed the traces. We could not find a trace of events that could not be interpreted by *i*ASTD. A more formal proof of the consistency of the translation must be performed, but first results are encouraging. Formal proof of translation rules is work in progress, and will be based on simulation.

Regarding the Event-B machine, 86 proof obligations were generated and 62 were automatically proved. The 24 remaining are proved manually and involved functional and set operators that are known for not being proved automatically. The manual proofs raised no specific difficulty. This Event-B specification was model checked for deadlocks and invariant violations using the consistency checking feature of ProB. More than 111 500 nodes were visited and 226 000 transitions activated. No deadlock nor invariant violation were found. More invariant properties might be written in order to be proved. Since ASTD only focuses on event control and not on event effects on the IS, when an event is executed, there is no way to know only by looking at the ASTD specification how IS state will evolve. Hence, no invariant can be generated during the translation. But it could be interesting to express invariants on ASTD as it was done with Statecharts [84]. For instance we could add an invariant to ASTD **Department** saying that whenever transition DeleteDepartment(d) is active, no complaint about this department must be registered in the system.

3.6 Limitations, Conclusion and Future Work

We have presented a set of translation rules allowing generation of Event-B contexts and machines from ASTD specifications. The animation of the resulting machine using ProB [55] animator helped to find errors and to tweak translation rules. Kleene closure and sequence operators were the most tricky to translate since these operators defines the ordering of events and because they introduce additional events in order to code semantics of ASTD in Event-B. A formal proof of the translation rules will be performed in order to entrust the translation process.

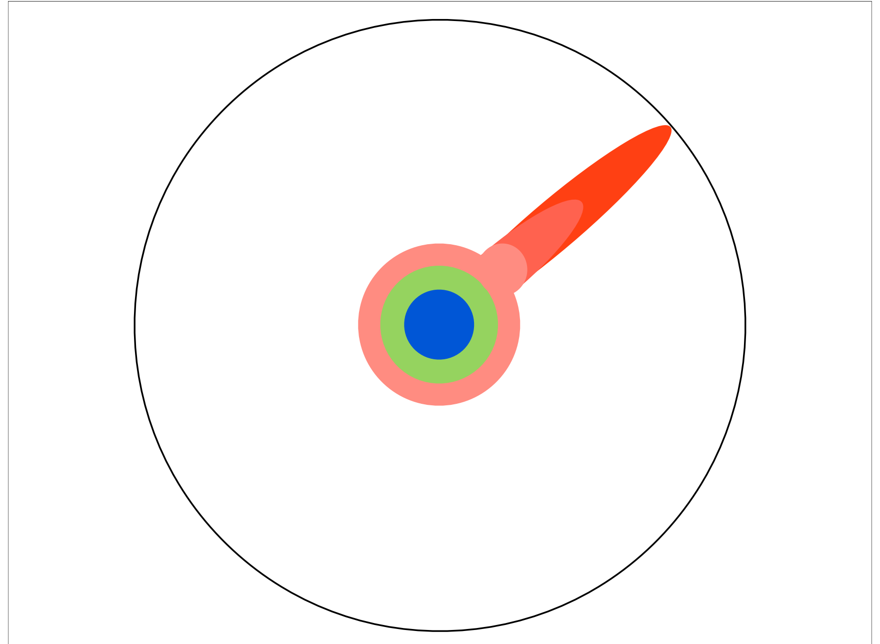
Refinement is one of the most important features of Event-B modelling process. In our approach, this aspect is missing. Indeed, we are translating an ASTD specification modelling a concrete system. Because of that, there is no need to refine the Event-B machine resulting from the translation process. It would have been relevant to introduce refinement in the translation process if a similar notion existed in ASTD, but it is currently not the case. Proof is an important aspect of Event-B that our approach would like to take advantage of. Alongside with formal IS specification, we advocate writing security or functional properties during the modeling process. This way, properties can be checked against the system as soon as it is modeled. Expressing these properties as Event-B invariants and proving invariant preservation in the translated machine is an important step of IS specification validation. Another feature of Event-B we do not use is composition. This may be very useful for the translation of some ASTD operators such as synchronization. It could lead to a more modular approach of translation, in a way similar to ASTD.

It would be interesting to compare the machine resulting of the translation process with a hand-written Event-B specification for the same system. Indeed, we would like to know if the automatic prover can do the same job with the hand-written and the translated machine. This study is work in progress and may result in an evolution of translation rules. Another step that we currently work on is to implement an ASTD modeler as a Rodin plugin. Using benefits from The Eclipse Graphical Modeling Framework (GMF) [18], a graphical editor could be used to build complete IS specifications. One could interpret them using the *i*ASTD [86] interpreter and then translate them to Event-B on the fly in order to perform model checking or proofs. This integrated tool would allow a great flexibility and would combine advantages of process algebra's power of expression, graphical representation's

ease of understanding and Event-B's tools for proving, checking and animating.

Chapitre 4

Traduction d'ASTD vers B



[65] Ph.D. étape 4 – Durant un doctorat, on étudie les connaissances les plus à la pointe. Au point qu'un jour, on se heurte aux limites des connaissances humaines, cette frontière qui, il y a quelques années, semblait si éloignée.

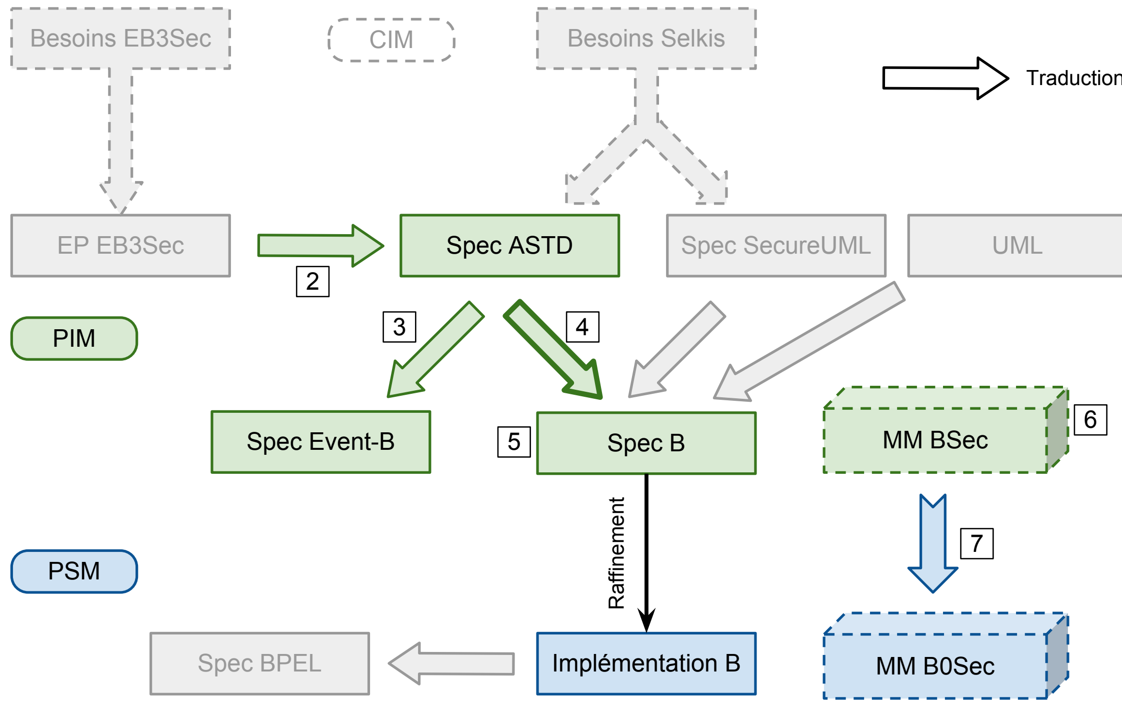


FIGURE 4.1 – Traduction de spécification ASTD en B et preuve de la traduction

Dans ce chapitre, nous nous intéressons à la traduction de spécifications ASTD en B, comme indiqué à l'étape 4 de la Figure 4.1. Cette traduction a trois motivations principales : (i) pouvoir combiner les machines B obtenues avec d'autres machines exprimant d'autres contraintes de contrôle d'accès ainsi que le comportement du système ; (ii) bénéficier des outils de preuve et de vérification de modèle offerts par la méthode B ; (iii) envisager une implémentation de la politique de contrôle d'accès via le raffinement B. Enfin cette traduction a été prouvée afin de pouvoir affirmer que les propriétés prouvées sur une machine B, résultat de la traduction, sont également vraies sur la spécification ASTD dont elle est issue.

4.1 Introduction

L'objectif principal de la traduction des ASTD en B ou Event-B est de pouvoir vérifier des propriétés sur les ASTD en utilisant les outils de B ou d'Event-B. Cela suppose que les deux spécifications (ASTD et B/Event-B) aient exactement le même comportement, c'est-à-dire que toute trace d'exécution d'opérations B/d'évènements Event-B correspond à une trace d'exécution ASTD et réciproquement. Or, lors de la traduction ASTD vers Event-B, de nouveaux évènements Event-B sont introduits. Ces derniers ne correspondent pas à des évènements de l'ASTD. Les traces acceptées sont donc différentes. Ces évènements ajoutés sont nécessaires pour traduire la fermeture de Kleene ou un ASTD de type séquence afin de coder respectivement l'initialisation avant une nouvelle itération ou le changement de côté d'une séquence. Il n'y a donc pas bi-simulation entre les spécifications ASTD et Event-B. On ne peut donc pas garantir qu'une propriété vraie en Event-B est également vraie sur la spécification ASTD.

Dans ce chapitre, nous proposons une traduction en B de spécifications ASTD. Cette traduction utilise les substitutions **SELECT** qui permettent d'éviter la création d'opérations additionnelles comme ce fut le cas en Event-B. Ainsi, si une spécification ASTD comporte n évènements d'étiquettes différentes, la machine B correspondant à sa traduction comportera n opérations. Cette traduction est décrite de manière inductive sur la structure de l'ASTD et se base sur les règles d'inférence de la sémantique d'exécution des ASTD. Dans la section 4.2 nous introduisons des notations pour simplifier l'expression des règles de la section 4.3. Puis nous proposons une ébauche de preuve de la traduction dans la section 4.4. L'application des règles de traductions sur une étude de cas est disponible en Appendix B. Des exemples dans chacune des sections décrivant la traduction d'un type d'ASTD sont également proposés. La liste des règles d'inférence de la sémantique d'exécution des ASTD est disponible en Appendix C.

4.2 Notations

Dans cette section, nous introduisons les notations utilisées dans la suite de ce chapitre. Ces notations ont pour but de simplifier l'écriture des règles de traduction et des preuves.

Notion	Notation
L'ensemble de tous les états de tous les ASTD	STATE
L'ensemble des types d'ASTD	TYPE
L'ensemble des évènements qu'un ASTD peut accepter	EVENT
L'ensemble des noms des ASTD	NAME
L'ASTD nommé a	$\mathbf{a}$
Le type de l'ASTD **a**	$\mathbf{a}^t$
L'état de l'ASTD **a**	$\mathbf{a}^s$
L'état initial de l'ASTD **a**	$\mathbf{a}^i$
Un évènement exécuté par **a**	$\sigma(\vec{x})$

TABLE 4.1 – Résumé des notations relatives aux ASTD

Notion	Notation
L'ensemble représentant NAME en B	$NAME$
La machine B, traduction de **a**	$\mathscr{M}_a = \theta(\mathbf{a})$
L'état de la machine B $\mathscr{M}_a$, représentant $\mathbf{a}^s$	$\mathscr{M}_a^s = \eta(\mathbf{a}^s)$
L'opération de $\mathscr{M}_a$ correspondant à l'évènement $\sigma(\vec{x})$ de l'ASTD **a**	$op(\sigma(\vec{x}))$
Une opération B nommée *sigma*	$sigma$
Sa précondition et le corps de sa substitution	$preOf(sigma)$, $thenOf(sigma)$

TABLE 4.2 – Résumé des notations relatives à B

4.2.1 Résumé

Le Table 4.1 et le Table 4.2 résument les notations introduites dans la suite de cette section.

4.2.2 Conventions

Soit STATE l'ensemble de tous les états possibles de tous les ASTD possibles ; soit TYPE l'ensemble des types ASTD, *i.e.* ASTDElem, Automaton, Sequence, Guard, Closure, Choice, Synchronization, QChoice, QSynchronization, ASTDCall. Soit EVENT l'ensemble des évènements qu'un ASTD peut accepter. Un évènement est noté $\sigma(\vec{x})$ où σ est l'étiquette de l'évènement et $\vec{x} = p_1, \ldots, p_n$ sont les paramètres de l'évènement. Les paramètres sont ty-

pés et nous supposons que ces types sont compatibles avec les types B, *i.e.* nous pouvons traduire ces types en B. Soit NAME l'ensemble des noms des ASTD. Soit *NAME* l'ensemble B modélisant NAME.

La fonction η - Traduction de l'état d'un ASTD

Soit η une fonction de traduction qui, à l'état d'un ASTD, associe un ensemble de variables B, et leur valeur respective, qui représentent cet état en B. Ces variables B sont appelées dans la suite variables d'état. Nous notons $\eta(\mathbf{a}^s)$ la traduction en variables B de l'état de l'ASTD **a**. Nous définissons l'état de la machine B traduction de l'ASTD **a** par $\mathscr{M}_a^s = \eta(\mathbf{a}^s)$. Les règles de traduction décrivant η sont présentées en section 4.3.

La fonction θ - Traduction d'un ASTD

Soit $\theta(\mathbf{a})$ la machine B résultant de la traduction complète de l'ASTD **a**. L'état de cette machine est le résultat de la traduction de l'état de l'ASTD en utilisant la fonction η. Les règles de traduction de θ sont décrites dans la section 4.3.

Initialisation

Nous utilisons parfois dans nos règles de traduction la notation :

$$[init(x)]S$$

Le terme $init(x)$ représente l'affectation de l'ensemble des variables d'état modélisant l'ASTD **x** dans son état initial. Ainsi, $init(x) = \eta(\mathbf{x}^i)$. S représente une substitution B. De même, $[init(x)]S$ est la substitution B telle que toutes les occurrences dans S des variables B dans $init(x)$ sont remplacées par leur valeur initiale respective. Ces valeurs initiales sont définies dans la section 4.3 pour chaque type d'ASTD à l'aide de constantes B nommées $init_\ldots$ et définies pour chaque sous-ASTD de la spécification. Ces constantes sont également utilisées dans l'initialisation de la machine B.

Accès aux préconditions des opérations B

Afin d'exprimer de manière plus simple nos règles de traduction, nous définissons ici deux fonctions. Soit $op(\sigma)$ une opération B qui possède une précondition *i.e.* la substitution de plus haut niveau est **PRE**. Alors nous définissons le prédicat $preOf(op(\sigma))$ comme étant la précondition de $op(\sigma)$. Nous définissons également la substitution $thenOf(op(\sigma))$ telle que :

$$
\begin{array}{lll}
op(\sigma) & \triangleq & \textbf{PRE} \\
 & & \quad preOf(op(\sigma)) \\
 & & \textbf{THEN} \\
 & & \quad thenOf(op(\sigma)) \\
 & & \textbf{END}
\end{array}
$$

Nous étendons ces fonctions à toutes les opérations. Si une opération n'est pas une substitution précondition, nous posons $preOf(op(\sigma)) = true$ et $thenOf(op(\sigma)) = op(\sigma)$. En utilisant la sémantique des substitutions généralisées, il est trivial de montrer que $op(\sigma)$ et **PRE** *true* **THEN** $op(\sigma)$ **END** sont deux substitutions équivalentes.

4.3 Traduction

Nous limitons notre traduction aux ASTD ne comportant pas d'automates non déterministes ou d'appels de processus mutuellement récursifs. Ces limitations peuvent être levées en adaptant les règles de traduction présentées ici. Cependant, ces limitations ne sont pas contraignantes pour la traduction d'ASTD modélisant des règles de sécurité ou des SI.

La définition de la traduction d'un ASTD en B est faite par induction sur la structure de l'ASTD. Pour traduire un ASTD, il faut connaitre la traduction de ses sous-ASTD. Ainsi, la traduction s'effectue en profondeur d'abord, et commence donc forcément par un ASTD de type automate qui n'a pas d'état hiérarchique, c'est-à-dire que tous ses états sont élémentaires.

Dans les sections qui suivent, nous reprenons les définitions faites dans le rapport technique [33] définissant formellement les ASTD et nous montrons la traduction de chacune

des structures et concepts présentés. Nous utilisons également les règles d'inférence des ASTD, disponibles en Appendix C.

4.3.1 Choix de modélisation

L'écriture des opérations B peut se faire de plusieurs manières. Les opérations peuvent être des substitutions précondition ou des substitutions de sélection sans précondition. L'utilisation de substitutions de sélection sans précondition est un style de modélisation dit évènementiel. Nous avons choisi d'utiliser les préconditions pour nos opérations afin d'éviter le blocage des opérations induit par les substitutions de sélection. En effet, quand dans un **SELECT** aucun des prédicats n'est vrai, l'opération est exécutée mais se bloque. Dans le cas des préconditions, si le prédicat est faux, l'opération n'est pas exécutée. Ce comportement correspond mieux à celui de l'exécution des ASTD.

À l'intérieur des substitutions **PRE**, nous utilisons des substitutions **SELECT**. Chacune des clauses des **SELECT** correspond à l'application d'une des règles du type d'ASTD que nous traduisons. Nous aurions pu utiliser des substitutions **IF** à la place, mais nous avons besoin du non-déterminisme. En effet, lors de l'exécution d'un évènement par un ASTD, il est parfois possible d'appliquer plusieurs règles. Le choix de la règle à appliquer est fait de manière non déterministe. En utilisant les substitutions **SELECT**, nous conservons ce principe dans notre traduction.

4.3.2 Automate

L'ensemble des ASTD de type automate est modélisé par l'ensemble $\mathsf{Automaton}$:

$$\mathsf{Automaton} \triangleq \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$$

- $name \in N$ est le nom de l'ASTD.
- $\Sigma \subseteq \mathsf{Event}$ est l'alphabet de l'automate. Nous créons en B pour chaque élément σ de Σ une opération nommée $op(\sigma)$ ayant pour substitution $skip$.
- $N \subseteq \text{NAME} - \{\mathsf{H}, \mathsf{H^*}\}$ est l'ensemble des noms des états de l'automate. H et $\mathsf{H^*}$ représentent respectivement l'état historique et l'état historique profond d'un ASTD de type automate. Ces deux états spéciaux sont issus des Statecharts de Harel.

- $\nu \in N \rightarrow \mathsf{ASTD}$ est une fonction indiquant si chaque état est élémentaire ou lui-même un ASTD. Nous utilisons cette fonction dans notre processus de traduction mais elle n'est pas traduite en B.
- $\delta = \langle \gamma, \sigma, \phi, \mathit{final?} \rangle$ est la relation définissant les transitions. Dans notre traduction, nous nous limitons aux ASTD de type automate qui sont déterministes. De ce fait, δ est, dans notre cas, une fonction.
  - $\sigma \in \mathsf{Event}$. Posons $op(\sigma) = sigma$ dans la suite.
  - γ représente la flèche (*i.e.*, une transition entre deux états d'un automate). Il y a trois types de flèches :
    1. $\langle \mathsf{loc}, n_1, n_2 \rangle$ représente une transition locale entre les états n_1 et n_2.
    2. $\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$ représente une transition de n_1 vers un sous-état $n_{2_\flat}$ de l'ASTD n_2 avec $\nu(n_2) \in \mathsf{Automaton}$.
    3. $\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$ représente une transition du sous-état $n_{1_\flat}$ de l'ASTD n_1 vers n_2 avec $\nu(n_1) \in \mathsf{Automaton}$.
  - ϕ est un prédicat de la logique du premier ordre. Les gardes peuvent utiliser plusieurs symboles :
    1. *Les attributs du système*. Nous ne nous occupons pas du calcul de ces attributs. Nous supposons que ces attributs sont calculés par une machine externe et que nous pouvons appeler ses opérations pour récupérer la valeur de ces attributs.
    2. *Les variables de quantification*.
    3. *Les paramètres d'une définition* ASTD .
    4. *Les constantes*.

    Nous supposons que la garde ϕ est un prédicat du premier ordre qui peut être traduit en B. Nous ne détaillons pas les règles de traduction de ce prédicat.
  - *final*? $\in \mathsf{Boolean}$ indique si la transition n'est active que dans les états finaux. Ces transitions finales sont indiquées par une décoration de leur flèche par une « boule » à leur source.
- $SF \subseteq N$ représente l'ensemble des états finaux simples.
- $DF \subseteq N$ représente l'ensemble des états finaux profonds. Un état final profond est final si et seulement si son ASTD est dans un état final. Les états ne peuvent pas être simultanément finaux simples et finaux profonds : $DF \cap SF = \varnothing$.

– $n_0 \in N$ est le nom de l'état initial.

Un ASTD de type automate est traduit par plusieurs termes B :

1. Un ensemble B nommé $AutState_name$ est créé. Il contient une représentation en B des noms des états de l'ASTD. C'est donc l'équivalent en B de N. Cet ensemble est caractérisé par $AutState_name \subset NAME$
2. Une constante B nommée $init_name$ est créée. Elle est de type $AutState_name$. Sa valeur est la représentation B de l'état n_0.
3. Deux ensembles B nommés SF_name et DF_name sont définis. Ils représentent respectivement SF et DF et sont tels que $SF \subset AutState_name$ et $DF \subset AutState_name$.
4. Afin de représenter δ, plusieurs constantes B sont créées. Pour chaque étiquette σ de Σ, une constante est créée. Elle est nommée $t_name_op(\sigma)$. Elle représente la fonction de transition partielle dans *name* pour l'étiquette σ.

   Puis nous appliquons les étapes suivantes pour chaque élément de δ étiqueté par σ. Afin de décrire le calcul de la relation représentée par cette constante, nous utilisons ci-dessous la notation B, par souci de concision. Bien entendu, il s'agit de spécifier l'algorithme de traduction ; les substitutions ci-dessous ne font pas partie de la machine B calculée pour la traduction, elles sont plutôt utilisées pour calculer cette machine B. À des fins d'illustration, nous faisons référence aux transitions de l'ASTD de la Figure 4.2.

   – Si la transition est locale, c'est-à-dire que sa flèche est de type loc, nous modifions la fonction de transition de la manière suivante :

   $$t_name_sigma := t_name_sigma \lhd\!\!\!+ \{n1 \mapsto n2\}$$

   Dans l'ASTD de la Figure 4.2, la transition $e3$ est locale à l'ASTD **n**. Les transitions $e2$ et $e4$ sont locales à l'ASTD **s1** et **s2** respectivement.

   – Si la transition est telle que sa flèche est de type tsub, nous modifions la fonction de transition de la manière suivante :

   $$t_name_sigma := t_name_sigma \lhd\!\!\!+ \{n1 \mapsto n2\}$$

La traduction de l'ASTD n_2 est également modifiée pour prendre en compte cette transition. Le terme $out(n1)$ est ajouté à l'ensemble des noms des états de $AutState_n2$, et sa fonction de transition est étendue de la manière suivante.

$$t_n2_sigma := t_n2_sigma \lhd\!\!\!+ \{out(n1) \mapsto n2b\}$$

où *out* est une fonction qui prend pour argument la représentation B d'un état et qui retourne une représentation B d'un état donc le nom est préfixé par *out_*. Par exemple $out(s1) = out_s1$. Ceci nous permet de représenter le fait qu'un état extérieur à l'ASTD est relié à un état de l'ASTD par une transition. Dans l'ASTD **n** de la Figure 4.2, la transition $e1$ est de type tsub.

– Si la transition est telle que sa flèche est de type fsub, nous modifions la fonction de transition de la manière suivante :

$$t_name_sigma := t_name_sigma \lhd\!\!\!+ \{n1 \mapsto n2\}$$

La traduction de l'ASTD n_1 n'est pas modifiée puisque l'exécution de cette transition ne modifie pas l'état de $n1$. Cette propriété permet de conserver l'historique d'un état. À une étape ultérieure de notre traduction, une condition est ajoutée sur cette transition afin de vérifier que $n1$ est bien dans l'état $n1b$. Il n'y a pas de modification de la fonction de transition de $n1$ pour que l'état de $n1$ ne soit pas modifié. Cela permet d'utiliser la variable codant l'état de $n1$ comme fonction historique, mémorisant l'état de $n1$ avant que l'ASTD quitte cet état et qui peut être utilisé si un état historique est défini dans $n1$. Dans l'ASTD **n** de la Figure 4.2, la transition $e5$ est de type fsub.

Un ASTD de type automate possède un état noté $\langle \mathsf{aut}_\circ, n, h, s \rangle$, tel que :

– $n \in$ NAME est le nom de l'état courant.
– $h \in \mathrm{NAME} \nrightarrow \mathsf{State}$ est une fonction partielle qui indique le dernier état visité des sous-états d'un automate. Cette fonction est utilisée pour implémenter la notion d'état historique introduite dans les statecharts et disponible dans la notation ASTD.
– $s \in \mathsf{State}$ est l'état de l'astd n. Cet état peut être composé ou élémentaire et noté elem.

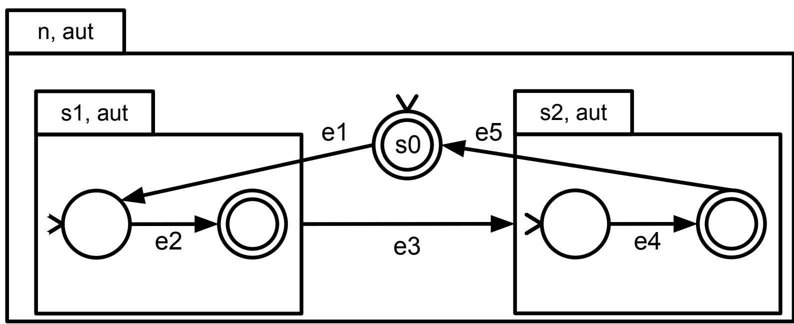


FIGURE 4.2 – Un exemple d'ASTD de type automate

Nous avons remarqué qu'il n'est pas nécessaire de traduire la fonction codant l'historique d'un ASTD. Avec notre approche, les variables d'état des sous-ASTD conservent la valeur des états visités. Il n'est donc pas nécessaire de stocker ces variables ailleurs.

L'état d'un ASTD de type automate, noté $\eta(\mathbf{a}^s)$ est représenté en B par :

- Une variable nommée $state_name$ et typée $state_name \in AutState_name$. Sa valeur est la représentation de n en B, le nom de l'état courant de l'automate.
- $\eta(s) = \eta(\nu(n)^s)$ représente en B s, l'état courant de l'ASTD qui peut être composé ou élémentaire.

Nous définissons ici un prédicat B déterminant si l'état d'un ASTD de type automate est final. Ce prédicat est nommé $final_name$ dans la suite, et fait référence à $final_x$ le prédicat qui détermine si l'état de l'ASTD $\mathbf{x}$ est final :

$$(state_name \in DF_name \wedge (\bigwedge_{x \in DF_name} (state_name = x) \Rightarrow final_x))$$
$$\vee\ state_name \in SF_name$$

Nous créons maintenant l'opération associée à σ que nous notons $op(\sigma)$. Soit T_σ l'ensemble des transitions étiquetées par σ apparaissant dans la fonction de transition δ. Soit C_t la condition d'activation pour $t \in T_\sigma$, calculée à partir du Table 4.3. Soit T_t la substitution correspondant à la mise à jour de l'état suite à l'exécution d'une transition $t \in T_\sigma$, calculée à partir du Table 4.3.

Type de la flèche de t	C_t	T_t
$\langle \mathsf{loc}, n_1, n_2 \rangle$	c_{all}	$t_{all} \parallel t_{init1,5}$
$\langle \mathsf{tsub}, n_1, n_2, n_{2\flat} \rangle$ avec $n_{2\flat} \notin \{\mathsf{H}, \mathsf{H}^*\}$	c_{all}	$t_{all} \parallel t_{tosub} \parallel t_{init2,3}$
$\langle \mathsf{tsub}, n_1, n_2, \mathsf{H} \rangle$	c_{all}	$t_{all} \parallel t_{init2,3}$
$\langle \mathsf{tsub}, n_1, n_2, \mathsf{H}^* \rangle$	c_{all}	t_{all}
$\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$	$c_{all} \wedge c_{fromsub}$	$t_{all} \parallel t_{init1,5}$
La transition a une garde g	$\wedge\ c_{guard}(g)$	
La transition est finale	$\wedge\ c_{final}$	
La transition a un paramètre p qui est une constante cst	$\wedge\ c_{const}(p, cst)$	

TABLE 4.3 – Prédicats et substitutions pour chaque type de transitions

$$
\begin{aligned}
c_{all} &\triangleq state_name \in dom(t_name_sigma) \\
c_{fromsub} &\triangleq (state_name = n1) \Rightarrow state_n1 = n1b \\
c_{final} &\triangleq (state_name = n1) \Rightarrow final_n1 \\
c_{guard}(g) &\triangleq (state_name = n1) \Rightarrow g \\
c_{const}(p, cst) &\triangleq (state_name = n1) \Rightarrow (p = cst) \\
t_{all} &\triangleq state_name := t_name_sigma(state_name) \\
t_{tosub} &\triangleq state_n2 := t_n2_sigma(out(state_name)) \\
t_{init1,5} &\triangleq initAll_n2 \\
t_{init2,3} &\triangleq initAll_n2b
\end{aligned}
$$

Soit M_σ l'ensemble des états hiérarchiques de l'ASTD dans lesquels σ apparaît. Soit op_m_σ avec $m_\sigma \in M_\sigma$ l'opération résultant de la traduction de l'ASTD m_σ. L'opération $op(\sigma)$ s'écrit alors :

$$
\begin{array}{rcl}
op(\sigma) & = & \textbf{PRE} \\
& & \quad \text{condition de typage des paramètres de l'opération} \wedge \\
& & \quad (\bigvee_{t \in T_\sigma} C_t \vee \bigvee_{m_\sigma \in M_\sigma} (state_a = m_\sigma \wedge preOf(op_m_\sigma))) \\
& & \textbf{THEN} \\
& & \quad \textbf{SELECT} \\
& & \qquad \textbf{WHEN } C_t \textbf{ THEN } T_t \qquad \text{pour chaque } t \in T_\sigma \\
& & \qquad \textbf{WHEN } state_a = m_\sigma \wedge preOf(op_m_\sigma) \\
& & \qquad\quad \textbf{THEN } thenOf(op_m_\sigma) \qquad \text{pour chaque } m_\sigma \in M_\sigma \\
& & \quad \textbf{END} \\
& & \textbf{END}
\end{array}
$$

Les substitutions $initAll_n2$ et $initAll_n2b$ initialisent les variables codant l'état des sous-ASTD de $n2$ et $n2b$ respectivement.

Le Table 4.3 ainsi que les conditions et substitutions décrites ci-dessus sont en fait les conditions et modifications sur l'état courant de l'ASTD dans les règles d'inférence de la sémantique d'exécution des ASTD. La seule différence avec les règles d'inférence se fait au niveau de la gestion des états historiques. L'ajout d'une fonction mémorisant l'état dans lequel est un ASTD automate B lui-même état d'un autre ASTD automate A n'est pas nécessaire puisque la variable codant l'état de B n'est modifiée que quand l'état courant de A est B. Les variables jouent ainsi le rôle de fonction historique sauvegardant les états des états hiérarchiques lorsque l'exécution quitte cet état. Pour obtenir la valeur historique d'un état, il suffit de lire la variable d'état lui correspondant.

Le Table 4.4 fait le lien entre les substitutions décrites dans le Table 4.3 et les règles d'inférence des ASTD de type automate, disponibles en Appendix C. De même, le Table 4.5 fait le lien avec les prédicats des règles d'inférence.

Condition	B	Règle aut	ASTD
c_{all}	$state_name \in dom(t_name_sigma)$	1, 2, 3, 4, 5	La transition existe
$c_{fromsub}$	$state_name = n1 \Rightarrow state_n1 = n1b$	5	$name(s) = n1_\flat$
c_{final}	$state_name = n1 \Rightarrow final_n1$	Ψ	$final? \Rightarrow final_{\nu(n_1)}(s)$
$c_{guard}(g)$	$state_name = n1 \Rightarrow g$	Ψ	g
$c_{const}(p, cst)$	$state_name = n1 \Rightarrow p = cst$	Ψ	$[\Gamma]\ \sigma' = \sigma$
$c_{other}(p)$	Le type de p est correct	Ψ	$[\Gamma]\ \sigma' = \sigma$

TABLE 4.4 – Lien entre les substitutions et les règles d'inférence

Substitution	B	Règle aut	ASTD
t_{all}	$state_name :=$ $t_name_sigma(state_name)$	1, 2, 3, 4, 5	Application de la transition
t_{tosub}	$state_n2 :=$ $t_n2_sigma(out(state_name))$	2	$(\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2b}, \ldots))$
$t_{init1,5}$	$initAll_n2$	1 et 5	$init(\nu(n_2))$
$t_{init2,3}$	$initAll_n2b$	2 et 3	$init(\nu(n_{2b}))$

TABLE 4.5 – Lien entre les prédicats et les règles d'inférence

Exemple de traduction

La Figure 4.3 présente un ASTD possédant des transitions couvrant les règles d'exécution aut_1, aut_2, aut_3, aut_4 et aut_5 des ASTD de type automate.

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de type automate de la Figure 4.3.

```
MACHINE
    a1(xx)

CONSTRAINTS
    xx ∈ INT

SETS
    AutState_s4 = {out_s1, s5, s6, s7} ;
    AutState_a1 = {s0, s1, s2, s3, s4}
```

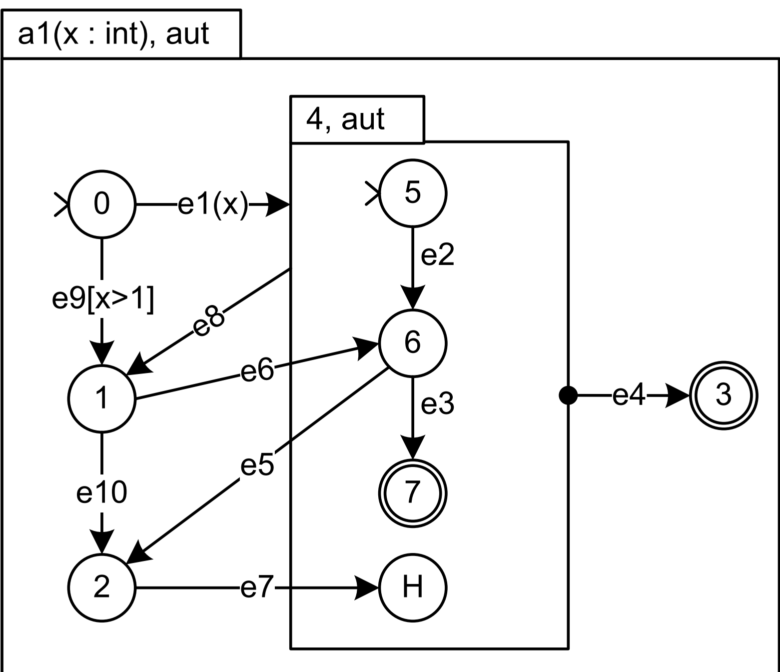


FIGURE 4.3 – Un ASTD de type automate avec tous les types de transitions possibles

```
CONSTANTS
    init_s4,
    t_s4_e2,
    t_s4_e3,
    DF_s4, SF_s4,
    init_a1,
    t_a1_e1,
    t_a1_e4,
    t_a1_e5,
    t_a1_e6,
    t_s4_e6,
    t_a1_e7,
    t_a1_e8,
    t_a1_e9,
    t_a1_e10,
    DF_a1, SF_a1


PROPERTIES
    DF_s4 ⊆  AutState_s4 ∧
    SF_s4 ⊆  AutState_s4 ∧
    DF_s4 ∩ SF_s4 = {} ∧
    DF_s4 = {} ∧
    SF_s4 = {s7} ∧
    init_s4 = s5 ∧
    t_s4_e2 = {s5 |-> s6} ∧
    t_s4_e3 = {s6 |-> s7}  ∧
    DF_a1 ⊆  AutState_a1 ∧
    SF_a1 ⊆  AutState_a1 ∧
    DF_a1 ∩ SF_a1 = {} ∧
    DF_a1 = {} ∧
    SF_a1 = {s3} ∧
    init_a1 = s0 ∧
    t_a1_e1 = {s0 |-> s4} ∧
    t_a1_e4 = {s4 |-> s3} ∧
    t_a1_e5 = {s4 |-> s2} ∧
    t_a1_e6 = {s1 |-> s4} ∧
    t_s4_e6 = {out_s1 |-> s6} ∧
```

```
    t_a1_e7 = {s2 |-> s4} ∧
    t_a1_e8 = {s4 |-> s1} ∧
    t_a1_e9 = {s0 |-> s1} ∧
    t_a1_e10= {s1 |-> s2}

VARIABLES
    State_s4, State_a1

INVARIANT
    State_s4 ∈ AutState_s4 ∧
    State_a1 ∈ AutState_a1

INITIALISATION
    State_s4 := init_s4 ||
    State_a1 := init_a1

OPERATIONS

    e1(x_a1) =
    PRE  x_a1 ∈ INT ∧
        (State_a1 ∈ dom ( t_a1_e1 ) ∧ x_a1 = xx)
    THEN
        SELECT
            State_a1 ∈ dom ( t_a1_e1 )
        THEN
            State_a1 := t_a1_e1 ( State_a1 )
        END
    END ;

    e2 =
    PRE
        State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e2 )
    THEN
        SELECT
            State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e2 )
        THEN
            SELECT
                State_s4 ∈ dom ( t_s4_e2 )
            THEN
```

```
            State_s4 := t_s4_e2 ( State_s4 )
        END
    END
END ;

e3 =
PRE
    State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e3 )
THEN
    SELECT
        State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e3 )
    THEN
        SELECT
            State_s4 ∈ dom ( t_s4_e3 )
        THEN
            State_s4 := t_s4_e3 ( State_s4 )
        END
    END
END ;

e4 =
PRE (State_a1 ∈ dom ( t_a1_e4 ) ∧ State_s4 ∈ SF_s4)
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e4 ) ∧ State_s4 ∈ SF_s4
    THEN
        State_a1 := t_a1_e4 ( State_a1 )
    END
END ;

e5 =
PRE (State_a1 ∈ dom ( t_a1_e5 ) ∧ State_s4 = s6)
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e5 ) ∧ State_s4 = s6
    THEN
        State_a1 := t_a1_e5 ( State_a1 )
    END
END ;
```

```
e6 =
PRE (State_a1 ∈ dom ( t_a1_e6 ))
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e6 )
    THEN
        State_a1 := t_a1_e6 ( State_a1 )  ||
        State_s4 := t_s4_e6 ( out_s1 )
    END
END ;

e7 =
PRE (State_a1 ∈ dom ( t_a1_e7 ))
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e7 )
    THEN
        State_a1 := t_a1_e7 ( State_a1 )
    END
END ;

e8 =
PRE (State_a1 ∈ dom ( t_a1_e8 ))
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e8 )
    THEN
        State_a1 := t_a1_e8 ( State_a1 )
    END
END ;

e9 =
PRE (State_a1 ∈ dom ( t_a1_e9 ) ∧ xx >1 )
THEN
    SELECT
        State_a1 ∈ dom ( t_a1_e9 ) ∧ xx >1
    THEN
        State_a1 := t_a1_e9 ( State_a1 )
```

```
        END
    END ;

    e10 =
    PRE (State_a1 ∈ dom ( t_a1_e10 ) )
    THEN
        SELECT
            State_a1 ∈ dom ( t_a1_e10 )
        THEN
            State_a1 := t_a1_e10 ( State_a1 )
        END
    END

END
```

Statistiques et optimisations Le typage de la machine B présentée a été validé (*type-check*) par l'atelier B et l'outil génère 13 obligations de preuves toutes prouvées automatiquement. Nous voyons en observant les opérations $e3$ et $e4$ que des règles de simplification des substitutions **SELECT** pourraient réduire la taille des substitutions générées. Cette optimisation peut s'appliquer quand des **SELECT** sont imbriqués et que les conditions des clauses **SELECT** profondes sont impliquées par celles de la clause du **SELECT** de niveau supérieur. Par exemple, l'opération $e3$ peut être réécrite :

```
    e3 =
    PRE
        State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e3 )
    THEN
        SELECT
            State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e3 )
        THEN
            State_s4 := t_s4_e3 ( State_s4 )
        END
    END
```

De plus, certains **SELECT** ne sont pas utiles car ils ne possèdent qu'une branche qui est toujours sélectionnée du fait des préconditions. Ces substitutions peuvent être retirées. Par exemple, l'opération $e3$ résultant de l'optimisation précédente serait réécrite de la manière suivante :

```
e3 =
PRE
    State_a1 = s4 ∧ State_s4 ∈ dom ( t_s4_e3 )
THEN
    State_s4 := t_s4_e3 ( State_s4 )
END
```

4.3.3 Séquence

L'ensemble des ASTD de type séquence est modélisé par $\langle \Rightarrow, name, fst, snd \rangle$, avec *fst* et *snd* représentant deux ASTD.

Un ASTD de type séquence possède un état noté $\langle \Rightarrow_\circ, [\mathsf{fst} \mid \mathsf{snd}], s \rangle$, tel que s représente l'état de *fst* ou de *snd* selon la valeur du second élément. Afin de représenter l'état d'un ASTD de type séquence en B, l'ensemble B $SEQUENCE = \{fst, snd\}$ est défini.

L'état d'un ASTD de type séquence est représenté en B par :

- Une variable nommée $stateSequence_name \in SEQUENCE$ qui vaut *fst* si l'état est fst et *snd* si l'état est snd.
- $\eta(s) = \eta(fst^s)$ ou $\eta(s) = \eta(snd^s)$ qui représente en B s, l'état courant de l'ASTD *fst* ou *snd* selon la valeur de $stateSequence_name$.

L'état initial d'un ASTD de type séquence est défini par la constante $init_Sequence = fst$.

L'état de l'ASTD *name* de type séquence est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$final_snd \wedge ((stateSequence_name = fst) \Rightarrow final_fst)$$

Si σ apparaît dans *fst* ou dans *snd*, nous disposons des traductions $op(\sigma)_{fst}$ et $op(\sigma)_{snd}$ respectivement. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_{fst}$ et $op(\sigma)_{snd}$. Soient

$$\begin{aligned}
C1 &= stateSequence_name = fst \wedge preOf(op(\sigma)_{fst}) \\
C2 &= stateSequence_name = fst \wedge final_fst \wedge [init(snd)]preOf(op(\sigma)_{snd}) \\
C3 &= stateSequence_name = snd \wedge preOf(op(\sigma)_{snd})
\end{aligned}$$

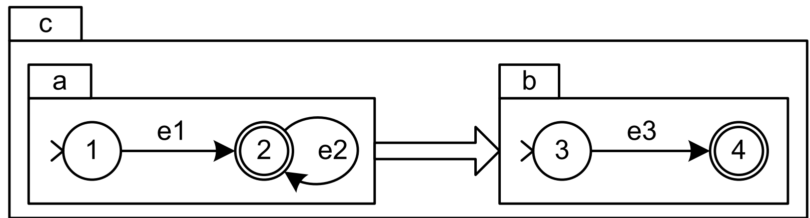


FIGURE 4.4 – Un exemple d'ASTD de type séquence

Alors

$$
\begin{aligned}
op(\sigma) \;=\; & \textbf{PRE} \\
& \quad C1 \vee \qquad \text{si } \sigma \text{ apparaît dans } \mathit{fst} \\
& \quad C2 \vee C3 \qquad \text{si } \sigma \text{ apparaît dans } \mathit{snd} \\
& \textbf{THEN} \\
& \quad \textbf{SELECT } C1 \textbf{ THEN } \mathit{thenOf}(op(\sigma)_{\mathit{fst}}) \\
& \quad \textbf{WHEN } C2 \textbf{ THEN } \mathit{stateSequence_a} := \mathit{snd} \\
& \qquad\qquad\qquad \parallel [\mathit{init}(\mathit{snd})]\mathit{thenOf}(op(\sigma)_{\mathit{snd}}) \\
& \quad \textbf{WHEN } C3 \textbf{ THEN } \mathit{thenOf}(op(\sigma)_{\mathit{snd}}) \\
& \quad \textbf{END}
\end{aligned}
$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type séquence, soit $\Rightarrow_1$, $\Rightarrow_2$ et $\Rightarrow_3$ qui correspondent aux cas $C1$, $C2$ et $C3$ respectivement.

Exemple de traduction

La Figure 4.4 présente un ASTD de type séquence.

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de type séquence de la Figure 4.4.

```
MACHINE
    sequence
```

```
SETS
    SEQUENCE = {fst, snd} ;
    AutState_a = {s1, s2} ;
    AutState_b = {s3, s4}

CONSTANTS
    init_a,
    init_b,
    init_Sequence,
    t_a_e1,
    t_a_e2,
    t_b_e3,
    DF_a, SF_a,
    DF_b, SF_b

PROPERTIES

    init_a = s1 ∧
    init_b = s3 ∧
    init_Sequence = fst ∧
    t_a_e1 = {s1 |-> s2} ∧
    t_a_e2 = {s2 |-> s2} ∧
    t_b_e3 = {s3 |-> s4} ∧
    DF_a ⊆  AutState_a ∧
    SF_a ⊆  AutState_a ∧
    DF_a ∩ SF_a = {} ∧
    DF_a = {} ∧
    SF_a = {s2} ∧
    DF_b ⊆  AutState_b ∧
    SF_b ⊆  AutState_b ∧
    DF_b ∩ SF_b = {} ∧
    DF_b = {} ∧
    SF_b = {s4}

VARIABLES
    State_a,
    State_b,
    StateSequence_c
```

```
INVARIANT
    State_a ∈ AutState_a ∧
    State_b ∈ AutState_b ∧
    StateSequence_c ∈ SEQUENCE

INITIALISATION
    State_a := init_a ||
    State_b := init_b ||
    StateSequence_c := init_Sequence

OPERATIONS
    e1 =
    PRE
        (StateSequence_c = fst ∧ State_a ∈ dom(t_a_e1))
    THEN
        SELECT
            StateSequence_c = fst ∧ State_a ∈ dom(t_a_e1)
        THEN
            SELECT
                State_a ∈ dom(t_a_e1)
            THEN
                State_a := t_a_e1(State_a)
            END
        END
    END;

    e2 =
    PRE
        (StateSequence_c = fst ∧ State_a ∈ dom(t_a_e2))
    THEN
        SELECT
            StateSequence_c = fst ∧ State_a ∈ dom(t_a_e2)
        THEN
            SELECT
                State_a ∈ dom(t_a_e2)
            THEN
                State_a := t_a_e2(State_a)
            END
```

```
        END
    END;

    e3 =
    PRE
        (StateSequence_c = fst ∧ State_a ∈ SF_a ∧ init_b ∈ dom(t_b_e3) ) or
        (StateSequence_c = snd ∧ State_b ∈ dom(t_b_e3) )
    THEN
        SELECT
            StateSequence_c = fst ∧ State_a ∈ SF_a ∧ init_b ∈ dom(t_b_e3)
        THEN
            SELECT
                init_b ∈ dom(t_b_e3)
            THEN
                StateSequence_c := snd ||
                State_b := t_b_e3(init_b)
            END
        WHEN
            StateSequence_c = snd ∧ State_b ∈ dom(t_b_e3)
        THEN
            SELECT
                State_b ∈ dom(t_b_e3)
            THEN
                State_b := t_b_e3(State_b)
            END
        END

    END

END
```

Statistiques et optimisations Le typage de la machine B présentée a été validé (*type-check*) par l'atelier B et l'outil génère sept obligations de preuves toutes prouvées automatiquement.

De la même manière que pour les ASTD de type automate, une optimisation des substitutions générées est envisageable. Par exemple pour l'opération $e3$, on obtient :

```
e3 =
PRE
    (StateSequence_c = fst ∧ State_a ∈ SF_a ∧ init_b ∈ dom(t_b_e3) ) or
    (StateSequence_c = snd ∧ State_b ∈ dom(t_b_e3) )
THEN
    SELECT
        StateSequence_c = fst ∧ State_a ∈ SF_a ∧ init_b ∈ dom(t_b_e3)
    THEN
        StateSequence_c := snd ||
        State_b := t_b_e3(init_b)
    WHEN
        StateSequence_c = snd ∧ State_b ∈ dom(t_b_e3)
    THEN
        State_b := t_b_e3(State_b)
    END

END
```

4.3.4 Choix

L'ensemble des ASTD de type choix est modélisé par $\langle |, name, l, r \rangle$, avec l et r représentant deux ASTD.

Un ASTD de type choix possède un état noté $\langle |_\circ, side, s \rangle$, tel que s représente l'état de l ou de r selon la valeur du second élément. Afin de représenter l'état d'un ASTD de type choix en B, l'ensemble B $CHOICES = \{leftS, rightS, none\}$ est défini.

L'état d'un ASTD de type choix est représenté en B par :

- Une variable nommée $stateChoice_name \in CHOICES$ qui vaut $leftS$ si l'état est left, $rightS$ si l'état est right et $none$ si l'état est $\perp$.
- $\eta(s)$ qui représente en B s, l'état courant de l'ASTD l ou r selon la valeur de l'état de l'ASTD modélisé par $stateChoice_name$.

L'état initial d'un ASTD de type choix est défini par la constante $init_Choice = none$.

L'état de l'ASTD *name* de type choix est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$\begin{array}{ll} & ((stateChoice_name = none) \Rightarrow [init(l)]final_l \wedge [init(r)]final_r) \\ \wedge & ((stateChoice_name = leftS) \Rightarrow [init(l)]final_l) \\ \wedge & ((stateChoice_name = rightS) \Rightarrow [init(r)]final_r) \end{array}$$

Si σ apparaît dans l ou dans r, nous disposons des traductions $op(\sigma)_l$ et $op(\sigma)_r$ respectivement. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_l$ et $op(\sigma)_r$. Soient

$$\begin{array}{lll} C1 & = & stateChoice_a = none \wedge [init(l)]preOf(op(\sigma)_l) \\ C2 & = & stateChoice_a = none \wedge [init(r)]preOf(op(\sigma)_r) \\ C3 & = & stateChoice_a = leftS \wedge preOf(op(\sigma)_l) \\ C4 & = & stateChoice_a = rightS \wedge preOf(op(\sigma)_r) \end{array}$$

Alors

$$\begin{array}{lll} op(\sigma) & = & \mathbf{PRE} \\ & & \quad C1 \vee C3 \vee \qquad \text{si } \sigma \text{ apparaît dans } l \\ & & \quad C2 \vee C4 \qquad \text{si } \sigma \text{ apparaît dans } r \\ & & \mathbf{THEN} \\ & & \qquad \mathbf{SELECT}\ C1\ \mathbf{THEN}\ stateChoice_a := leftS \parallel [init(l)]thenOf(op(\sigma)_l) \\ & & \qquad \mathbf{WHEN}\ C2\ \mathbf{THEN}\ stateChoice_a := rightS \parallel [init(r)]thenOf(op(\sigma)_r) \\ & & \qquad \mathbf{WHEN}\ C3\ \mathbf{THEN}\ thenOf(op(\sigma)_l) \\ & & \qquad \mathbf{WHEN}\ C4\ \mathbf{THEN}\ thenOf(op(\sigma)_r) \\ & & \qquad \mathbf{END} \end{array}$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type choix, soit Choice_1, Choice_2, Choice_3 et Choice_4 qui correspondent aux cas $C1$, $C2$, $C3$ et $C4$ respectivement.

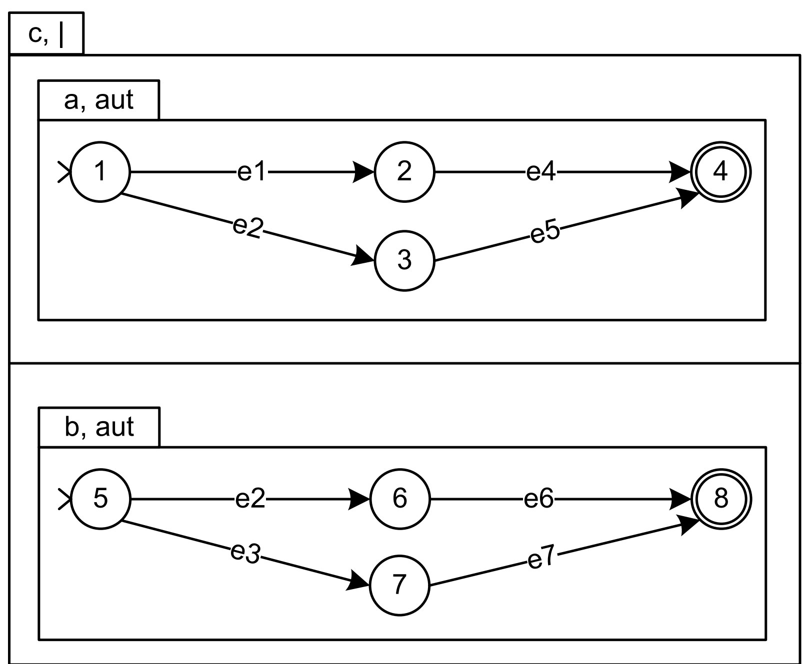


FIGURE 4.5 – Un exemple d'ASTD de type choix

Exemple de traduction

La Figure 4.5 présente un ASTD de type choix composé de deux ASTD de type automate.

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de type choix de la Figure 4.5. Nous avons appliqué les optimisations sur les **SELECT** décrites sur les traductions des ASTD de types automate et séquence.

```
MACHINE
    choix

SETS
    CHOICES = {leftS, rightS, none} ;
    AutState_a = {s1, s2, s3, s4} ;
    AutState_b = {s5, s6, s7, s8}

CONSTANTS
    init_a,
```

```
    init_b,
    init_Choice,
    t_a_e1,
    t_a_e2,
    t_a_e4,
    t_a_e5,
    t_b_e2,
    t_b_e3,
    t_b_e6,
    t_b_e7,
    DF_a, SF_a,
    DF_b, SF_b

PROPERTIES

    init_a = s1 ∧
    init_b = s5 ∧
    init_Choice = none ∧
    t_a_e1 = {s1 |-> s2} ∧
    t_a_e2 = {s1 |-> s3} ∧
    t_a_e4 = {s2 |-> s4} ∧
    t_a_e5 = {s3 |-> s4} ∧
    t_b_e2 = {s5 |-> s6} ∧
    t_b_e3 = {s5 |-> s7} ∧
    t_b_e6 = {s6 |-> s8} ∧
    t_b_e7 = {s7 |-> s8} ∧
    DF_a ⊆  AutState_a ∧
    SF_a ⊆  AutState_a ∧
    DF_a ∩ SF_a = {} ∧
    DF_a = {} ∧
    SF_a = {s4} ∧
    DF_b ⊆  AutState_b ∧
    SF_b ⊆  AutState_b ∧
    DF_b ∩ SF_b = {} ∧
    DF_b = {} ∧
    SF_b = {s8}

VARIABLES
    State_a,
```

```
    State_b,
    StateChoice_c

INVARIANT
    State_a ∈ AutState_a ∧
    State_b ∈ AutState_b ∧
    StateChoice_c ∈ CHOICES

INITIALISATION
    State_a := init_a ||
    State_b := init_b ||
    StateChoice_c := init_Choice

OPERATIONS
    e1 =
    PRE
        (StateChoice_c = none ∧ init_a ∈ dom(t_a_e1))
        ∨ (StateChoice_c = leftS ∧ State_a ∈ dom(t_a_e1))
    THEN
        SELECT
            StateChoice_c = none ∧ init_a ∈ dom(t_a_e1)
        THEN
            StateChoice_c := leftS|| State_a := t_a_e1(init_a)
        WHEN
            StateChoice_c = leftS ∧ State_a ∈ dom(t_a_e1)
        THEN
            State_a := t_a_e1(State_a)
        END
    END ;

    e2 =
    PRE
        (StateChoice_c = none ∧ init_a ∈ dom(t_a_e2)) or
        (StateChoice_c = none ∧ init_b ∈ dom(t_b_e2)) or
        (StateChoice_c = leftS  ∧ State_a ∈ dom(t_a_e2)) or
        (StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e2))
    THEN
        SELECT
            StateChoice_c = none ∧ init_a ∈ dom(t_a_e2)
```

```
    THEN
        StateChoice_c := leftS || State_a := t_a_e2(init_a)
    WHEN
        StateChoice_c = none ∧ init_b ∈ dom(t_b_e2)
    THEN
        StateChoice_c := rightS || State_b := t_b_e2(init_b)
    WHEN
        StateChoice_c = leftS ∧ State_a ∈ dom(t_a_e2)
    THEN
        State_a := t_a_e2(State_a)
    WHEN
        StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e2)
    THEN
        State_b := t_b_e2(State_b)
    END
END;

e4 =
PRE
    (StateChoice_c = none ∧ init_a ∈ dom(t_a_e4))
    ∨ (StateChoice_c = leftS  ∧ State_a ∈ dom(t_a_e4))
THEN
    SELECT
        StateChoice_c = none ∧ init_a ∈ dom(t_a_e4)
    THEN
        StateChoice_c := leftS || State_a := t_a_e4(init_a)
    WHEN
        StateChoice_c = leftS  ∧ State_a ∈ dom(t_a_e4)
    THEN
        State_a := t_a_e4(State_a)
    END
END ;

e5 =
PRE
    (StateChoice_c = none ∧ init_a ∈ dom(t_a_e5))
    ∨ (StateChoice_c = leftS  ∧ State_a ∈ dom(t_a_e5))
THEN
    SELECT
```

```
        StateChoice_c = none ∧ init_a ∈ dom(t_a_e5)
    THEN
        StateChoice_c := leftS || State_a := t_a_e5(init_a)
    WHEN
        StateChoice_c = leftS  ∧ State_a ∈ dom(t_a_e5)
    THEN
        State_a := t_a_e5(State_a)
    END
END ;

e3 =
PRE
    (StateChoice_c = none ∧ init_b ∈ dom(t_b_e3))
    ∨ (StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e3))
THEN
    SELECT
        StateChoice_c = none ∧ init_b ∈ dom(t_b_e3)
    THEN
        StateChoice_c := rightS|| State_b := t_b_e3(init_b)
    WHEN
        StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e3)
    THEN
        State_b := t_b_e3(State_b)
    END
END ;

e6 =
PRE
    (StateChoice_c = none ∧ init_b ∈ dom(t_b_e6))
    ∨ (StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e6))
THEN
    SELECT
        StateChoice_c = none ∧ init_b ∈ dom(t_b_e6)
    THEN
        StateChoice_c := rightS|| State_b := t_b_e6(init_b)
    WHEN
        StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e6)
    THEN
        State_b := t_b_e6(State_b)
```

```
        END
    END ;

    e7 =
    PRE
        (StateChoice_c = none ∧ init_b ∈ dom(t_b_e7))
        ∨ (StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e7))
    THEN
        SELECT
            StateChoice_c = none ∧ init_b ∈ dom(t_b_e7)
        THEN
            StateChoice_c := rightS|| State_b := t_b_e7(init_b)
        WHEN
            StateChoice_c = rightS ∧ State_b ∈ dom(t_b_e7)
        THEN
            State_b := t_b_e7(State_b)
        END
    END

END
```

Statistiques Le typage de la machine B présentée a été validé (*typecheck*) par l'atelier B et l'outil génère 19 obligations de preuves toutes prouvées automatiquement.

4.3.5 Fermeture de Kleene

L'ensemble des ASTD de type fermeture de Kleene est modélisé par $\langle \star, name, b \rangle$. avec b représentant un ASTD. Un ASTD de type fermeture de Kleene possède un état noté $\langle \star_\circ, started?, s \rangle$, tel que $started? \in \mathsf{Boolean}$ indique si l'exécution de b a commencé. Afin de représenter l'état d'un ASTD de type fermeture de Kleene en B, l'ensemble B $KLEENE = \{started, notstarted\}$ est défini.

L'état d'un ASTD de type fermeture de Kleene est représenté en B par :

- Une variable nommée $stateKleene_name \in KLEENE$ qui vaut $started$ si $started? = true$, $notstarted$ si $started? = false$.
- $\eta(s)$ qui représente en B s, l'état courant de l'ASTD b.

L'état initial d'un ASTD de type fermeture de Kleene est défini par la constante $init_Kleene = notstarted$.

L'état de l'ASTD *name* de type fermeture de Kleene est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$StateKleene_name = notstarted \vee final_b$$

Nous disposons de la traduction $op(\sigma)_b$ de l'opération correspondant à σ dans b. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_b$. Soient

$$\begin{aligned} C1 &= (final_b \vee stateKleene_name = notstarted) \wedge [init(b)]preOf(op(\sigma)_b) \\ C2 &= preOf(op(\sigma)_b) \end{aligned}$$

Alors

$$\begin{aligned} op(\sigma) = \; & \textbf{PRE} \\ & \quad C1 \vee C2 \\ & \textbf{THEN} \\ & \quad stateKleene_name := started \parallel \\ & \qquad \textbf{SELECT } C1 \textbf{ THEN } [init(b)]thenOf(op(\sigma)_b) \\ & \qquad \textbf{WHEN } C2 \textbf{ THEN } thenOf(op(\sigma)_b) \\ & \qquad \textbf{END} \end{aligned}$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type fermeture de Kleene, soit $\mathsf{Closure}_1$ et $\mathsf{Closure}_2$ correspondant respectivement à $C1$ et $C2$.

Exemple de traduction

La Figure 4.6 présente un ASTD de type séquence composé de deux ASTD de type fermeture de Kleene appliqués sur des automates.

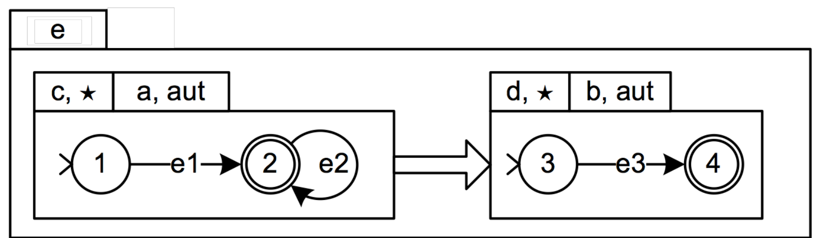


FIGURE 4.6 – Un exemple d'ASTD de type séquence contenant deux ASTD de type fermeture de Kleene

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de la Figure 4.6. Cet ASTD contient deux sous-ASTD de type fermeture de Kleene. Nous avons réduit les prédicats contenant des parties trivialement vraies ou fausses afin de simplifier la lecture. Nous avons également appliqué les optimisations sur les **SELECT** décrites sur les traductions précédentes.

```
MACHINE
    seq_e

SETS
    KLEENE = {started, notstarted} ;
    SEQUENCE = {fst, snd} ;
    AutState_a = {s1, s2} ;
    AutState_b = {s3, s4}

CONSTANTS
    init_a,
    init_b,
    init_Sequence,
    init_Kleene,
    t_a_e1,
    t_a_e2,
    t_b_e3,
    DF_a, SF_a,
    DF_b, SF_b

PROPERTIES
```

```
    init_a = s1 ∧
    init_b = s3 ∧
    init_Sequence = fst ∧
    init_Kleene = notstarted ∧
    t_a_e1 = {s1 |-> s2} ∧
    t_a_e2 = {s2 |-> s2} ∧
    t_b_e3 = {s3 |-> s4} ∧
    DF_a ⊆  AutState_a ∧
    SF_a ⊆  AutState_a ∧
    DF_a ∩ SF_a = {} ∧
    DF_a = {} ∧
    SF_a = {s2} ∧
    DF_b ⊆  AutState_b ∧
    SF_b ⊆  AutState_b ∧
    DF_b ∩ SF_b = {} ∧
    DF_b = {} ∧
    SF_b = {s4}

VARIABLES
    State_a,
    State_b,
    StateKleene_c,
    StateKleene_d,
    StateSequence_e


INVARIANT
    State_a ∈ AutState_a ∧
    State_b ∈ AutState_b ∧
    StateKleene_c ∈ KLEENE ∧
    StateKleene_d ∈ KLEENE ∧
    StateSequence_e ∈ SEQUENCE

INITIALISATION
    State_a := init_a ||
    State_b := init_b ||
    StateKleene_c := init_Kleene ||
    StateKleene_d := init_Kleene ||
    StateSequence_e := init_Sequence
```

```
OPERATIONS

    e1 =
    PRE
        StateSequence_e = fst ∧
        (((State_a ∈ SF_a ∨ StateKleene_c = notstarted) ∧
             init_a ∈ dom(t_a_e1)) or
        (State_a ∈ dom(t_a_e1)))
    THEN
        StateKleene_c := started ||
        SELECT
            (State_a ∈ SF_a ∨ StateKleene_c = notstarted) ∧
             init_a ∈ dom(t_a_e1)
        THEN
            State_a := t_a_e1(init_a)
        WHEN
            State_a ∈ dom(t_a_e1)
        THEN
            State_a := t_a_e1(State_a)
        END
    END;

    e2 =
    PRE
        StateSequence_e = fst ∧
        (((State_a ∈ SF_a ∨ StateKleene_c = notstarted)
            ∧ init_a ∈ dom(t_a_e2)) or
        (State_a ∈ dom(t_a_e2)))
    THEN
        StateKleene_c := started ||
        SELECT
            (State_a ∈ SF_a ∨ StateKleene_c = notstarted) ∧
            init_a ∈ dom(t_a_e2)
        THEN
            State_a := t_a_e2(init_a)
        WHEN
            State_a ∈ dom(t_a_e2)
        THEN
```

```
        State_a := t_a_e2(State_a)
    END
END;

e3 =
PRE
    (StateSequence_e = fst ∧ (StateKleene_c = notstarted ∨ State_a ∈ SF_a)
        ∧ init_b ∈ dom(t_b_e3)) or
    StateSequence_e = snd ∧ (
        ((State_b ∈ SF_b ∨ StateKleene_d = notstarted)
        ∧ init_b ∈ dom(t_b_e3)) or
        (State_b ∈ dom(t_b_e3)))
THEN
    SELECT
        StateSequence_e = fst ∧
        (StateKleene_c = notstarted ∨ State_a ∈ SF_a) ∧
        init_b ∈ dom(t_b_e3)
    THEN
        StateSequence_e := snd ||
        StateKleene_d := started ||
        State_b := t_b_e3(init_b)
    WHEN
        StateSequence_e = snd ∧ (
            ((State_b ∈ SF_b ∨ StateKleene_d = notstarted)
            ∧ init_b ∈ dom(t_b_e3)) or
            (State_b ∈ dom(t_b_e3)))
    THEN
        StateKleene_d := started ||
        SELECT
            (State_b ∈ SF_b ∨ StateKleene_d = notstarted)
            ∧ init_b ∈ dom(t_b_e3)
        THEN
            State_b := t_b_e3(init_b)
        WHEN
            State_b ∈ dom(t_b_e3)
        THEN
            State_b := t_b_e3(State_b)
        END
    END
```

```
    END

END
```

Statistiques Le typage de la machine B présentée a été validé (*typecheck*) par l'atelier B et l'outil génère 21 obligations de preuves toutes prouvées automatiquement.

4.3.6 Synchronisation paramétrée

L'ensemble des ASTD de type synchronisation paramétrée est modélisé par

$$\langle |[]|, name, \Delta, l, r\rangle$$

avec Δ représentant l'ensemble des étiquettes des évènements synchronisés. Les termes l et r représentant les deux ASTD à synchroniser.

Un ASTD de type synchronisation paramétrée possède un état noté $\langle |[]|_\circ, s_l, s_r\rangle$, tel que s_l représente l'état de l et s_r représente l'état de r.

L'état d'un ASTD de type synchronisation paramétrée est représenté en B par :

- $\eta(s_l)$ qui représente en B s_l, l'état courant de l'ASTD l.
- $\eta(s_r)$ qui représente en B s_r, l'état courant de l'ASTD r.

L'état initial d'un ASTD de type synchronisation paramétrée est défini par l'état initial de ses deux opérandes.

L'état de l'ASTD *name* de type synchronisation paramétrée est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$final_l \wedge final_r$$

Nous disposons des traductions $op(\sigma)_l$ et $op(\sigma)_r$ de l'opération correspondant à σ dans l et r respectivement. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_l$ et $op(\sigma)_r$. Il y a deux façons de combiner les opérations : si $\sigma \in \Delta$ ou non.

Si $\sigma \in \Delta$ alors :

$$
\begin{aligned}
op(\sigma) \quad \triangleq \quad & \mathbf{PRE}\ \mathit{preOf}(op(\sigma)_l) \wedge \mathit{preOf}(op(\sigma)_r)\ \mathbf{THEN} \\
& \quad \mathit{thenOf}(op(\sigma)_l) \parallel \mathit{thenOf}(op(\sigma)_r) \\
& \mathbf{END}
\end{aligned}
$$

Nous considérons que si l'une des opérations n'existe pas, elle vaut **PRE** $false$ **THEN** $skip$. Ainsi, en cas de synchronisation, il y a blocage car la précondition est toujours fausse.

Si $\sigma \notin \Delta$ et que σ apparaît des deux côtés de l'ASTD alors

$$
\begin{aligned}
op(\sigma) \quad \triangleq \quad & \mathbf{PRE}\ \mathit{preOf}(op(\sigma)_l) \vee \mathit{preOf}(op(\sigma)_r)\ \mathbf{THEN} \\
& \quad \mathbf{SELECT}\ \mathit{preOf}(op(\sigma)_l)\ \mathbf{THEN}\ \mathit{thenOf}(op(\sigma)_l) \\
& \quad \mathbf{WHEN}\ \mathit{preOf}(op(\sigma)_r)\ \mathbf{THEN}\ \mathit{thenOf}(op(\sigma)_r) \\
& \quad \mathbf{END\ END}
\end{aligned}
$$

Si $\sigma \notin \Delta$ et que σ n'apparaît pas des deux côtés de l'ASTD alors nous recopions simplement l'opération telle qu'elle apparaît dans l ou r.

Dans tous les cas, l'opération B calculée correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type synchronisation, soit $\mathsf{Synchronization}_1$, $\mathsf{Synchronization}_2$ et $\mathsf{Synchronization}_3$.

Exemple de traduction

La Figure 4.7 présente un ASTD de type synchronisation paramétrée incluant deux ASTD de type automate.

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de type synchronization de la Figure 4.7.

```
MACHINE
    csync

SETS
    AutState_a = {s1,s2,s3,s4} ;
```

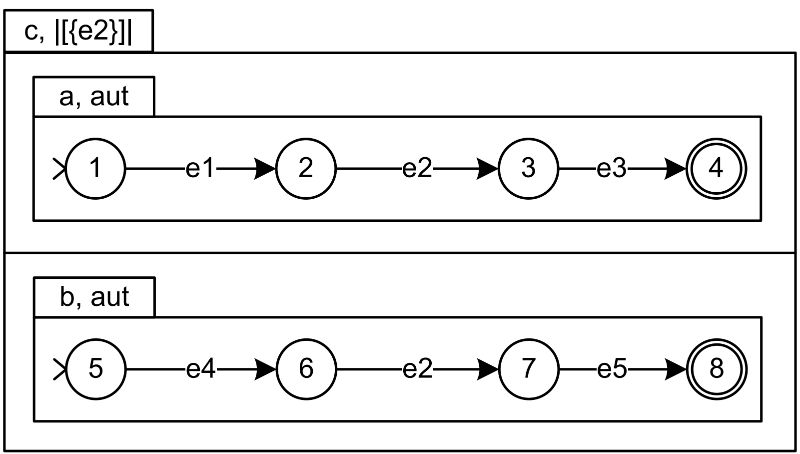


FIGURE 4.7 – Un ASTD de type synchronisation incluant deux ASTD de type automate

```
    AutState_b = {s5,s6,s7,s8}

CONSTANTS
    init_a,
    t_a_e1,
    t_a_e2,
    t_a_e3,
    DF_a, SF_a,
    init_b,
    t_b_e4,
    t_b_e2,
    t_b_e5,
    DF_b, SF_b

PROPERTIES
    DF_a ⊆  AutState_a ∧
    SF_a ⊆  AutState_a ∧
    DF_a ∩ SF_a = {} ∧
    DF_a = {} ∧
    SF_a = {s4} ∧
    init_a = s1 ∧
    t_a_e1 = {s1 |-> s2} ∧
    t_a_e2 = {s2 |-> s3} ∧
```

```
    t_a_e3 = {s3 |-> s4} ∧
    DF_b ⊆  AutState_b ∧
    SF_b ⊆  AutState_b ∧
    DF_b ∩ SF_b = {} ∧
    DF_b = {} ∧
    SF_b = {s8} ∧
    init_b = s5 ∧
    t_b_e4 = {s5 |-> s6} ∧
    t_b_e2 = {s6 |-> s7} ∧
    t_b_e5 = {s7 |-> s8}

VARIABLES
    State_a, State_b

INVARIANT
    State_a ∈ AutState_a ∧
    State_b ∈ AutState_b

INITIALISATION
    State_a := init_a ||
    State_b := init_b

OPERATIONS
    e1 =
    PRE (State_a ∈ dom ( t_a_e1 ))
    THEN
        SELECT
            State_a ∈ dom ( t_a_e1 )
        THEN
            State_a := t_a_e1 ( State_a )
        END
    END ;

    e3 =
    PRE (State_a ∈ dom ( t_a_e3 ))
    THEN
        SELECT
            State_a ∈ dom ( t_a_e3 )
        THEN
```

```
        State_a := t_a_e3 ( State_a )
    END
END ;

e4 =
PRE (State_b ∈ dom ( t_b_e4 ))
THEN
    SELECT
        State_b ∈ dom ( t_b_e4 )
    THEN
        State_b := t_b_e4 ( State_b )
    END
END ;


e5 =
PRE (State_b ∈ dom ( t_b_e5 ))
THEN
    SELECT
        State_b ∈ dom ( t_b_e5 )
    THEN
        State_b := t_b_e5 ( State_b )
    END
END ;

e2 =
PRE (State_a ∈ dom ( t_a_e2 )) ∧ (State_b ∈ dom ( t_b_e2 ))
THEN
    SELECT
        State_a ∈ dom ( t_a_e2 )
    THEN
        State_a := t_a_e2 ( State_a )
    END
    ||
    SELECT
        State_b ∈ dom ( t_b_e2 )
    THEN
        State_b := t_b_e2 ( State_b )
    END
```

```
    END

END
```

Statistiques et optimisations Le typage de la machine B présentée a été validé (*type-check*) par l'atelier B et l'outil génère huit obligations de preuves toutes prouvées automatiquement.

De la même manière que pour les ASTD de type automate et séquence, une optimisation des substitutions générées est possible. Par exemple pour l'opération $e2$, on obtiendrait :

```
e2 =
PRE (State_a ∈ dom ( t_a_e2 )) ∧ (State_b ∈ dom ( t_b_e2 ))
THEN
        State_a := t_a_e2 ( State_a )
    ||
        State_b := t_b_e2 ( State_b )
END
```

4.3.7 Choix quantifié

L'ensemble des ASTD de type choix quantifié est modélisé par $\langle |:, name, x, T, b\rangle$, avec $x \in \mathsf{Var}$ représentant une variable, T un type et b un ASTD. Nous créons un ensemble B nommé T_name pour représenter T. Dans l'opération utilisant la variable quantifiée, nous utilisons x_name au lieu de x.

Un ASTD de type choix quantifié possède un état noté $\langle |:_\circ, [\bot \mid v], s\rangle$, tel que $v \in T$ indique la valeur de la variable quantifiée si elle a été choisie, $\bot$ sinon et s représente l'état de b. Afin de représenter l'état d'un ASTD de type choix quantifié en B, l'ensemble B $QCHOICE = \{notchosen, chosen\}$ est défini.

L'état d'un ASTD de type choix quantifié est représenté en B par :
- Une variable nommée $stateQchoiceMade_name \in QCHOICE$ qui vaut $notchosen$ si $v = \bot$ est dans l'état, $chosen$ sinon.
- Une variable nommée $stateQchoiceValue_name \in T_name$ qui vaut l'équivalent de v en B. Si la valeur v n'a pas encore été choisie, une valeur est choisie de façon non déterministe.

– $\eta(s)$ qui représente en B s, l'état courant de l'ASTD b.

L'état initial d'un ASTD de type choix quantifié est défini par la constante $init_QChoice = notchosen$.

L'état de l'ASTD *name* de type choix quantifié est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$((stateQchoiceMade_name = notchosen) \Rightarrow \exists v \in T_name : [init(b)\ ;\ x_name := v] final_b)$$
$$\wedge((stateQchoiceMade_name = chosen) \Rightarrow [x_name := stateQchoiceValue_name] final_b)$$

Nous disposons de la traduction $op(\sigma)_b$ de l'opération correspondant à σ dans b. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_b$.

$$\begin{aligned} op(\sigma) \quad \triangleq \quad & \textbf{PRE } stateQchoiceMade_a = notchosen \wedge [init(b)]preOf(op(\sigma)_b) \\ & \vee x_name = stateQchoiceValue \wedge preOf(op(\sigma)_b) \textbf{ THEN} \\ & \textbf{SELECT } stateQchoiceMade_a = notchosen \\ & \quad \textbf{THEN } (stateQchoiceMade := chosen \parallel stateQchoiceValue := x) \\ & \qquad \parallel [init(b)]thenOf(op(\sigma)_b) \\ & \textbf{WHEN } x_name = stateQchoiceValue \\ & \quad \textbf{THEN } thenOf(op(\sigma)_b) \\ & \textbf{END END} \end{aligned}$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type choix quantifié, soit $\mathsf{QChoice}_1$ et $\mathsf{QChoice}_2$.

Un exemple de traduction d'ASTD de type choix quantifié est présenté en section B.1 au sein d'une étude de cas comportant plusieurs types d'ASTD.

4.3.8 Synchronisation quantifiée

L'ensemble des ASTD de type synchronisation quantifiée est modélisé par l'ensemble $\langle |[]|:, name, x, T, \Delta, b\rangle$, avec $x \in \mathsf{Var}$ représentant une variable, T un type et b un ASTD. Le symbole Δ représente l'ensemble des étiquettes des évènements synchronisés. Nous créons

un ensemble B nommé T_name pour représenter T. Dans l'opération utilisant la variable quantifiée, nous utilisons x_name au lieu de x.

Un ASTD de type synchronisation quantifiée possède un état noté $\langle |[]|:_\circ, f \rangle$, tel que $f \in T \rightarrow$ STATE indique l'état de b associé à la valeur de la variable quantifiée.

L'état d'un ASTD de type synchronisation quantifiée est modélisé en B en modifiant le type de toutes les variables d'état de b. Pour chaque variable d'état, si son type est U, alors il devient $T_name \rightarrow U$.

L'état initial d'un ASTD de type synchronisation quantifiée est défini par l'état initial de toutes les variables d'état de b modifiées qui est le produit cartésien de T_name par l'ancienne valeur de la variable à l'état initial. Ainsi, toutes les constantes définissant l'état initial des variables doivent être adaptées.

L'état de l'ASTD *name* de type synchronisation quantifiée est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$\forall v.(v \in T_name \Rightarrow [x_name := v] final_b())$$

Nous disposons de la traduction $op(\sigma)_b$ de l'opération correspondant à σ dans b. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_b$. Il y a trois façons de combiner les opérations : si $\sigma \in \Delta$ ou non et selon la présence de la variable quantifiée dans les paramètres ou non.

Si $\sigma \in \Delta$ et que la variable de quantification n'est pas un paramètre de l'évènement :

$$\begin{aligned} op(\sigma) \ \triangleq\ & \textbf{PRE}\ \forall v.(v \in T_name \wedge [x_name := v] preOf(op(\sigma)_b)) \\ & \textbf{THEN}\ modifiedThenOf(op(\sigma)_b)\ \textbf{END} \end{aligned}$$

La fonction de traduction *modifiedThenOf* calcule une nouvelle substitution en fonction de quelques règles. Ces règles modifient la manière dont les variables d'état sont lues et écrites. Soit $P(x_name)$ un prédicat dans lequel une ou plusieurs variables d'état sont lues, comme dans les **IF** ou **PRE**. Le nouveau prédicat remplaçant $P(x_name)$ est $\forall v(v \in T_name \wedge P(v))$. Soit $S \triangleq sv(x_name) := newValue(x_name)$ une substitution d'affectation dans laquelle la variable d'état sv prend une nouvelle valeur notée $newValue(x_name)$. La nouvelle substitution qui remplace S est $sv := \lambda v.(v \in T_name | newValue(v))$. Notons qu'un évènement peut être synchronisé et avoir la variable de quantification comme para-

mètre. Ce genre de spécification conduit systématiquement à un blocage et n'est pas une spécification souhaitable dans un système. Nous supposons que ce cas n'est pas présent dans nos traductions.

Si $\sigma \notin \Delta$ et que la variable quantifiée n'est pas un paramètre de l'évènement alors :

$$\begin{aligned} op(\sigma) \ \triangleq \ & \textbf{PRE } \exists v.(v \in T_name \wedge [x_name := v] preOf(op(\sigma)_b)) \textbf{ THEN} \\ & \textbf{ANY } v \textbf{ WHERE } v \in T_name \wedge [x_name := v] preOf(op(\sigma)_b) \\ & \textbf{THEN } [x_name := v] thenOf(op(\sigma)_b) \textbf{ END END} \end{aligned}$$

Si $\sigma \notin \Delta$ et que la variable quantifiée est un paramètre de l'évènement alors nous ne modifions pas l'opération B $op(\sigma)$: nous reprenons l'opération $op(\sigma)_b$, et nous ne faisons qu'appliquer la modification du type des variables d'état présentée précédemment.

Ces opérations B correspondent exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type synchronisation quantifiée (soit $\mathsf{QSynchronization}_1$ et $\mathsf{QSynchronization}_2$). En effet, si l'on étudie la structure arborescente créée par les substitutions conditionnelles **IF**, on retrouve la conclusion de $\mathsf{QSynchronization}_1$ dans le cas $\sigma \notin \Delta$; la conclusion de $\mathsf{QSynchronization}_2$ dans le cas $\sigma \in \Delta$. Les prémisses des règles d'inférence sont les conditions des substitutions conditionnelles.

Exemple de traduction

La Figure 4.8 présente un ASTD de type synchronisation quantifiée composé d'un ASTD de type automate. Les trois cas détaillés dans cette section sont présents. Le terme $e1(x)$ représente un évènement non synchronisé dont la variable de quantification apparaît dans les paramètres. Le terme $e2$ est un évènement synchronisé. Le terme $e3$ est un évènement non synchronisé qui n'utilise pas la variable de quantification.

Traduction La machine B décrite ci-dessous est la traduction de l'ASTD de type synchronisation quantifiée de la Figure 4.8. Nous avons réduit les prédicats contenant des parties trivialement vraies ou fausses afin de simplifier la lecture. Nous avons également appliqué les optimisations sur les **SELECT** décrites sur les traductions précédentes.

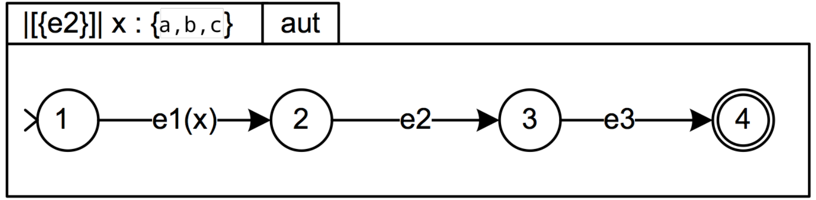


FIGURE 4.8 – Un exemple d'ASTD de type synchronisation quantifiée

```
MACHINE
    qsynch

SETS
    AutState_aut = {s1, s2, s3, s4} ;
    T_synch = {aa, bb, cc}

CONSTANTS
    init_aut,
    t_aut_e1,
    t_aut_e2,
    t_aut_e3,
    DF_aut, SF_aut

PROPERTIES
    init_aut = s1 ∧
    t_aut_e1 = {s1 |-> s2} ∧
    t_aut_e2 = {s2 |-> s3} ∧
    t_aut_e3 = {s3 |-> s4} ∧
    DF_aut ⊆  AutState_aut ∧
    SF_aut ⊆  AutState_aut ∧
    DF_aut ∩ SF_aut = {} ∧
    DF_aut = {} ∧
    SF_aut = {s4}

VARIABLES
    State_aut

INVARIANT
    State_aut ∈ T_synch --> AutState_aut
```

```
INITIALISATION
    State_aut :=  T_synch x {init_aut}

OPERATIONS

    e1(x_synch) =
    PRE
        x_synch : T_synch ∧ State_aut(x_synch) ∈ dom ( t_aut_e1 )
    THEN
        State_aut(x_synch) := t_aut_e1 ( State_aut(x_synch) )
    END;

    e2 =
    PRE
        ∀ vv.( vv ∈ T_synch => State_aut(vv) ∈ dom ( t_aut_e2 ))
    THEN
        State_aut := lamda vv.( vv ∈ T_synch | t_aut_e2 ( State_aut(vv)) )
    END;

    e3 =
    PRE
        ∃ vv.( vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ))
    THEN
        ANY vv WHERE vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ) THEN
            State_aut(vv) := t_aut_e3 ( State_aut(vv) )
        END
    END

END
```

Statistiques Le typage de la machine B présentée a été validé (*typecheck*) par l'atelier B et l'outil génère huit obligations de preuves toutes prouvées automatiquement.

4.3.9 Garde

L'ensemble des ASTD de type garde est modélisé par $\langle \Rightarrow, name, g, b\rangle$, avec g un prédicat et b représentant un ASTD. Un ASTD de type garde possède un état noté $\langle \Rightarrow_\circ, started, s\rangle$, tel que $started \in$ Boolean indique si la garde a déjà été vérifiée ou non et s représente l'état de b. Afin de représenter l'état d'un ASTD de type garde en B, l'ensemble B $GUARD = \{checked, notchecked\}$ est défini.

L'état d'un ASTD de type garde est représenté en B par :

- Une variable nommée $stateGuard_name \in GUARD$ qui vaut $checked$ si $started = true$, $notchecked$ si $started = false$.
- $\eta(s)$ qui représente en B s, l'état courant de l'ASTD b.

L'état initial d'un ASTD de type garde est défini par $init_Guard = notchecked$.

L'état de l'ASTD $name$ de type garde est dit final si et seulement si le prédicat $final_name$ défini ci-dessous est vrai :

$$(StateGuard_name = notexecuted \Rightarrow [init(b)]final_b)$$

$$\wedge(StateGuard_name = notexecuted \Rightarrow final_b)$$

Nous disposons de la traduction $op(\sigma)_b$ de l'opération correspondant à σ dans b. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_b$.

$$\begin{array}{lll} op(\sigma) & \stackrel{\Delta}{=} & \textbf{PRE}\ (g \wedge stateGuard_a = notchecked \wedge [init(b)]preOf(op(\sigma)_b)) \vee \\ & & preOf(op(\sigma)_b)\ \textbf{THEN} \\ & & \quad \textbf{SELECT}\ g \wedge stateGuard_a = notchecked \wedge [init(b)]preOf(op(\sigma)_b) \\ & & \quad\quad \textbf{THEN}\ stateGuard_a := checked \parallel [init(b)]thenOf(op(\sigma)_b) \\ & & \quad \textbf{WHEN}\ preOf(op(\sigma)_b) \\ & & \quad\quad \textbf{THEN}\ thenOf(op(\sigma)_b)\ \textbf{END END} \end{array}$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type garde, soit Guard_1 et Guard_2.

4.3.10 Appel d'ASTD

L'ensemble des ASTD de type appel est modélisé par $\langle \mathsf{cal}, name, P(\vec{v})\rangle$, avec P qui représente une référence à un ASTD tel que $P(\vec{x} : \vec{T}) \triangleq b$ et pour chaque $v_i \in \vec{v}$, nous avons $v_i \in T_i$. Un ASTD de type garde possède un état noté $\langle \mathsf{cal}_\circ, [\bot \mid s]\rangle$, tel que s représente l'état de b. Afin de représenter l'état d'un ASTD de type appel en B, l'ensemble B $CALL = \{called, notcalled\}$ est défini.

L'état d'un ASTD de type appel est représenté en B par :

- Une variable nommée $stateCall_name \in CALL$ qui vaut $called$ si l'ASTD a déjà été appelé, $notcalled$ sinon, et que l'état est $\bot$.
- $\eta(s)$ qui représente en B s, l'état courant de l'ASTD b.

L'état initial d'un ASTD de type appel est défini par la constante $init_Call = notcalled$.

L'état de l'ASTD *name* de type appel est dit final si et seulement si le prédicat défini ci-dessous est vrai :

$$(StateCall_name = notcalled \Rightarrow [\vec{x} := \vec{v}; init(b)]final_b)$$

$$\wedge(StateCall_name = called \Rightarrow [\vec{x} := \vec{v}]final_b)$$

Nous disposons de la traduction $op(\sigma)_b$ de l'opération correspondant à σ dans b. Nous pouvons construire $op(\sigma)$ à partir de $op(\sigma)_b$.

$$\begin{aligned} op(\sigma) \triangleq\ & \textbf{PRE}\ (stateCall_a = notcalled \wedge [init(b)]preOf(op(\sigma)_b)) \vee \\ & preOf(op(\sigma)_b)\ \textbf{THEN} \\ & \quad \textbf{SELECT}\ stateCall_a = notcalled \wedge [init(b)]preOf(op(\sigma)_b) \\ & \quad\quad \textbf{THEN}\ stateCall_a = called \parallel [init(b)]thenOf(op(\sigma)_b) \\ & \quad \textbf{WHEN}\ preOf(op(\sigma)_b) \\ & \quad\quad \textbf{THEN}\ thenOf(op(\sigma)_b)\ \textbf{END END} \end{aligned}$$

Cette opération B correspond exactement aux conditions ainsi qu'aux substitutions dans les règles d'inférence décrivant la sémantique d'exécution des ASTD de type garde, soit $\mathsf{ASTDCall}_1$ et $\mathsf{ASTDCall}_2$.

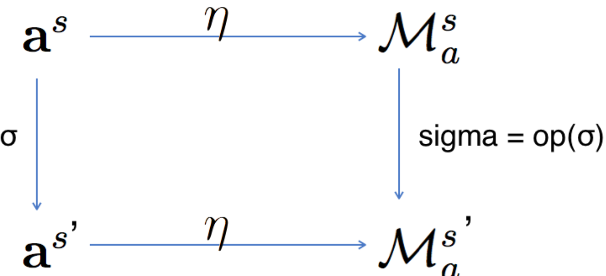


FIGURE 4.9 – Preuve de la traduction en utilisant la fonction η

Un exemple de traduction d'ASTD de type appel d'ASTD est présenté en section B.1 au sein d'une étude de cas comportant plusieurs types d'ASTD.

4.4 Principes de la preuve de traduction

Dans cette section, nous souhaitons prouver que la traduction d'un ASTD en une machine B respecte la sémantique des ASTD en comparant l'état de l'ASTD avant et après l'exécution d'un évènement avec les variables d'état avant et après l'exécution de l'opération correspondante dans la machine B. La Figure 4.9 présente la méthodologie de la preuve. Soient **a** un ASTD qui peut exécuter $\sigma(\vec{x})$ et $\mathbf{a}^s$ l'état de cet ASTD. Soit $\mathbf{a}^{s\prime}$ l'état de l'ASTD **a** résultant de l'exécution de $\sigma(\vec{x})$. Soient $\mathscr{M}_a = \theta(\mathbf{a})$ la machine résultant de la traduction de l'ASTD **a**, $\mathscr{M}_a^s = \eta(\mathbf{a}^s)$ et $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$ l'ensemble des variables d'état de la machine B avant et après l'exécution de l'opération $op(\sigma)$. Nous souhaitons montrer que $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$.

Afin de pouvoir prouver cette égalité, nous considérons $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$, l'arbre de preuve d'exécution de $\sigma(\vec{x})$ par **a** à l'état $\mathbf{a}^s$. Le preuve de l'égalité $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$ se fait par induction sur la profondeur de $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$. La traduction des ASTD en B a été construite pour que cette preuve soit simplifiée. C'est la grande proximité entre les règles d'inférence des ASTD et les substitutions B qui simplifie cette preuve.

4.4.1 Cas de base

Si $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$ est de profondeur 1, alors l'arbre de preuve d'exécution n'est composé que d'un axiome. Seules cinq règles d'inférence de la sémantique d'exécution des ASTD sont

des axiomes. Nous faisons ici une disjonction des cas selon la règle utilisée. Les règles sont nommées selon le type d'ASTD et sont disponible en Appendix C.

Axiome aut_1

Puisque $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$ est uniquement composé de la règle aut_1, nous savons que la transition exécutée est une transition de **a**, et qu'elle est locale.

Avant l'exécution de $\sigma(\vec{x})$, la variable B $state_a$ est telle que $state_a = n1$ car l'état courant de l'ASTD est $(\mathsf{aut}_\circ, n_1, h, s)$. Selon la prémisse de aut_1, il existe une transition étiquetée comme $\sigma(\vec{x})$ telle que sa source soit $n1$ et sa destination $n2$. De ce fait, $(n1 \mapsto n2) \in t_a_sigma$. L'opération $op(\sigma)$ est donc une susbtitution de sélection avec **WHEN** C **THEN** T, avec

$$T = state_a := t_a_sigma(state_a) \parallel initAll_n2$$

$$C = state_a \in dom(t_a_sigma) \wedge phi$$

et phi un prédicat vérifiant que la garde de la transition et ses paramètres sont valides. De ce fait, l'état courant de la machine est tel que $op(\sigma)$ peut s'exécuter et que la substitution T est appliquée sur les variables d'état. Ainsi, après l'exécution, $state_a = n2$ et si $n2$ n'est pas élémentaire, $state_n2 = init_n2$. Les autres variables d'état ne sont pas modifiées, y compris les éventuelles variables d'état de $n1$.

D'après la règle d'inférence aut_1, l'état de l'ASTD après l'exécution de $\sigma(\vec{x})$ est :

$$(\mathsf{aut}_\circ, n_2, h', init(\nu(n_2)))$$

La traduction de cet état en B est $state_aut = n2$ et si $n2$ n'est pas élémentaire, $state_n2 = init_n2$. L'initialisation est appliquée à toutes les variables d'état de $n2$. Puisque les variables d'état de $n1$ si elles existent ne sont pas modifiées quand l'ASTD **a** n'est pas dans l'état $n1$, les variables d'état de $n1$ jouent le rôle de fonction historique, et leurs valeurs après l'exécution de $op(\sigma)$ correspondent à celles de h'.

De ce fait, $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$. ■

Axiomes aut_2, aut_3, aut_4, aut_5

La seule différence avec aut_1 pour chacune de ces règles sont les valeurs de C et T. Pour aut_2 par exemple :

$$C = state_name \in dom(t_name_sigma) \land phi$$

$$T = state_name := t_name_sigma(state_name)$$

$$\| \; state_n2 := t_n2_sigma(out(state_name)) \; \| \; initAll_n2b$$

Le reste de la preuve est similaire à celle de aut_1. On retrouve après l'exécution de l'opération $op(\sigma)$ un état en B correspondant à l'état traduit de l'ASTD après l'exécution de $\sigma(\vec{x})$. Cela est dû au fait que les substitutions B et leurs préconditions correspondent aux règles d'inférence des ASTD.

4.4.2 Pas d'induction

Supposons que nous ayons prouvé que la traduction conserve la sémantique des ASTD pour toute preuve d'exécution de taille n. Soit $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$ de profondeur $n+1$ avec $n > 1$. $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$ est donc composée d'au moins n règles et d'au moins un axiome (un, dans le cas ou l'arbre est droit, plusieurs sinon). Les axiomes correspondent aux feuilles de l'arbre et la racine est donc une règle. Soit R la règle racine, et T le type d'ASTD correspondant. Soient $A_1 \ldots A_m$ les m arbres de preuves de profondeur maximale n obtenus en retirant R de $\Pi^{\mathbf{a}}_{\sigma(\vec{x})}$. Nous devons donc montrer que les prémisses R établissent les conclusions du ou des arbres de preuves $A_1 \ldots A_m$.

La substitution du type d'ASTD T s'exprime en premier dans l'opération puisque le calcul des opérations s'effectue en profondeur d'abord, et que **a** de type T est l'ASTD de plus haut niveau. Sachant que l'exécution de $\sigma(\vec{x})$ dans **a** est possible nous savons donc que le prédicat constituant la précondition de l'opération est évalué à *true*. La substitution doit donc exécuter l'une des branches présentée dans la section 4.3. Deux cas sont possibles.

Soit la substitution qui s'exécute est un **SELECT** et une seule branche est active, ou est une substitution d'un autre type, et dans ce cas la substitution qui s'exécute correspond exactement à la règle R. Si la substitution peut s'exécuter alors la règle peut s'appliquer et réciproquement, si la règle d'inférence peut s'appliquer, alors la substitution peut s'exécu-

ter. On obtient alors par induction que les variables d'état de la machine B après l'exécution de $op(\sigma)$ correspondent à $\eta(\mathbf{a}^{s\prime})$ puisque l'on sait qu'une variable d'état ne peut pas être modifiée par un ASTD de niveau supérieur. De ce fait, nous avons $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$. ■

Soit la substitution qui s'exécute est de type **SELECT** mais plusieurs branches sont actives et dans cas, la preuve d'exécution correspond à l'une des substitutions qui peut s'appliquer. Il n'est cependant pas possible de garantir laquelle de ces substitutions va s'appliquer. Mais celle correspondant à la preuve d'exécution de $\sigma(\vec{x})$ dans $\mathbf{a}$ est l'une de ces substitutions. Cela signifie que plusieurs exécutions sont possibles pour l'ASTD initial et que le choix a été fait d'avoir cette preuve d'exécution pour $\sigma(\vec{x})$. Ainsi si l'on prend l'ensemble des états résultant de ces exécutions possibles de $\sigma(\vec{x})$ et que l'on prend l'ensemble des variables d'état résultant de l'exécution des substitutions non déterministes, nous devons pouvoir associer à chaque état de l'ASTD un ensemble de variables d'état. Si les preuves d'exécution correspondantes sont de taille inférieure à $n+1$, alors la correspondance $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$ est établie par hypothèse d'induction. Sinon, la preuve est de taille supérieure à $n+1$ mais est bornée par la profondeur maximale de l'ASTD que l'on note *max*. Il est donc possible de prouver l'égalité pour $n = max - 1$. On aura alors $\mathscr{M}_a^{s\prime} = \eta(\mathbf{a}^{s\prime})$. ■

4.5 Conclusion

Dans ce chapitre, nous avons présenté un ensemble de règles de traduction d'une spécification ASTD en B. Ces règles génèrent une machine B codant la sémantique de l'ASTD et représentant l'état de l'ASTD à l'aide de variables. À chaque évènement ASTD correspond une unique opération B. Lors de l'exécution de cette opération, les variables B sont modifiées de telle sorte que leur nouvelle valeur soit la représentation de l'état de l'ASTD après l'exécution de l'évènement. La traduction assure que les systèmes de transition de l'ASTD et de la machine B générée sont isomorphes. Comme conséquence, toute propriété satisfaite par le système de transition de la machine B est aussi satisfaite par l'ASTD. Par exemple, les preuves de propriétés temporelles effectuées sur la machine B sont aussi valide pour l'ASTD. Si les attributs de l'ASTD sont codés de manière cohérente en B, alors toute preuve d'invariance en B est également valide pour l'ASTD.

Comme suite de ce travail, on pourrait envisager la traduction d'ASTD dont les automates ne sont pas déterministes, ainsi que les ASTD avec des appels mutuellement récursifs.

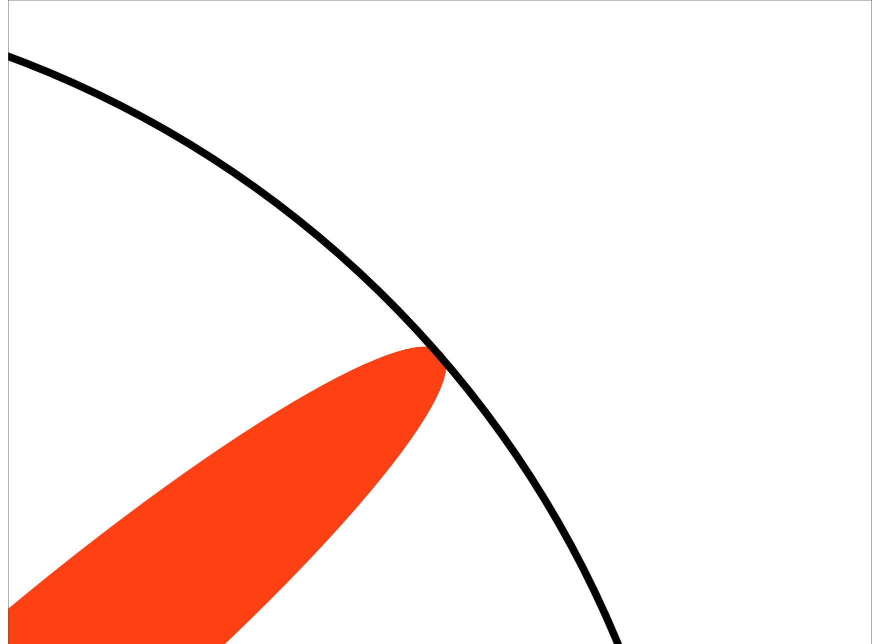
Cependant ces cas ne sont pas bloquants puisque les spécifications ASTD considérées dans le cadre des SI et des règles de contrôle d'accès n'utilisent pas ces fonctionnalités. En effet, il est d'une part toujours possible de déterminiser un automate, et d'autre part, nos algorithmes de traduction EB^3 vers ASTD ne génèrent pas d'ASTD avec des appels mutuellement récursifs. Enfin, il est possible de réduire la taille des substitutions **SELECT** en effectuant les mêmes optimisations que celles faites dans [87]. De plus, il serait possible de couper certains branches des **SELECT** en évaluant certains prédicats qui valent trivialement soit *true false*. Enfin, un outil pourra être développé pour traduire automatiquement une spécification ASTD en machine B.

Deuxième partie

Une modélisation B des politiques de contrôle d'accès

Chapitre 5

Combinaison d'UML, ASTD et B pour la spécification formelle d'un filtre de contrôle d'accès



[65] Ph.D. étape 5 – Et pourtant on veut continuer à apprendre, repousser cette limite qui semble si rigide. Alors on pousse. Parfois en vain, parfois sans savoir dans quelle direction, mais on pousse.

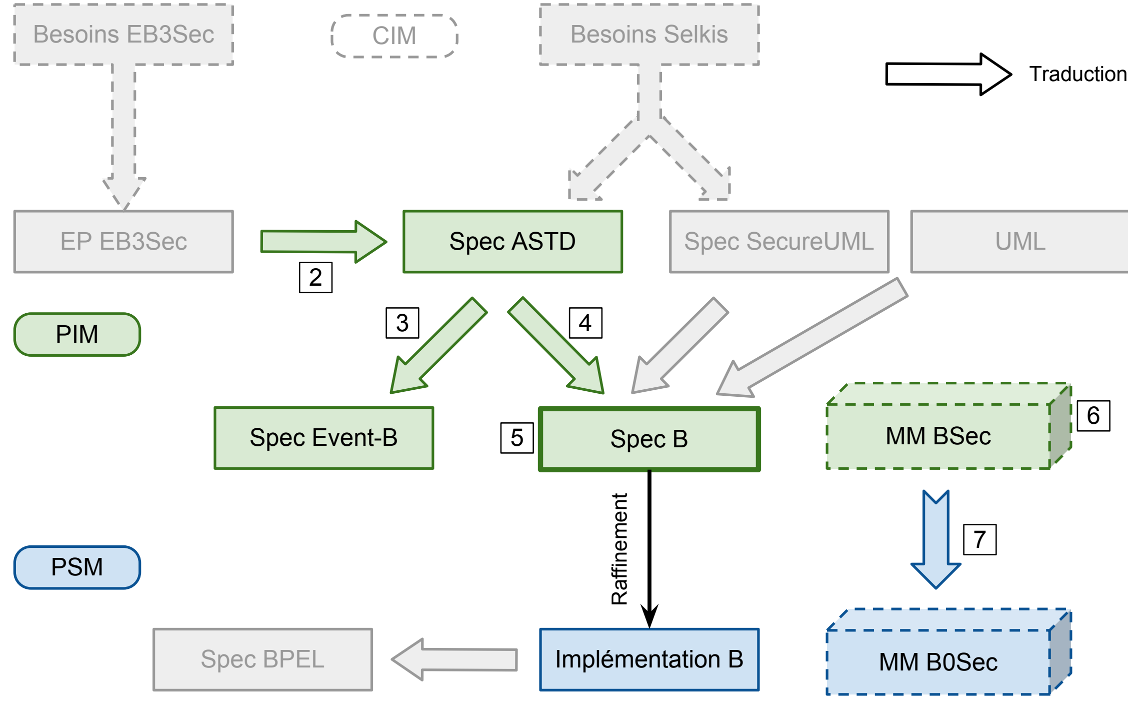


FIGURE 5.1 – Modèle formel B de politiques de contrôle d'accès

Dans ce chapitre, nous nous intéressons à la spécification en B d'un filtre de contrôle d'accès, comme indiqué à l'étape 5 de la Figure 5.1. Ce filtre est composé de plusieurs machines B dont au moins une est issue d'une traduction d'un modèle exprimé avec la notation ASTD. Ainsi ce chapitre combine la machine traduite avec d'autres machines dans le but d'obtenir un modèle formel de filtre de contrôle d'accès qui inclut le modèle fonctionnel du système ainsi que les modèles de la politique de contrôle d'accès. L'objectif est de valider le comportement complet du couple filtre + système.

Résumé

La combinaison de méthodes formelles et semi-formelles est de plus en plus nécessaire pour produire des spécifications qui peuvent à la fois être comprises et donc validées par le concepteur et l'utilisateur mais aussi suffisamment précises pour être vérifiées via des approches formelles. Cela motive notre approche d'utiliser des paradigmes complémentaires afin de spécifier certains aspects de la sécurité des systèmes d'information. Cet article présente une méthodologie pour spécifier des politiques de contrôle d'accès de systèmes d'information à l'aide de notations graphiques : UML pour le modèle fonctionnel, SecureUML pour le modèle statique de contrôle d'accès et ASTD pour le modèle dynamique de contrôle d'accès. Ces diagrammes sont traduits en un ensemble de machines B. Finalement, nous présentons la spécification formelle d'un filtre de contrôle d'accès qui coordonne les différentes règles de contrôle d'accès et la spécification fonctionnelle des opérations. Le but de ces spécifications B est de vérifier rigoureusement la politique de contrôle d'accès d'un système d'information en utilisant les nombreux outils de la méthode B.

Commentaires

Ma contribution au sein de cet article comprend la description des machines B présentées ; l'application de la méthodologie décrite à l'étude de cas ; des tests et de l'animation des machines B ; la rédaction d'environ 60% du texte ; ainsi que la relecture, correction et adaptation de l'article consécutivement aux commentaires des arbitres. Une première version de l'article a été soumise et acceptée à la quatrième édition du workshop international IEEE *UML and Formal Methods* (UML&FM'2011) ayant eu lieu à Limerick en Irlande le 20 juin 2011. Nous présentons ici la version étendue et acceptée au journal *Innovations in Systems and Software Engineering (ISSE).*

Combining UML, ASTD and B for the Formal Specification of an Access Control Filter

Jérémy Milhau [1,2] Akram Idani[3] Régine Laleau[2]
Mohamed-Amine Labiadh[3] Yves Ledru[3] Marc Frappier[1]

1 : GRIL, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke QC, Canada
{Jeremy.Milhau,Marc.Frappier}@usherbrooke.ca

2 : LACL, Université Paris-Est, IUT Sénart Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
{Frederic.Gervais,Laleau}@u-pec.fr

3 : UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS
Laboratoire d'Informatique de Grenoble UMR 5217, 38041 Grenoble, France
{Yves.Ledru,Akram.Idani,Mohamed-Amine.Labiadh}@imag.fr

Abstract Combination of formal and semi-formal methods is more and more required to produce specifications that can be, on the one hand, understood and thus validated by both designers and users and, on the other hand, precise enough to be verified by formal methods. This motivates our aim to use these complementary paradigms in order to deal with security aspects of information systems. This paper presents a methodology to specify access control policies starting with a set of graphical diagrams: UML for the functional model, SecureUML for static access control and ASTD for dynamic access control. These diagrams are then translated into a set of B machines. Finally, we present the formal specification of an access control filter that coordinates the different kinds of access control rules and the specification of functional operations. The goal of such B specifications is to rigorously check the access control policy of an information system taking advantage of tools from the B method.

Keywords Access control policy, B, SecureUML, ASTD

5.1 Introduction

Our aim is the formal specification of information systems (IS) access control policies. Roughly speaking, an IS helps an organization to collect and manipulate all its relevant data. Access control is part of security issues that can grant or deny the execution of actions depending on a policy. An access control policy defines, for an authenticated user, which actions he is allowed or forbidden to execute depending on several criteria such as role, organization, etc. It is the combination of atomic rules. Depending on the kind of rules an access control designer wants to express, several languages and notations can be used. In an access control policy specification, static and dynamic rules may be required in order to express all access control requirements. In our work, we consider static rules independently of the functional behavior of the system. Permissions, prohibitions and static Separation of Duty (SoD) are static constraints. A static SoD constraint means that if a user is assigned to one role, he is prohibited from being a member of a second role [34]. Dynamic constraints require to take into account the history of the system, that is the set of actions already performed on a system, which is represented by the system state and its evolutions. For instance, obligations and dynamic SoD are dynamic constraints. With dynamic SoD, users may be authorized for roles that may conflict, but limitations are imposed, based on the history of actions performed and roles taken by the user.

In our approach, we have chosen to use SecureUML [57] to express static rules and the ASTD notation [32] for dynamic rules. These notations are presented in Section 5.3. These graphical specifications are then translated into B machines [3] using translation rules described in Section 5.4. Similarly, as we need also to specify the functional model of the IS since access control rules can refer to elements of this model, its UML specification is translated into B. The next step consists in specifying the access control filter that coordinates the different kinds of access control rules and the functional model. The B specification of this filter is presented in Section 5.5. We start in the next section by describing the context of the work.

5.2 Context

5.2.1 The Selkis Project

The Selkis project [1] funded by the French national research agency (ANR) aims to define a development strategy for secure healthcare network IS from requirements engineering to implementation. The results of this project can be applied to any type of secure IS, but the medical field was chosen because of the complexity and diversity of security requirements. An approach based on formal methods was chosen in order to build implementations correct by design and to perform proofs and model checking over rules that check properties of the specification.

The approach adopted in the Selkis project advocates a separation between the access control policy and the functional model at the requirements, specification and implementation levels. An implementation of an access control filter is produced. It intercepts actions before they are executed. If the action and other parameters match access control rules, the action can be executed by the IS. In the other case, the execution is denied.

5.2.2 Illustrative Example

In order to illustrate our approach, we consider an example from a medical information system. The structural representation of this example is modeled by the UML class diagram of Fig. 5.2. This diagram manages medical information about patients in a hospital. Class MedicalRecord stores medical information about a given patient, such as his medications, using the attribute *data*. Every patient has at most one medical record which is not depending of hospitals. A doctor can practice in at most one hospital and he can leave and join a hospital using methods *joinHospital* and *leaveHospital*. The information system imposes the following constraint:

C1. A patient cannot leave a hospital if his medical record is not validated,

The access control policy associated to our example includes the following rules:

R1 Doctors can read both public and private attributes of a medical record but they can only modify public attributes (*i.e.* data),

1. http://lacl.fr/Selkis/

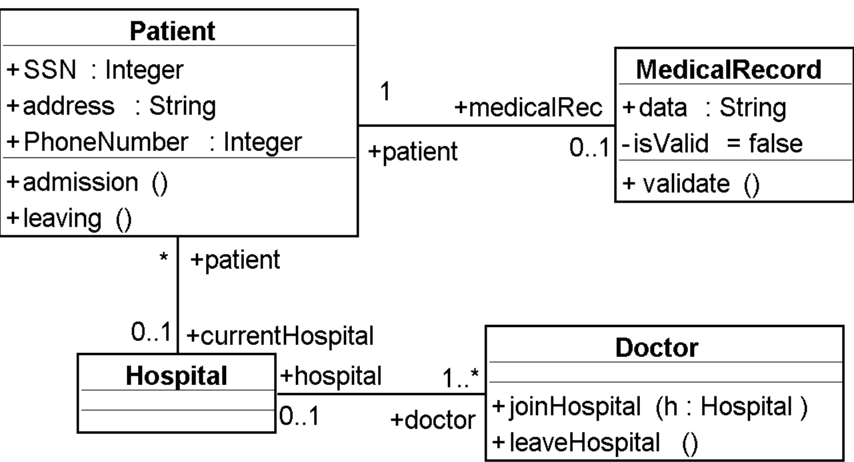


Figure 5.2: Functional model

R2 Doctors can only validate medical records using the *validate* method,

R3 Modification and validation of a medical record of a given patient, can only be done by a doctor who does belong to the same hospital as the patient,

R4 If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read access to his medical record.

R5 Any modification of a patient's medical record must be eventually validated. Several modifications can be validated by a single validation.

Other rules exist in order to define the permissions to execute actions of classes *Patient* and *Doctor* but they are not presented in this paper for the sake of concision.

5.3 Graphical Models for Access Control

In our methodology we propose to use SecureUML [57] to model static access control, and ASTD [32] diagrams for dynamic access control. Static access control is based on RBAC (Role-Based Access Control) [83], which describes authorizations granted to users on resources. Dynamic access control rules can refer to previous states of the IS.

5.3.1 SecureUML

SecureUML [57] is a graphical modeling language designed to integrate information relevant to access control into application models defined with the Unified Modeling Language (UML). It extends a functional UML model using concepts of role-based access control models (RBAC) in order to model roles and their permissions. This is the main reason why we have chosen SecureUML rather than UMLSec [50]. In SecureUML, users are grouped into roles and may play several roles with respect to the secure system. Fig. 5.3 uses concepts of SecureUML to model rules R1 and R2. It shows how medical records are secured when they are accessed by doctors.

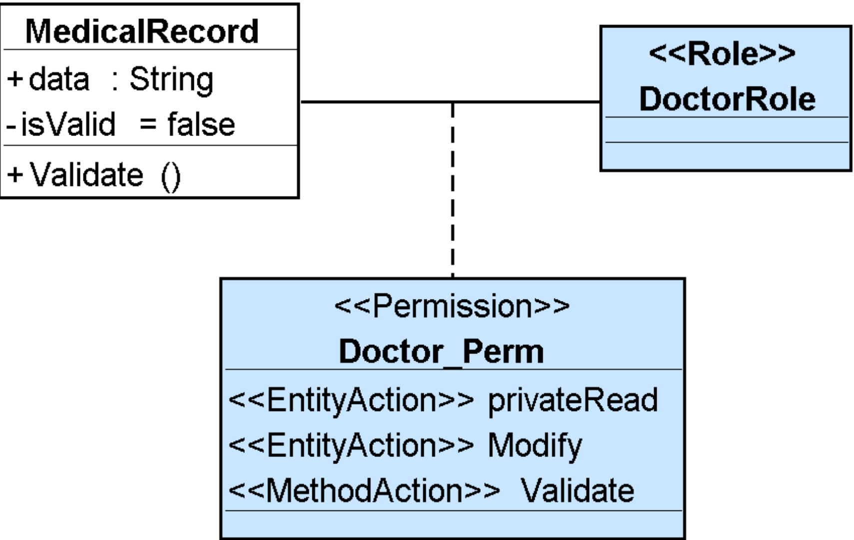


Figure 5.3: Access control rules for medical records

This SecureUML specification indicates that users with role *Doctor* can read private and public attributes (denoted by `≪EntityAction≫ privateRead`), modify public attributes (`≪EntityAction≫ Modify`). This part of the SecureUML specification models rule R1. Rule R2 is modeled by the permission that allows doctors to execute the method of the class *MedicalRecord*: *Validate* (`≪MethodAction≫ Validate`).

Rule R3 refers to an authorization constraint associated to the permission to access a medical record and which links the security and the functional parts of this model. It requires (i) to navigate through the functional model to retrieve the patient associated to the medical record, and his/her current hospital, (ii) to retrieve the doctor corresponding to

the user asking to access the medical record and retrieve his/her associated hospital, (iii) to compare these two hospitals. We can add this constraint by annotating the graphical SecureUML model. However, in order to keep Fig. 5.3 readable, rule R3 is described using a B predicate in Section 5.4.3.

It would be more difficult to express rule R4 using SecureUML, but not impossible. Rule R4 is of dynamic nature because it is based on information about past states of the system. Thus, in order to express it in SecureUML, it would be necessary to add few artificial variables to the functional model to store this kind of information. However, for larger functional models which deal with real information systems it becomes error prone to do so because these variables will be less manageable. This kind of variables is not needed when using the ASTD notation as it offers features streamlining the specification of dynamic aspects. We think that using a graphical notation explicitly modeling states and transitions between states is more intuitive than coding manually state variables that may introduce errors in the specification. This approach of coding states into variables is generally used in SecureUML, since it does not provide operators to specify ordering constraints between actions.

5.3.2 The ASTD Notation

The ASTD notation is a graphical representation having a formal semantics. It was created to specify systems, in particular IS. ASTD was introduced as an extension of Harel's Statecharts [41] and is based on operators from EB^3 [38] (a process algebra dedicated to IS specifications). Readers are invited to consult a formal and mathematical description of the ASTD notation in [32].

In our IS hospital example, access control rule R4 is a dynamic rule since it refers to several actions and defines ordering constraints upon them. Hence this rule is modeled using the ASTD notation as described in Fig. 5.4. The ASTD uses a notation separating actions of the IS and its parameters from security parameters such as the user and his/her role in the IS. It is denoted $< \overrightarrow{s}, a(\overrightarrow{p}) >$ where $\overrightarrow{s}$ is the list of access control parameters (user and role in our example), a is the action the user wants the IS to execute and $\overrightarrow{p}$ are the functional parameters of the action. Since in our example the combination of user and role is checked in the SecureUML model, there is no need to check it again in

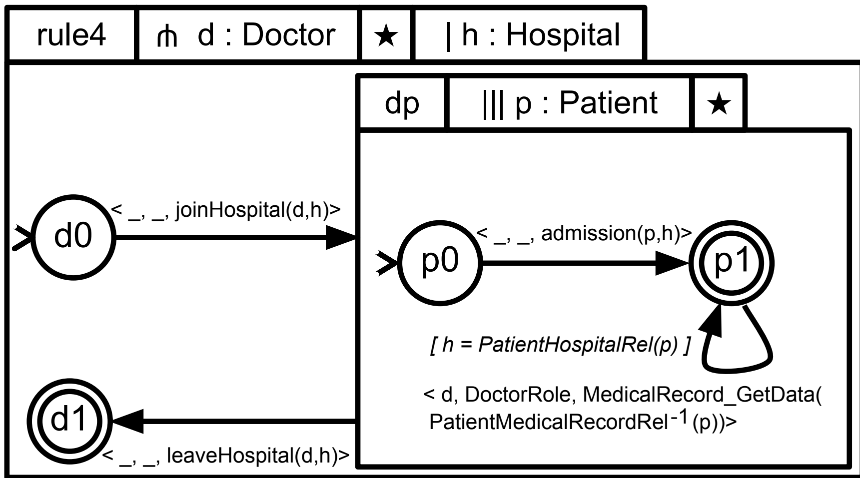


Figure 5.4: ASTD model of rule R4: If a patient has left the hospital, only doctors belonging to the hospital during the patient's stay will keep read access to his medical record

the ASTD model. Some security parameters are wildcards (denoted _), meaning that the specification accepts any values for these parameters. However, if specific constraints upon security parameters are required, they can be specified in the ASTD for instance in the action MedicalRecord_GetData.

The first operator denoted ⫛ d : *Doctor* is a quantified weak synchronization over all doctors of the system. There are as many instances of the quantified ASTD as the number of instances of class *Doctor*. When an action is received, all the instances of the ASTD that can execute it do so at the same time. In other words, if there are instances of the ASTD that can synchronize, they have to do so. The quantified choice operator | h : *Hospital* means that a single instance h of class *Hospital* is associated to the instance d of *Doctor*. This link is created by action JoinHospital(d,h) when a doctor is assigned to a hospital. The link between these instances is removed when the doctor leaves the hospital with the action LeaveHospital(d,h). After leaving a hospital, a doctor can join another one, starting a new link between d and h. This iteration is possible thanks to the Kleene closure operator (denoted by $\star$), meaning that the sub-ASTD can be iterated as many times as needed. Finally,

ASTD named dp describes the ordering constraint of action $\mathsf{admission}(p,h)$ and action :

$$< d, DoctorRole, \mathsf{MedicalRecord_GetData}(i) >$$
$$\text{with } i = PatientMedicalRecordRel^{-1}(p)$$

for all instances of the *Patient* class (due to operator quantified interleave $|||p$: *Patient*). This action must be executed by d, a user who has the role *DoctorRole*. It is guarded by the predicate

$$h = PatientHospitalRel(p)$$

in order to check that the hospital of the patient p and h, the hospital where d works, are the same.

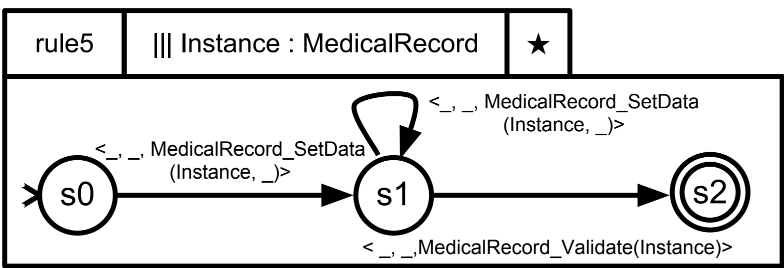


Figure 5.5: ASTD model of rule R5: Any modification of a patient's medical record must be eventually validated. Several modifications can be validated by a single validation

Similarly, we can model rule R5 that depicts an obligation. After one or several modifications of a given medical record, the record must be validated. Rule R5 is modeled in Fig. 5.5 using the ASTD notation. This ASTD describes the process for all medical records, and this process can be repeated. This is modeled using $|||$ *Instance* : *MedicalRecord* and the Kleene closure *. Then we have an automaton describing the ordering of actions *MedicalRecord_SetData* and *MedicalRecord_Validate*. This automaton means that one or several *MedicalRecord_SetData* can be executed and are eventually followed by single *MedicalRecord_Validate*.

In OrBAC [22], the notion of context may be useful for some dynamic constraints. Indeed, contexts allow to refine permissions in order to give rights in specific circumstances (*e.g.* emergency situations). They can govern periods of validity of privileges, or reduce/ex-

tend the access rights inherited from a role. Especially, OrBAC proposes the notion of provisional context [13] which depends on previous actions the user has performed in the system. This assumes that the information system manages a log that stores data about previous activities of users in the system. The OrBAC approach requires then the effective implementation of the system. In our approach, the advantage of ASTD models, compared with OrBAC provisional contexts, is that they address the conceptual phases of a secure IS development process rather than implementation or runtime.

5.4 Translations into B Specifications

The idea of integrating formal and graphical notations (*i.e.* B and UML) has been studied since several years [35], and was commonly motivated by the complementarities of these two types of notation. Indeed, the disadvantages of semi-formal methods can be overcome thanks to contributions of formal methods and vice versa. UML graphical views are synthetic, structural and intuitive. However, their semantics are often described as blurred. Therefore, the construction of systems based on these methods can sometimes lead to ambiguous models. On the other hand, the major strengths of formal methods are precision of the abstract mathematical notations and automatic reasoning.

In our approach, we propose to translate both SecureUML and ASTD specifications into B in order to check the global consistency of access control policies. Moreover we also need the functional model since security rules can require to read functional attributes values. Thus, the translation into B is composed of three steps corresponding to the functional, static access control and dynamic access control models.

5.4.1 The B Method

The B method [3] is a method that supports a large segment of the software development life cycle: specification, refinement and implementation. It ensures, thanks to refinement steps and proofs, that the code matches to the specification. The B method is based on Abstract Machine Notation (AMN) and the use of formally proved refinements. Its mathematical basis is founded on first-order logic, integer arithmetic and set theory. A B specification is structured into machines which contain state variables, invariant prop-

erties expressed on the variables and operations specified in the generalized substitution language, which is a generalization of Dijkstras' guarded command language. The refinement mechanism consists in reformulating, by successive steps, the variables and the operations of an abstract machine, so as to finally lead to a module which will constitute a running program. The intermediate steps of reformulation are called refinements and the last one is the implementation.

5.4.2 A Formal Functional Model

The initial functional model (fig. 5.2) is a UML class diagram showing entities and relationships. In order to formally reason on this model, we propose to translate it into the B notation. Translation from UML diagrams into B specifications was addressed by several research works [60, 58, 79]. In order to take advantage of these complementary works we integrated their translation rules in a unified MDE framework [47]. This allows, on the one hand, to combine and adapt their rules, and on the other hand, to extend them in order to take into account translations of UML extensions (*i.e.* SecureUML). The translation of the functional model is strongly inspired by these approaches. We do not use the UML-B [80] framework of RODIN toolset because our objectives are quite different. On the one hand, the RODIN toolset is dedicated to the Event-B language and on the other hand the UML-B [80] approach gives a UML syntax to B which does not cover UML constructs that we need in our approach such as composition or navigation.

Translation principles. The functional model is translated into a unique B machine containing sets, variables and associations derived from classes, attributes, and class relations. As proposed by the existing approaches, class named *MedicalRecord* of Fig. 5.2 leads to an abstract set *MEDICALRECORD* and a variable *MedicalRecord* representing respectively the set of possible instances and the set of existing instances of class *MedicalRecord*. Fig. 5.6 illustrates basic B structures related to class *MedicalRecord*.

Our MDE platform generates basic operations such as constructors, destructors, getters, setters in order to allow state evolution of the functional formal model. This step takes into account some basic structural invariants related to mandatory and/or unique attributes,

```
MACHINE
    Functional_Model
SETS
    MEDICALRECORD ;
    THEDATA ;
    ...
ABSTRACT_VARIABLES
    MedicalRecord ,
    MedicalRecord__data ,
    MedicalRecord__isValid ,
     ...
INVARIANT
     MedicalRecord ⊆ MEDICALRECORD ∧
     MedicalRecord__data ∈ MedicalRecord ⇸ THEDATA ∧
     MedicalRecord__isValid ∈ MedicalRecord → BOOL ∧
     PatientMedicalRecordRel ∈ MedicalRecord ↣ Patient
     ...
OPERATION
executed ← MedicalRecord__Validate(Instance) ≙
    PRE
        Instance ∈ MedicalRecord∧
        MedicalRecord__isValid ( Instance ) = FALSE
    THEN
        MedicalRecord__isValid ( Instance ) = TRUE ;
        executed := ok
    END;
```

Figure 5.6: Basic B structures related to medical records

inheritance, composition, multiplicities... An example of a basic getter which allows to get medical record data is given in Fig. 5.7.

The operation returns a medical record data and information about the success of its execution. This last point is useful for the access control filter detailed later. From a functional point of view, reading medical records data should always succeed.

B specifications resulting from our translation process can be enriched in order to take into account less obvious functional constraints. Let us consider for example constraint C1 which means that if attribute *isValid* of a medical record is *false* then the patient of this medical record must be linked to some hospital. This can be expressed by the following invariant:

```
result, executed ← MedicalRecord__GetData(Instance) ≙
    PRE
        Instance ∈ MedicalRecord
    THEN
        result := MedicalRecord__data(Instance) ||
        executed := ok
    END;
```

Figure 5.7: Operation *MedicalRecord__GetData*

$$\forall pp \cdot (pp \in Patient \wedge MedicalRecord__isValid(PatientMedicalRecordRel^{-1}(pp)) = \mathbf{FALSE} \Rightarrow PatientHospitalRel[\{pp\}] \neq \varnothing)$$

Contrary to operation *MedicalRecord__GetData*, the success of operation *MedicalRecord__SetData* is constrained by C1. In fact, this basic setter modifies attribute *data* of a MedicalRecord and also sets the attribute *isValid* to *false*. Fig. 5.8 presents this B operation and shows how the previous invariant is respected.

```
executed ← MedicalRecord__SetData(Instance, data) ≙
    PRE
        Instance ∈ MedicalRecord ∧
        data ∈ THEDATA
    THEN
        IF
            PatientMedicalRecordRel(Instance) ∈
                        dom(PatientHospitalRel)
        THEN
            MedicalRecord__data(Instance) := data ||
            MedicalRecord__isValid(Instance) := FALSE ||
            executed := ok
        ELSE
            executed := failure
        END
    END;
```

Figure 5.8: Operation *MedicalRecord__SetData*

Failure of *MedicalRecord__SetData* indicates to the access control filter that someone tried to modify data of a medical record of a patient who is not admitted in any hospital and that this action is forbidden by the functional part of the model.

5.4.3 A Formal Static Access Control Model

To our knowledge there is no attempt to translate secureUML models into B. Nevertheless, in [75] the authors gave an Event-B specification for OrBAC policies which is mainly dedicated to formally prove the process of deploying a security policy. In our work, we mainly deal with the modeling level of a security policy and its impact on the functional model.

In order to translate the security part of our model, we propose a mapping which leads to structures that represent data types. First, we propose a B formalization of a variant of the SecureUML meta-model. Then, the security model elements are directly injected in this B specification. For example, in the following we give some enumerated sets issued from Fig. 5.3.

SETS
  $ENTITIES = \{MedicalRecord \ldots\};$
  $ATTRIBUTES = \{data \ldots\};$
  $ACTIONS = \{\text{MedicalRecord__SetData} \ldots\};$
  $PERMISSIONS = \{Doctor_Perm \ldots\};$
  $ROLES = \{DoctorRole \ldots\};$
  ...

Invariants of the security formal model define various relations between these elements and also structural constraints imposed by the meta-model. For example, permission assignments and role hierarchy are defined by:

$$\text{PermissionAssignement} \in PERMISSIONS \rightarrow (ROLES \times ENTITIES)$$
$$\wedge\ Roles_Hierarchy \in ROLES \leftrightarrow ROLES\ \wedge$$
$$\mathbf{closure1}(Roles_Hierarchy) \cap \mathbf{id}(ROLES) = \varnothing$$

This means that a permission links at the most one pair (role $\mapsto$ entity), and that *Roles_Hierarchy* has no cycle. Invariants also include RBAC constraints such as the definition of SSD (Static Separation of Duty) which forbids a user to take conflicting roles even in different sessions.

The initialisation clause evaluates the various relations according to the SecureUML meta-model instance. This allows to check that the security model respects the structural constraints of the SecureUML meta-model such as the non-circular role hierarchy. A brief overview of the initialisation clause is given below:

INITIALISATION
$AttributeOf := \{(data \mapsto MedicalRecord), \ldots\}$
$AttributeKind := \{(data \mapsto public), \ldots\}$
$OperationOf := \{(MedicalRecord__SetData \mapsto MedicalRecord), \ldots\}$
$setterOf := \{(MedicalRecord__SetData \mapsto data), \ldots\}$
$PermissionAssignement :=$
$\quad \{(Doctor_Perm \mapsto (DoctorRole \mapsto MedicalRecord)), \ldots\}$
$\text{EntityActions} := \{(Doctor_Perm \mapsto \{privateRead, modify\}), \ldots\}$
$\ldots$

Operations of the B specification derived from the security model are dedicated to control access to the operational part of the functional formal model. We associate to each functional operation a secured operation in the security model which verifies, based on the initial state, that a user has permission to call the functional operation. For example, operation *secure_MedicalRecord__SetData* presented in Fig. 5.9 is intended to verify accesses to the functional operation *MedicalRecord__SetData* of Fig. 5.8. Secured operations add parameters *user* and *role* corresponding respectively to the user who is trying to invoke the operation and one of his roles ($role \in roleOf(user)$). Predicate "$MedicalRecord__SetData \in isPermitted[\{role\}]$" verifies whether operation *MedicalRecord__setData* is allowed to the connected user using a particular role. Indeed, set *isPermitted* computes, from the initial state, the set of authorized functional operations for each role. For instance, it contains the couple ($DoctorRole \mapsto MedicalRecord__SetData$).

As mentioned in Section 5.3.1, rule R3 refers to a constraint which is added to the SecureUML model using an annotation linked to permission *Doctor_Perm*. It is expressed in the B language as a precondition of modification actions (≪`EntityAction`≫ `Modify`).

```
answer ← secure_MedicalRecord__SetData(Instance, user, role) ≙
  PRE
    Instance ∈ MedicalRecord ∧
    user ∈ USERS ∧ role ∈ ROLES ∧
  THEN
    IF
      role ∈ roleOf(user) ∧
      MedicalRecord__SetData ∈ isPermitted[{role}] ∧
      (user ∈ Doctor ⇒ HospitalDoctorRel(user) =
      PatientHospitalRel(PatientMedicalRecordRel(Instance)))
    THEN
      answer := granted
    ELSE
      answer := denied
    END
  END
```

Figure 5.9: Operation *secure_MedicalRecord__SetData*

This annotation is taken into account in the **IF** statement. Then rule R3 is a predicate which conditions the granting of the execution of action *MedicalRecord__SetData* (a setter of the *data* attribute) and which means that the doctor must be employed by the current hospital of the patient:

```
P(user,instance) ≙
  (user ∈ Doctor ⇒ HospitalDoctorRel(user) =
  PatientHospitalRel(PatientMedicalRecordRel(instance)))
```

This constraint shows the impact of the functional model on the access control model because the hospital in which the patient is admitted and the hospital of the connected doctor originate from the functional model. In the tool translating SecureUML models into B machines, this kind of annotation is taken into account and inserted with no modification in the B operation. This requires that the designer expresses the constraint as B statements. In SecureUML these constraints can be modeled using OCL, however there is no tool that can validate both static and functional models with such constraints [59].

Fig. 5.10 details the operation that check if the action *GetData* on a medical record is granted: *secure_MedicalRecord__GetData*. This operation is another example of the translation of the SecureUML model into B.

$answer \leftarrow$ **secure_MedicalRecord__GetData**($Instance$, $user$, $role$) $\widehat{=}$
PRE
$Instance \in MedicalRecord \wedge$
$user \in USERS \wedge role \in ROLES$
THEN
IF
$role \in roleOf(user)$
THEN
$answer := granted$
ELSE
$answer := denied$
END
END

Figure 5.10: Operation *secure_MedicalRecord__GetData*

5.4.4 Dynamic Access Control Model Translation

In [64] we have specified translation rules from ASTD to Event-B. However, one goal of the Selkis project is to implement an access control filter for information systems. Thus, we need the refinement process of the B method that leads to a proved implementation. In order to do so, we have adapted translation rules of [64] for B as described in [63]. This translation is made in three steps. The first step starts from the root ASTD and goes down to the leaves: a variable is created for each ASTD in order to encode its state. The second step creates a B operation for each transition label. In this step, a set of constant B functions is also created to encode the transitions of all the automata. Each transition label of each automaton has its own B transition function. Finally, the third step starts from the leaves ASTD and goes up to the root ASTD. It modifies the B operations according to the semantics of the type of each nested ASTD.

Fig. 5.11 presents the B operation for the action *Dynamic_MedicalRecord__GetData*. This action is affected by rule R4 described by the ASTD presented in Fig. 5.4. The B translation of the ASTD generates several conditions that refer to the current state of the IS. The first two conditions $user \in Doctor$ and $role = DoctorRole$ ensure that the user willing to read the medical record is currently a doctor and is connected with the role $DoctorRole$. The third condition is the translation of the guard on the action of the ASTD as presented in Fig. 5.4 ; it ensures that $\mathsf{joinHospital}(d,h)$ was executed, hence the value of the hospital

h for the doctor d has been recorded by the system thanks to the *StateQchoice* function associated with the quantified choice operator of ASTD. The next condition

$$StateAutomaton_DoctorHospital(user) = dp$$

ensures that the doctor did not leave the hospital by checking that the ASTD is still in the state *dp*, *i.e.* before the execution of leaveHospital(d,h). Finally, the last condition

$$StateAutomaton_dp(user, Instance) = p1$$

ensures that the patient was indeed admitted in the hospital after the doctor has joined the hospital. $StateAutomaton_dp$ and $StateAutomaton_DoctorHospital$ are partial functions used to model respectively the state of the inner automaton and the state of the upper automaton, in which dp is a place. Complete description of these functions is provided in [63].

```
answer ← Dynamic_MedicalRecord__GetData(Instance,user,role ) ≙
  PRE
    Instance ∈ MedicalRecord ∧
    user ∈ USERS ∧
    role ∈ ROLES
  THEN
    LET pp BE pp = PatientMedicalRecordRel (Instance) IN
     IF
       user ∈ Doctor ∧
       role = DoctorRole ∧
       StateQchoice(user) = PatientHospitalRel(pp) ∧
       StateAutomaton_DoctorHospital(user) = dp ∧
       StateAutomaton_dp(user,pp) = p1
     THEN
       StateAutomaton_dp(user,pp) := p1 ||
       answer := granted
     ELSE
       answer := denied
     END
    END
  END
```

Figure 5.11: Operation *Dynamic_MedicalRecord__GetData*

5.5 Specification of the Access Control Filter

Once the different kinds of access control rules have been specified, it is necessary to define how they are combined and how the final decision of permitting the execution of an action by a specific user is calculated. This is the role of the access control filter. We use the B refinement process to specify it. First an abstract B model is built and its refinement allows the decision algorithm to be described.

5.5.1 Abstract Access Control Filter Specification

The abstract filter B machine is built by creating one B operation for each action to secure in the IS plus one for the rollback of the system. The rollback operation (presented in Fig. 5.12) is needed in the case where access control grants the execution of one action that cannot be executed by the functional part of the system. Hence, dynamic access control filter that depends on the state of the IS must be restored to a state where the action has not been executed. The other operations describe inputs and outputs for the filter's operations. An example of these operations is presented in Fig. 5.12 with the filtered version of the B operation *Filter_MedicalRecord__GetData*. At this level of specification, we only describe the goal of the operation *i.e.* granting or denying the execution of the action. Substitution *CHOICE* is a non-deterministic choice between three substitutions. The algorithm that calculates which answer will be returned is detailed in the next step, the refined filter specification.

5.5.2 Refinement of the Access Control Filter

The abstract access control filter is refined into a more concrete filter that includes static, dynamic and functional models. The inclusion of these machines permits the execution of operations from them. The refinement of these operations introduces calls to static and dynamic access control operations ; if the policy grants the execution, the functional operation is executed. Fig. 5.13 presents the overall view of the approach, with the graphical specifications translated into B machines combined in an access control filter. The top layer presents the different models expressed using graphical notations that are then translated into B machines. The second part of Fig. 5.13 introduces the abstract access control policy specification composed by all the translated B machines.

```
Rollback()≙
  PRE
    rollbacklock = locked
  THEN
    rollbacklock := unlocked
  END

answer, executed, data
 ← Filter_MedicalRecord__GetData(Instance, user, role) ≙
  PRE
    Instance ∈ MedicalRecord ∧
    user ∈ USERS ∧
    role ∈ ROLES ∧
    rollbacklock = unlocked
  THEN
    CHOICE
      answer := granted || executed := ok
      || data :∈ THEDATA || rollbacklock := unlocked
       OR
      answer := granted || executed := notExecuted
      || data :∈ THEDATA || rollbacklock := locked
       OR
      answer := denied || executed :∈ {ok, notExecuted}
      || data :∈ THEDATA
      || rollbacklock :∈ {locked, unlocked}
    END
  END
```

Figure 5.12: Abstract operations *Rollback* and *Filter_MedicalRecord__GetData*

The access control policy designer has to specify the algorithm that computes the answer of his/her policy. We call this feature the decision algorithm. An acceptable decision algorithm is to put static and dynamic filters in conjunction. This means that both static and dynamic access control policies must grant the execution in order for the filter to grant the execution. In our example, we have chosen this solution, but any other combination can be specified. We could imagine another way to combine policies for our example: in the case of emergency, the dynamic access control policy could be replaced by a new one, more permissive, in order to reduce constraints and improve efficiency of medical staff. Fig. 5.14

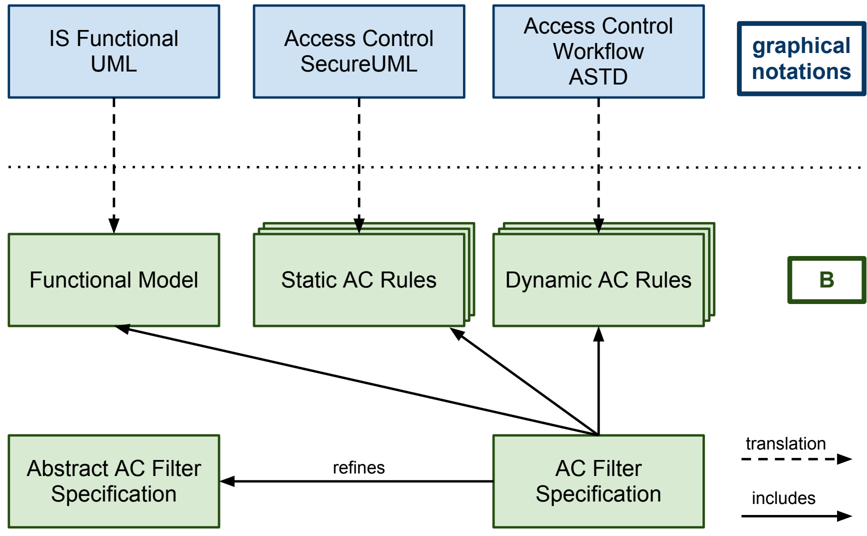


Figure 5.13: Overall view of the access control filter

is the refinement of operation *Filter_MedicalRecord__GetData* introduced in Fig. 5.12 and describes the combining algorithm for static and dynamic policies.

Rollback is an important part of our filter. Contrary to the static access control specification that does not evolve when executed, the dynamic access control specification is based on a state that must be consistent with the state of the IS. In the case where both static and dynamic policies grant the execution and that the execution fails at the functional level, we have to restore the state of the dynamic policy to its previous state. We do not detail the refinement of the rollback operation, but it consists in calling a B operation of the dynamic access control B specification in order to restore it to its previous state. However this requires that the previous state was saved into variables before. In order to do so, we have defined an operation called *saveDynamicState* in the dynamic access control that will backup state variables before any call to operations.

Such an access control filter can be implemented using the BPEL language [6] and can be included in a service oriented architecture (SOA) environment [19]. When IS are implemented using Web services, like in SOA, security features are often implemented in a Policy Enforcement Manager (PEM). A PEM is based on two main parts: the Policy Decision Point (PDP) and the Policy Enforcement Point (PEP). The PDP takes the decision to grant or deny the execution based on several informations such as the policy and combining

algorithms. The PEP is the link between the PDP and the functional IS. Our filter perfectly corresponds to a PEM since it provides exactly the functionalities of both a PEP and a PDP.

5.5.3 Verification and Validation Purpose

The various graphical diagrams are dedicated to a better understanding of several aspects of an information system and their corresponding formal models support a rigorous reasoning about these aspects. For our case study, we proved the machines and the refinement using Atelier-B prover. Validation can be performed by animating the model. This consists of playing several scenarios with the ProB [55] tool:

Normal scenario

- A secretary S of a hospital H creates a patient P, admits him in the hospital and associates to him an empty medical record;
- A doctor D joins the hospital H;
- The doctor D modifies attribute data of the medical record of P and validates it;
- The patient P leaves the hospital H.

Attack scenarios

- A secretary trying to validate a medical record;
- A doctor trying to modify a medical record of a patient who belongs to another hospital;
- A patient who leaves a hospital with a medical record which is not validated.

It is also possible to verify properties against the whole system, *i.e.* the combination of the filter, the access control machines and the functional machine. We can verify that temporal properties hold using model checkers such as ProB [55] or using the proof-based verification approach developed by Mammar et al. in [61] and Frappier et al. in [25]. For instance, we could verify that rule R5 is correctly enforced by checking that the CTL property

$$\mathbf{AG}\ (\neg\ MedicalRecord_isValid(i) \Rightarrow$$

$$\mathbf{AGEF}\ pre(MedicalRecord_Validate(i)))$$

holds against the specification of the filter. Indeed, the predicate

$$\neg\ MedicalRecord_isValid(i)$$

means that *i*, an instance of the *MedicalRecord* class, has been modified but not yet validated. Informally, this property means that if an instance of the *MedicalRecord* class has been modified but not yet validated (predicate $\neg\ MedicalRecord_isValid(i)$) it is always possible to validate it. In other words, since we want to ensure that after one or more modifications, the medical record will be validated, then we want to check that on all execution paths we can always find a path that eventually leads to a state where the precondition of the B operation $MedicalRecord_Validate(i)$ holds.

5.6 Conclusion

In our work, we apply a combination of formal and graphical techniques in order to propose a technique covering all aspects of access control policies in the context of information systems. Our methodology starts by various kinds of graphical models and produces a complete formal B specification. We use UML class diagrams to model structural functional models, SecureUML to express static access control rules and ASTD to represent history-based rules. In order to remedy to the lack of tools for verifying or analyzing (formally) these diagrams we translate them into B. Our work is then intended to formalize access control policies in order to reason on the derived formal specifications using associated tools: AtelierB prover and ProB model checker and animator [55, 56]. Another contribution of our approach is that it becomes possible to design information systems as a whole using graphical views for its functional and access control aspects, and then generate a complete formal specification for the whole system.

Works on OrBAC [22] propose procedures to analyze security policies, however they do not take into account functional models. Links between access control rules and the functional specification cannot be formally checked. This is done rather in the implementation or deployment steps. In [57], authors have conducted an in-depth analysis of the literature on research works that combine graphical and formal methods for designing IS, including both functional and access control purposes. To our knowledge, the closest work

to ours is presented in [88], where functional and security models are merged into a single UML model which is translated into Alloy. However, the access control rules described in [88] are mainly of static nature. Moreover Alloy proposes verification techniques based on model-checking whereas B also provides a theorem prover.

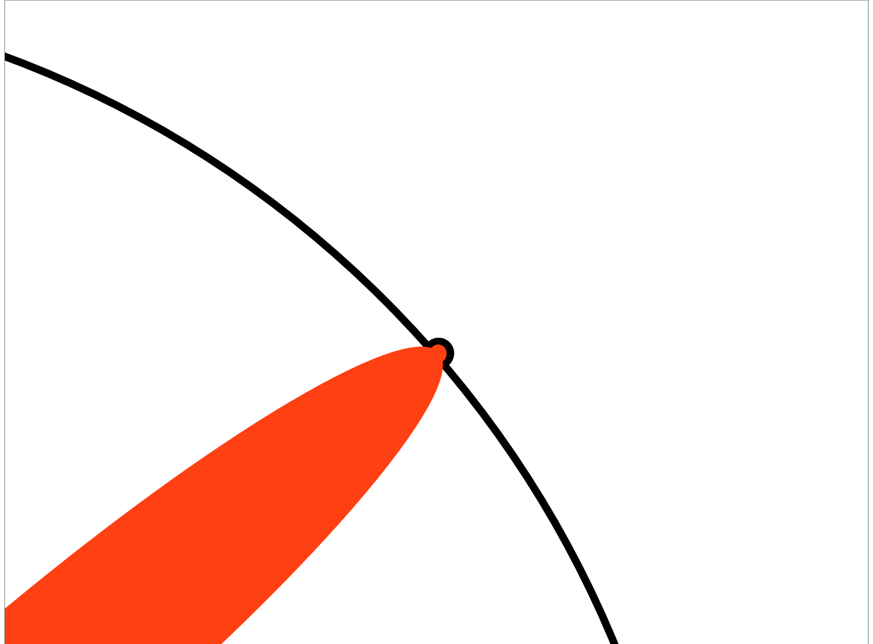
We are currently working on the tool implementing the translation of ASTD into B. Further work includes the evaluation of our approach on case studies of the Selkis Project.

```
answer, executed, data
  ← Filter_MedicalRecord__GetData(Instance, user, role) ≙      PRE
    Instance ∈ MedicalRecord ∧
    user ∈ USERS ∧
    role ∈ ROLES ∧
    rollbacklock = unlocked
  THEN
    VAR static, dynamic, functional, return IN
      static ←
       secure_MedicalRecord__GetData(Instance,user,role) ;
      IF static = granted
      THEN
        dynamic ←
          dynamic_MedicalRecord__GetData(Instance,user,role) ;
        IF dynamic = denied
        THEN
          answer := denied ; executed := notExecuted ;
          data :∈ THEDATA
        ELSE
          functional, return
            ← MedicalRecord__GetData(Instance) ;
          IF functional = ok
          THEN
            answer := granted ; executed := ok ; data := return
          ELSE
            rollbacklock := locked ;
            answer := granted ; executed := notExecuted ;
            data :∈ THEDATA
      ELSE
        answer := denied ; executed := notExecuted ;
        data :∈ THEDATA
    END
  END
```

Figure 5.14: Refinement of *Filter_MedicalRecord__GetData*

Chapitre 6

Un méta-modèle B pour l'expression de politiques de contrôle d'accès



[65] Ph.D. étape 6 – Et soudain, après des mois d'efforts, la frontière est repoussée. Un Ph.D. c'est cette boursouflure sur le bord des connaissances humaines.

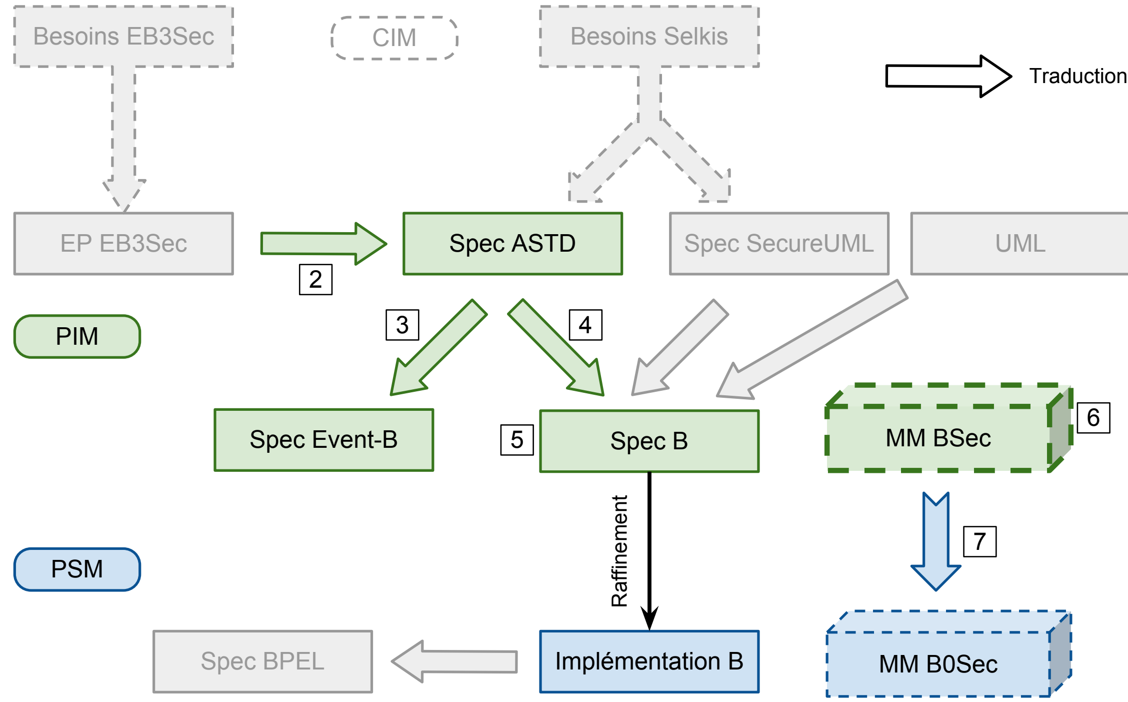


FIGURE 6.1 – Méta-modèle B_{sec}

Dans ce chapitre, nous nous intéressons à la définition d'un méta-modèle étendant le méta-modèle B permettant de définir un filtre de contrôle d'accès, comme indiqué à l'étape 6 de la Figure 6.1. Cet article doit être considéré comme venant en complément de celui présenté dans le chapitre 5. Il décrit un méta-modèle permettant de définir les concepts importants d'une politique de contrôle d'accès en B en prévision du raffinement qui conduira vers l'implémentation.

Résumé

La vérification et la validation d'une politique de contrôle d'accès pour un système d'information est une tâche difficile mais nécessaire. Afin de tirer avantage des caractéristiques formelles et des outils de la méthode B, nous introduisons dans cet article un méta-modèle de modélisation B pour les politiques de contrôle d'accès. Ce méta-modèle est une base nécessaire pour le développement d'un prototype formel d'un filtre de contrôle d'accès combiné au système d'information, afin de vérifier et valider une politique avant son implémentation.

Commentaires

Ma contribution au sein de cet article comprend le développement du méta-modèle combinant SecureUML et ASTD pour l'expression de politiques de contrôle d'accès ; ainsi que de la rédaction majoritaire du texte. Une première version de cet article a été soumise et acceptée à la quatrième édition du workshop international *Foundations & Practice of Security* (FPS 2011) ayant eu lieu à Paris en France du 12 au 13 mai 2011. Nous présentons ici une version étendue qui fait l'objet d'une publication dans la série *Lecture Notes of Computer Sciences* de Springer dans le volume 6888.

A Metamodel of the B Modeling of Access-Control Policies

Jérémy Milhau [1,2], Marc Frappier [1], Régine Laleau [2]

1 : GRIL, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke QC, Canada
{Jeremy.Milhau,Marc.Frappier}@usherbrooke.ca

2 : LACL, Université Paris-Est, IUT Sénart Fontainebleau, Département Informatique
Route Forestière Hurtault, 77300 Fontainebleau, France
{Laleau}@u-pec.fr

Abstract Verification and validation of access-control policies for information systems is a difficult yet necessary task. In order to take advantage of the formal properties and tools of the B method, we introduce in this paper a metamodel of the B modeling of access control policies. This metamodel leads to the development of a formal prototype of an access control filter combined to the system. It allows verification and validation of policies before implementation.

Keywords Metamodel, access control, IS, formal method, MDA, B [1]

6.1 Introduction

In the context of information systems (IS), we advocate the use of formal methods in order to prevent unexpected behavior and produce a software correct by design. Following this approach, we have proposed a way to specify the functional part of an IS and to systematically implement it using the APIS platform [31]. We now focus on the specification of access-control (AC) policies, *i.e.* the expression of rules describing if an action can be executed in the system given a context. We think that separating AC policies from the functional core of a system fosters maintainability and reduces the complexity of future modifications.

1. This research is funded by ANR (France) as part of the SELKIS project (ANR-08-SEGI-018) and by NSERC (Canada).

However, expressing AC rules is not a trivial task. Several languages and notations have been proposed, each one with its strengths and weaknesses. Our approach encompasses two steps to specify AC rules. The first one consists in expressing AC rules with UML-like graphical notations, thus they can be understood and validated by stakeholders. Then these graphical notations are translated into formal notations in order to be verified and animated [68].

6.1.1 Combining Static and Dynamic rules

AC rules can be characterized by several criteria. For instance, they can be either static or dynamic. According to Neumann and Strembeck in [71], dynamic rules are defined as rules that can only be evaluated at execution time according to the context of execution. Static rules are defined as invariants on the state of a system. For example a static rule would grant the execution of action a by user u at any time, whereas a dynamic rule would grant the execution of action a by user u only if action b was not executed by u before. In order to be evaluated, such dynamic rules require an history of previously executed actions, which can be represented in either the IS state or the policy enforcement manager state. They define allowed or forbidden workflows of actions in a system. We have chosen two notations: Secure-UML and ASTD in order to model AC policies [68].

6.1.2 A Formal Notation for AC rules

The B notation, developed by Abrial [3], is an abstract machine notation based on set theory and first order logic. It involves a development approach based on refinements of models called *machines*. The initial machine is the most abstract model of the system. Each subsequent refinement step introduces more concrete elements into the model until a final refinement produces an *implementation*, *i.e.* a machine that can be implemented using a programing language such as C. Each machine is composed of static and dynamic parts. The static part refers to *constants* and *invariants*, *i.e.* properties of the machine. The dynamic part refers to *variables* and *operations* that modify variables of the machine.

We would like to take advantage of verification and validation tools associated with the B method such as ProB [55], an animator and model checker for B specifications. For that reason, we have chosen to use B to express AC policies.

In order to express dynamic rules in a platform-independent model (PIM), we used the ASTD notation [32], an extension of Harel's Statecharts using process algebra operators. This notation offers an automata-like representation of workflows which can be combined using sequence, guard, choice, Kleene closure, interleave, synchronization and quantifications over choice and interleave. It also provides a graphical representation of dynamic rules. For static rules, we use a SecureUML-based notation [57]. We then translate both ASTD specifications [64] and SecureUML diagrams into B machines (named *Dynamic_Filter* and *Static_Filter* respectively in the following sections). The final step of our work is to build an AC filter implemented using the BPEL notation [23]. We want to use the refinement approach of B to obtain the implemented filter. In order to do so, we have to identify the concepts required in the implementation and to extend the B metamodel with these new *types* for AC modeling. We then provide refinement rules between the new types and their implementation.

6.2 A Proposal for AC Modeling using B

Our proposed metamodel is an extension of the B metamodel [46] with AC aspects. It describes an AC engine that is able to grant or deny the execution of an action of the IS according to an AC policy. If certain conditions are met then the action is executed. If one of the conditions does not hold, then it denies the execution. Conditions, *i.e.* AC rules, can refer to security parameters (such as user, role, organization, ...) or functional (business) parameters.

Our goal is the refinement to several platform-specific models (PSMs). In order to do so, we have to specify concepts such as rules or policies that are not explicitly specified in the original B metamodel. In our PIM metamodel, we specialize classes from the B Metamodel into new ones that correspond to concepts used for defining AC policies. This helps the refinement process by linking a concept of the abstract level to a concept of a more concrete level. The metamodel also combines the static and dynamic parts of the specification of an AC policy. Since the metamodel is composed of many classes and is quite large, we will describe it here in a fragmented way.

6.2.1 Combinaison of B Machines

Fig. 6.2 presents the part of the metamodel that describes machines and operations used to model a complete AC engine. We will detail in the following sections this part of the metamodel from top to bottom. The first line is composed of B machines as denoted by the generalization link between classes. The AC engine is composed of at least four machines. Each one plays a role in the process of granting or denying the execution. The four kinds of machines composing an engine are the following:

1. The functional machine is the core of the IS. It describes how the system evolves over time and how attributes and classes of the system are modified by the execution of an action. The functional model of the system can also introduce ordering constraints on the actions to be executed.
2. Several *static filters* can be included in the engine. Each one of them describes a static policy, *i.e.* a set of static AC rules. Such rules are expressed in an RBAC-like (Role based access-control) notation. For example, there may be one static filter for authorization and one for prohibition.
3. Several *dynamic filters* can be included in the engine. Each one of them describes a dynamic policy, *i.e.* a set of dynamic AC rules that can grant or deny an execution according to the state of the IS, or the state of the dynamic policy itself. All dynamic filters may not be needed at all time. One of the dynamic filters can be activated, for example, in emergency situations only.
4. The AC filter is the main part (controller) of the engine. It is in charge of asking each machine if the action can be executed according to several parameters. It includes all the other machines so it can call their operations. Then it combines the answers it received in order to decide if the action is granted or not. According to the context, it can decide to ignore the decisions made by the static filters and the dynamic filters, for instance in the case of emergencies in an hospital.

Our metamodel also takes into account operations of these B machines. To illustrate them, we use a case study of an hospital. We consider an action of the functional part of the system, called Admission(*Patient*), that denotes the admission of a patient in one of the units of the hospital. Such an action is generally performed by a user of the IS who is a doctor.

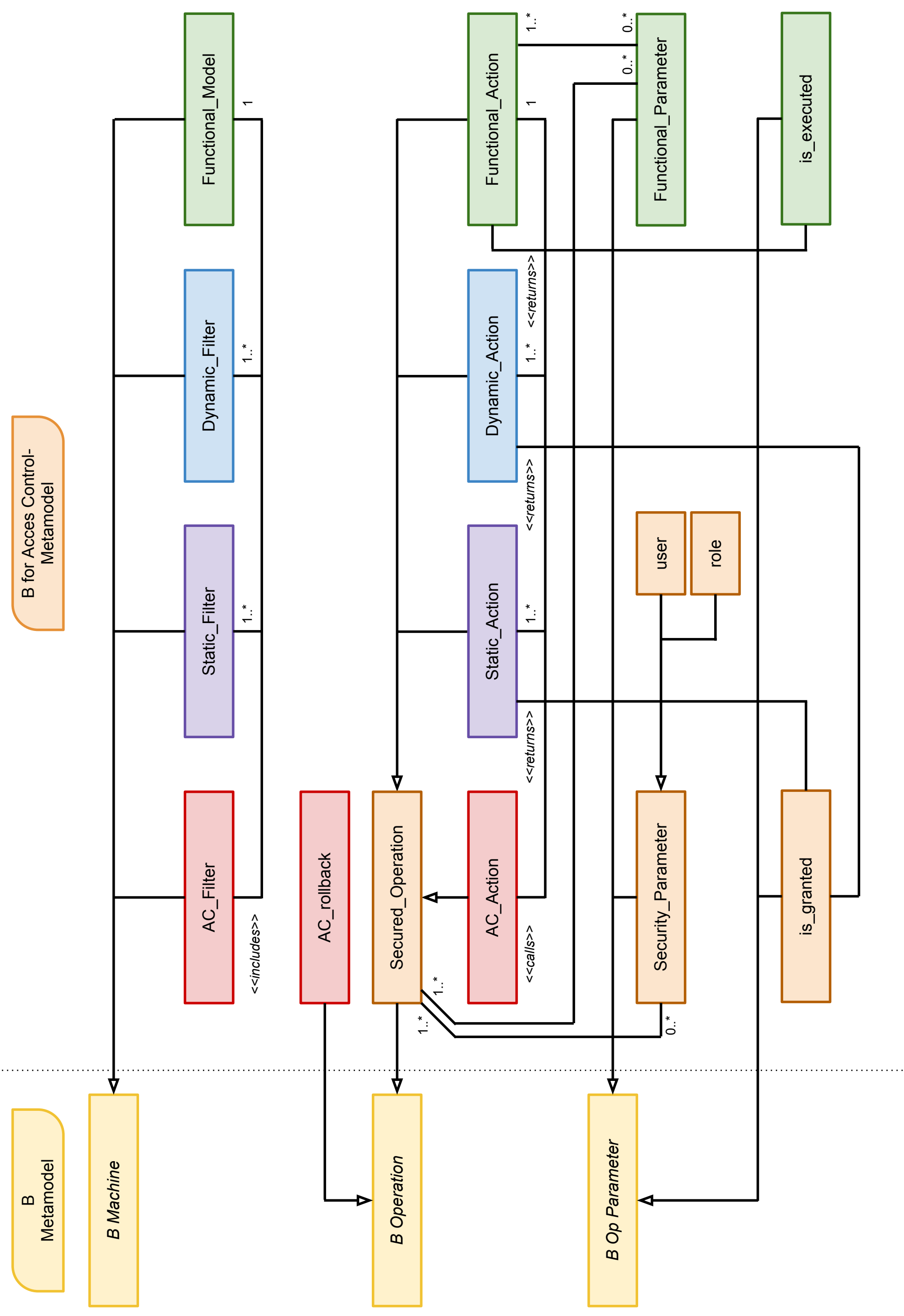


Figure 6.2: Main part of the metamodel for AC modeling in B

1. The functional operation Admission(*Patient*) is in the functional machine. It is an example of a *Functional_Action*. It specifies all the modifications that the execution of the action performs on all the entities of the IS and their attributes. It can also describe conditions in order for the action to be executed, for example the patient must have paid his medical bills before being admitted.
2. The static operation Static_Admission(*Patient*, *User*) is in the static machine. It is an example of a *Static_Action*. It will return *granted* or *denied* according to the static policy. Note that in our example a security parameter *User* was added, compared to Admission(*Patient*). This addition can be used to check that the user is a doctor and not a nurse for instance.
3. The dynamic operation Dynamic_Admission(*Patient*, *User*) is in the dynamic machine. It is an example of a *Dynamic_Action*. It will return *granted* or *denied* according to the dynamic policy. The security parameter *User* can be useful if we want to check that the user of the IS (that should be a doctor, according to the static policy) is a doctor in the same hospital as the patient.
4. Finally, the operation AC_Admission(*Patient*, *User*) is in the AC filter machine. It is an example of an *AC_Action*. It calls all static and dynamic operations described above and computes whether or not to execute the action Admission(*Patient*) of the functional machine. In our example, the algorithm can take into account emergency situations and the filter will bypass the static policy if, for instance, an emergency is declared.

We also need another operation called rollback( ) that is part of the AC filter machine. This operation may be required in specific cases. For instance, if an action was granted to be executed by all static and dynamic machines but the execution fails in the functional machine, the rollback operation must be called. Indeed, the dynamic policy state is modified when the answer *granted* is given. In the event of the failure of the execution in the functional machine, the previous state of the dynamic policy must be restored. This is the role of operation rollback.

Operation parameters are either security parameters or functional parameters. Security parameters refers to user, their role and any other non-functional related information, whereas functional parameters are parameters of the actions of the IS. All operations from

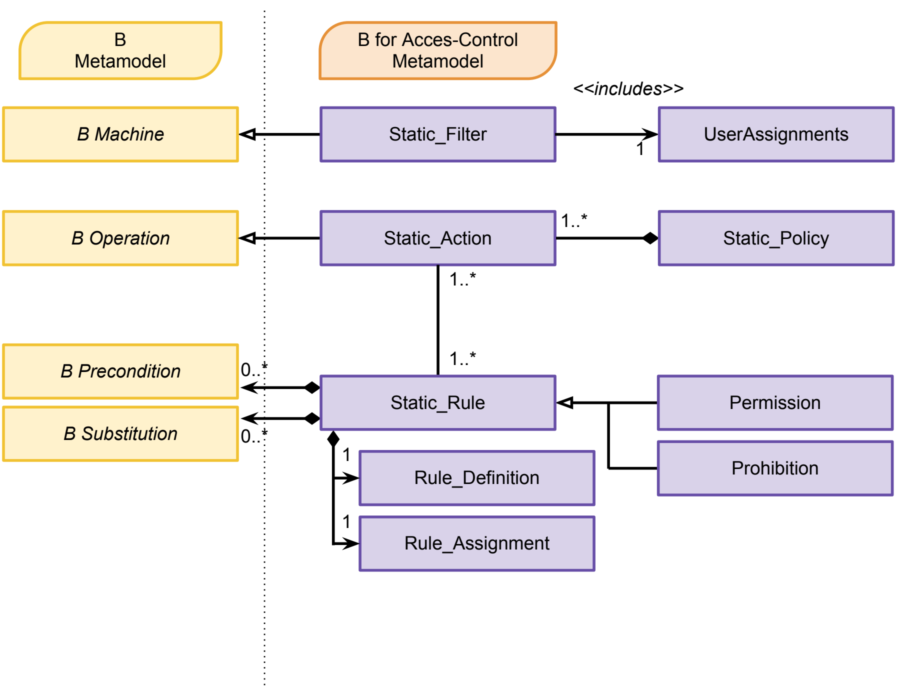


Figure 6.3: Details of the static machine of the metamodel for AC modeling in B

static and dynamic machines return *granted* or *denied* as represented by the *is_granted* class and functional operations return *success* or *failure*.

6.2.2 The Static Filter

The static filter machine defines a static AC policy that will be enforced on the system. In our example, we have chosen an RBAC-like policy. Each rule of such policy can be expressed as a permission or a prohibition. A rule is a combination of instances of role and action classes. For instance, in an hospital, we can give the permission to any user connected as a doctor to perform action Admission(*Patient*).

In Fig. 6.3 the class *Rule_Definition* defines either a permission to perform an action or a prohibition. Such a definition is then linked to a role by the class *Rule_Assignment*. The combination of both is called *Static_Rule* and can be used in one or more B operations (*Static_Action*). The set of all *Static_Action* define the *Static_Policy* of the *Static_Filter*

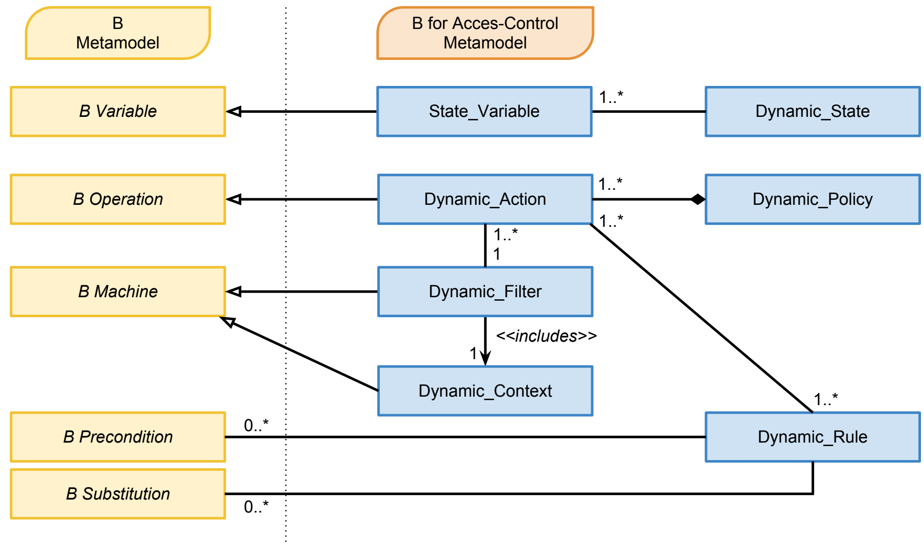


Figure 6.4: Details of the dynamic machine of the metamodel for AC modeling in B

machine. The *Static_Filter* includes the machine *UserAssignments* that defines a role hierarchy and the the roles that a user can play.

6.2.3 The Dynamic Filter

The dynamic filter machine defines a dynamic AC policy that will be enforced on the system. Such policies can be defined using the ASTD notation. When describing an ASTD, there are several parts to take into account. An ASTD is composed of a topology and a state. The topology refers to the structure of the ASTD. It is unaffected by the execution of actions. On the contrary, the state of the ASTD evolves each time an action is executed.

In order to model a workflow using the B formalism, we must encode the topology and the state. We do so in our metamodel as depicted by Fig. 6.4. The state of the workflow (*Dynamic_State*) is stored into several variables called *State_Variable*. The *Dynamic_Policy* is composed of several *Dynamic_Actions* encoding the topology of the workflow into *Dynamic_Rules*. Each rule is composed of B preconditions that can test the value of several *State_Variables* and B substitutions that will modify them, hence the evolving *Dynamic_State*.

6.3 Conclusion

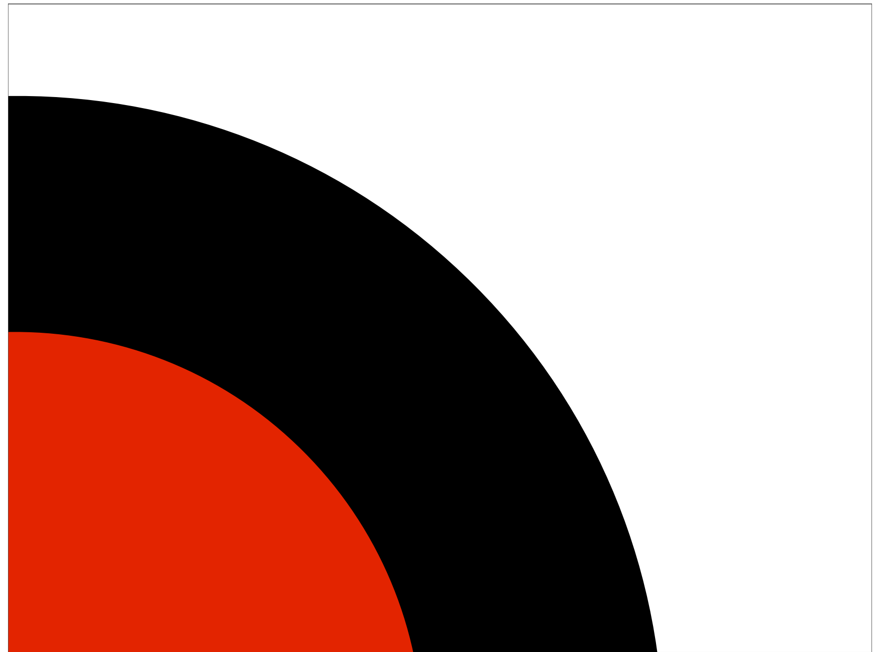
We have defined a metamodel allowing to use the B notation to model access-control policies to be enforced upon an information system. Such B model can be used in order to validate the policy and perform verification and proof of properties, improving the trustworthiness and security of such information systems. In order to validate our approach we have implemented our metamodel using a medical IS case study [68].

The ORKA project [14] proposed a framework to compare the features of AC methodologies. Each method is based on at least one notation that is very good at expressing one type of rule (*i.e.* static rules for *RBAC, dynamic rules for ASTD, etc.). But expressing other types of rules may be difficult (as for static rules with ASTD) or requires the coding of a context introducing state variables. Our approach solves this problem by providing adapted languages for each rules. The combining of all parts of the policy into one single formal model also helps to validate the entire specification. The ORKA project evaluates methods according to criteria such as formalism of the notation, availability of verification and validation tools, availability of a graphical notation for the policy and the possibility to express constraints. Our approach was developed in order to have all these features available.

We are currently working on the next step, *i.e.* translating a model using the refinement mechanisms of B. This will ensure that a correct implementation of AC policies is provided and improve reliability on such a critical part of the system. We are also working on tools for the translation steps of our approach.

Chapitre 7

Une stratégie de raffinement pour l'implémentation de politiques de contrôle d'accès



[65] Ph.D. étape 7 – Pour nous, ce Ph.D. c’est tout une affaire ! Le monde semble complètement différent.

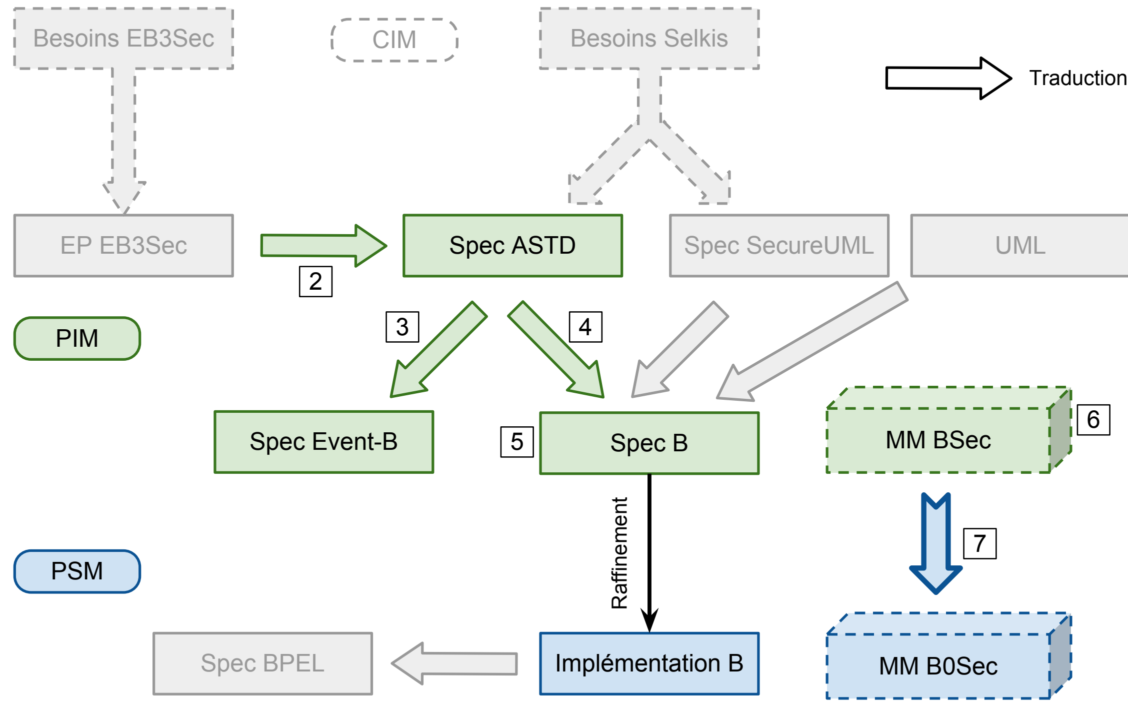


FIGURE 7.1 – Raffinement guidé vers l'implémentation

Dans le chapitre 5 nous avons introduit un modèle formel de filtre de contrôle d'accès que nous avons ensuite détaillé dans le chapitre 6 en définissant un méta-modèle MMB_{sec}, extension du méta-modèle B pour identifier les concepts B correspondant à un filtre de contrôle d'accès. Dans ce chapitre, nous nous intéressons à la définition d'une stratégie de raffinement B permettant d'obtenir l'implémentation B d'un filtre de contrôle d'accès. Cette stratégie de raffinement s'intègre dans le plan de la thèse via l'étape 7 de la Figure 7.1.

7.1 Introduction

Dans le cadre du projet Selkis, plusieurs implémentations d'un filtre de contrôle d'accès sur des architectures différentes ont été développées. Afin de préciser les concepts nécessaires à la définition d'une implémentation d'un filtre de contrôle d'accès, un méta-modèle appelé MM_{imp} a été défini [20]. Il décrit l'architecture d'un filtre de contrôle d'accès dans le cadre d'une architecture orientée services (SOA). Ce méta-modèle a été développé par Michel Embe Jiague *et al.* en collaboration avec les partenaires industriels du projet Selkis. L'objectif de ce chapitre est de définir une stratégie de raffinement B du filtre de contrôle d'accès afin d'obtenir une implémentation qui raffine la spécification abstraite de la politique de contrôle d'accès et qui respecte MM_{imp}. Cette implémentation B peut ensuite être traduite dans une implémentation du filtre de sécurité respectant l'architecture désirée. Nous avons choisi de traduire cette implémentation en BPEL pour pouvoir la comparer à celle de Embe Jiague [19] qui est également réalisée en utilisant BPEL dans une architecture SOA. Cette dernière implémentation est obtenue par la traduction directe de la spécification ASTD en BPEL, sans passer par B.

Dans la section 7.2, nous étudions le méta-modèle d'implémentation MM_{imp} proposé par Embe Jiague *et al.* et ses liens avec le méta-modèle MMB_{sec} décrit dans le chapitre précédent. Nous définissons une stratégie de raffinement qui permet de se conformer à ce méta-modèle dans la section 7.3. Enfin, dans la section 7.4, nous proposons les grandes lignes d'une traduction de l'implémentation B en BPEL.

Dans la suite de ce chapitre, nous nous intéressons à l'implémentation B de la machine *AC_filter* ainsi que l'implémentation des autres machines qu'elle inclut, telles que définies dans le chapitre 5 à la Figure 5.13. Ces implémentations B s'obtiennent par raffinement des machines, en levant le caractère non déterministe des substitutions utilisées ainsi qu'en procédant à un raffinement des structures de données. Les machines à raffiner sont :

1. Le filtre de contrôle d'accès : la machine *AC filter Specification* dans la Figure 5.13, *AC_filter* dans MMB_{sec}
2. La machine décrivant les aspects fonctionnels : *Functional_Model* dans la Figure 5.13 et dans MMB_{sec}
3. La machine décrivant la politique statique, issue de la traduction de la spécification SecureUML : *Static AC Rules* dans la Figure 5.13, *Static_filter* dans MMB_{sec}

4. La machine décrivant la politique dynamique, issue de la traduction de la spécification ASTD : *Dynamic AC Rules* dans la Figure 5.13, *Dynamic_filter* dans MMB_{sec}

Pour l'implémentation de la machine décrivant les aspects fonctionnels, nous pouvons utiliser le processus couvert par les travaux de Mammar *et al.* [69]. Les auteurs proposent la génération de transactions SQL et le code Java associé pour les spécifications B des applications bases de données. Le raffinement et l'implémentation de la machine décrivant la politique statique n'entrent pas dans le cadre de cette thèse. Il est cependant possible d'utiliser, en les adaptant, les travaux présentés dans [10] où l'auteur détaille l'implémentation B de politiques de contrôle d'accès de type RBAC et définit des obligations de preuves permettant de garantir que l'implémentation respecte bien la spécification de la politique de contrôle d'accès. Dans la suite de cette section, nous nous intéressons donc à l'implémentation du filtre de contrôle d'accès et à l'implémentation de la machine B décrivant le comportement dynamique de la politique issue de la traduction des ASTD en B.

7.2 MM_{imp} : Un méta-modèle d'implémentation de politiques de contrôle d'accès

Dans [20], un méta-modèle MM_{imp} pour un PEM (*Policy Enforcement Manager*) gérant le contrôle d'accès dans une architecture SOA a été présenté. Ce méta-modèle définit les composants requis pour le PEM. Les auteurs donnent également les diagrammes de séquence décrivant le comportement du PEM. La Figure 7.2 décrit l'ordonnancement des actions sur chaque élément du filtre de contrôle d'accès avant l'appel du service, *i.e.* l'aspect fonctionnel du système. Dans ce diagramme de séquence, plusieurs acteurs sont mentionnés. Un utilisateur authentifié (*:AutenticatedUser*) envoie une requête notée $E(bp)$ comprenant des paramètres métiers, notés bp. Un intercepteur (*:Interceptor*) récupère cette requête avant sa transmission au service concerné, la complète en ajoutant les paramètres de sécurité (acp) et envoie une nouvelle requête ($E(acp,bp)$) au PEP (*Policy Enforcement Point* noté *:PEP* dans le diagramme). Le PEP effectue alors une requete d'autorisation notée $AR(E,acp,bp)$ pour l'action E et ses paramètres bp ainsi que les paramètres de sécurité acp auprès du PDP principal (*:rootPDP*). Une fois la réponse du PDP connue par le PEP,

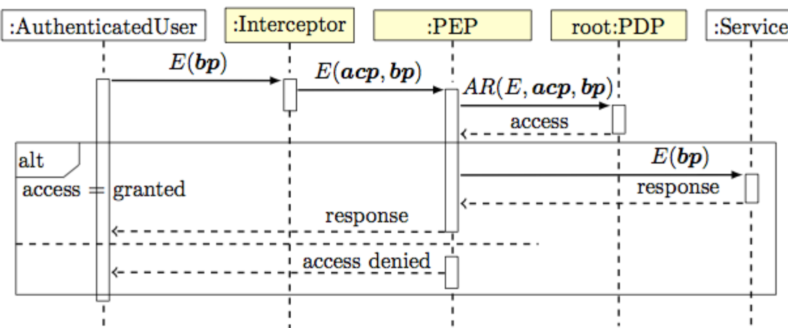


FIGURE 7.2 – Diagramme de séquence : exécution de l'action $E(bp)$ par l'utilisateur

il transmet la requête initiale de l'utilisateur au service si le PDP l'autorise. Il indique à l'utilisateur que sa requête est refusée si le PDP n'a pas autorisé l'exécution.

Dans le cadre des machines B modélisant un filtre de contrôle d'accès, l'*Interceptor*, le *PEP* et *root :PDP* correspondent à la machine B *AC_filter*. En effet, cette machine reçoit la requête de l'utilisateur $E(bp)$, lui ajoute les paramètres de sécurité et va consulter le filtre qui retourne alors la réponse de la politique. Si l'accès est autorisé, la machine transmet la requête $E(bp)$ de l'utilisateur à la machine fonctionnelle (*Service* dans le diagramme de séquence) qui lui retourne sa réponse. Si l'accès est refusé, l'utilisateur est notifié.

Le diagramme de séquence présenté dans la Figure 7.3 détaille le processus de décision conduit par le PDP. Lorsque le PDP principal reçoit une requête d'autorisation $AR(E, acp, bp)$, il la transmet à un des PDP esclaves (*slave :PDP*). Ces PDP sont facultatifs, ils peuvent être utilisés lorsque la politique peut changer en fonction du contexte. Les PDP esclaves sont utiles par exemple quand dans un hôpital en situation d'urgence les contraintes de la politique de contrôle d'accès sont temporairement levées. Un PDP esclave permissif est alors mis en place, et se substitue au PDP esclave habituel. Cet exemple n'est évidement pas restrictif, et nous pouvons imaginer d'autres utilisations pour ces multiples PDP esclaves. Le PDP esclave qui a reçu la requête d'autorisation consulte alors l'ensemble des filtres. Leurs réponses sont combinées et envoyées au moteur de décision (*:DecisionEngine*) qui répond à la requête d'autorisation positivement si la politique autorise l'exécution de $E(bp)$ dans le contexte de sécurité acp.

Dans le cadre des machines B modélisant un filtre de contrôle d'accès, il n'y a qu'un *slave :PDP* chargé de consulter les différents filtres (les machines *Dynamic_filter* et *Sta-*

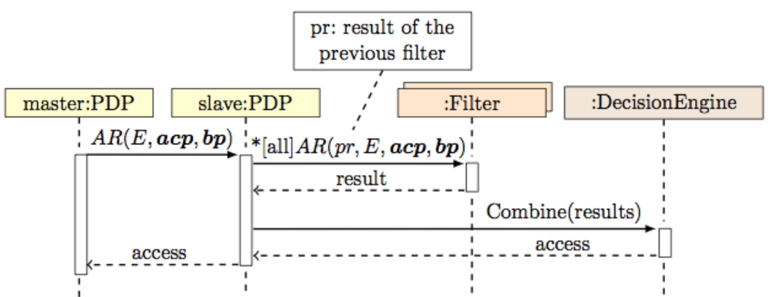


FIGURE 7.3 – Diagramme de séquence : prise de décision en consultant les filtres

tic_filter) et combiner leur réponse avant de déclarer si la politique autorise ou non l'exécution. Ce rôle est confié à la machine B *AC_filter*. Cette machine est également le moteur de décision puisque dans chacune des opérations B, une substitution **IF** décrit l'algorithme de calcul de la décision de la politique de contrôle d'accès pour une des actions du système en fonction des valeurs de retour des opérations des machines *Dynamic_filter* et *Static_filter*.

La Figure 7.4 fait le lien entre le diagramme de classes décrivant le filtre dynamique du méta-modèle MM_{imp} et les classes issues du méta-modèle MMB_{sec} présenté à la Figure 6.4. Les pointillés rapprochent les classes exprimant des concepts similaires. Ainsi, le filtre du méta-modèle MM_{imp} (noté *filter* dans le diagramme *dynAC*) correspond à notre machine *Dynamic_Filter*. La politique dynamique (*Policy*) correspond à l'ensemble des opérations B (*Dynamic_Action*) de la machine *Dynamic_Filter*. Enfin, une règle (*Rule*), correspond à une substitution B (*Dynamic_Rule*) d'une opération B (*Dynamic_Action*).

La Figure 7.5 résume les liens entre les classes *AC_Filter* et *Dynamic_Filter* du méta-modèle MMB_{sec} présenté à la Figure 6.2 et les acteurs des diagrammes de séquence de Embe Jiague *et al.*. Un rapprochement entre les concepts décrits est modélisé par l'utilisation de pointillés. Ainsi, la machine B *AC_Filter* du MMB_{sec} correspond à la fois au *MasterPDP*, au *SlavePDP* ainsi qu'au *DecisionEngine* du diagramme de séquence. Le filtre dynamique correspond à un filtre dans le diagramme de séquence.

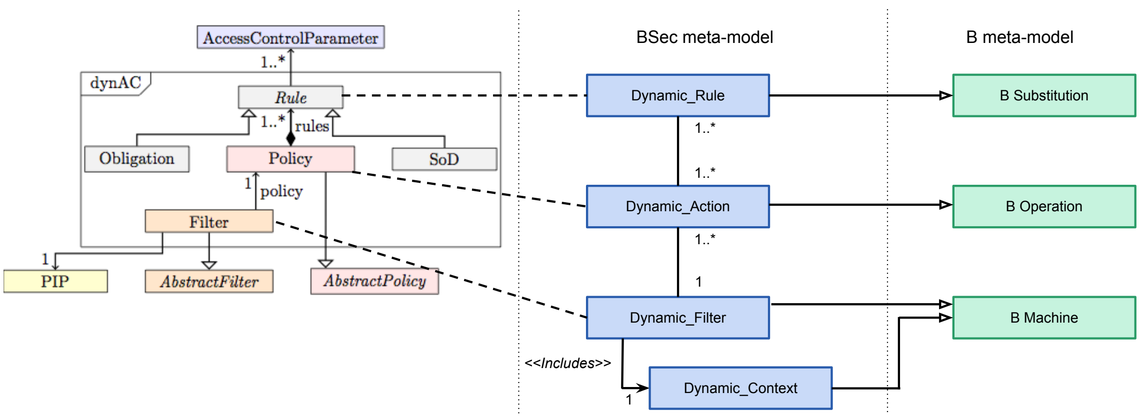


FIGURE 7.4 – Rapprochement du méta-modèle MM_{imp} et du méta-modèle MMB_{sec} : partie dynamique

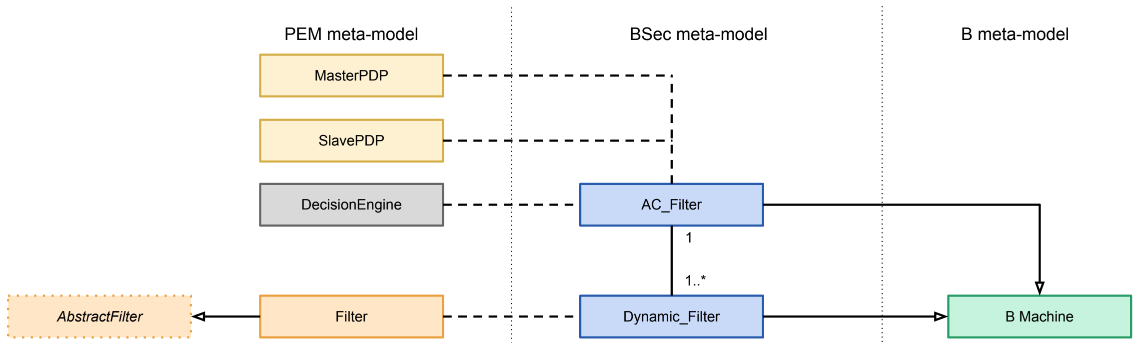


FIGURE 7.5 – Rapprochement des acteurs du diagramme de séquence de Embe Jiague *et al.* et du méta-modèle MMB_{sec}

7.3 Raffinement du filtre de contrôle d'accès

Les contraintes sur les structures de données et sur les substitutions dans une implémentation B sont définies. Certaines substitutions sont interdites dans les implémentations comme par exemple les substitutions **PRE** ou **SELECT**. Concernant les structures de données typant les constantes, ensembles et variables, celles-ci doivent être implémentables, c'est-à-dire pouvoir être traduites en structures de données utilisables dans un langage de programmation impératif tel que C. Ainsi les ensembles abstraits doivent être représentés par de nouveaux ensembles comme par exemple des ensembles d'entiers. Les variables représentant un sous-ensemble d'un ensemble B sont modélisées par un tableau indicé par des entiers et contenant un booléen. Par exemple, soit E un ensemble abstrait contenant six éléments implémenté par l'ensemble $0..5$, et soit v une variable abstraite typée comme étant un sous-ensemble de E. L'implémentation de v est une variable de type tableau, de taille 6, et contenant $true$ dans les cases correspondant aux éléments abstraits de E qui sont élément de v. D'une manière similaire, seules les fonctions totales B dont l'ensemble de départ est implémentable peuvent être représentées par des tableaux. Il faut donc compléter les fonctions abstraites utilisées afin de les rendre implémentables.

7.3.1 Implémentation du filtre de contrôle d'accès : *AC_Filter*

La machine B modélisant un filtre de contrôle d'accès définie dans le chapitre 5 utilise des substitutions déterministes à une exception près. Dans le cadre d'un refus de l'exécution par la politique de contrôle d'accès, une valeur quelconque est retournée comme résultat de l'action *MedicalRecord__GetData*. Cette substitution est notée $data :\in THEDATA$ dans la Figure 5.14. Les autres substitutions utilisées à savoir **IF**, **VAR**, séquencement (**;**) et appels d'opérations conviennent pour une implémentation telle que décrite dans [3]. Lors du raffinement, il est alors nécessaire d'attribuer une valeur par défaut à la variable de retour $data$ qui est choisie de manière déterministe. Par exemple, si l'ensemble $THEDATA$ est constitué de chaînes de caractères de taille bornée, la chaîne vide pourrait convenir comme valeur de retour en cas de refus de l'exécution de l'action *MedicalRecord__GetData* par le filtre de contrôle d'accès.

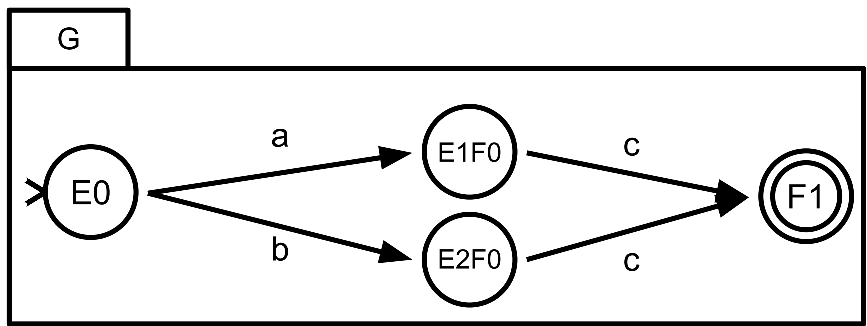


FIGURE 7.6 – Exemple d'ASTD de type automate

7.3.2 Implémentation du filtre dynamique : *Dynamic_Filter*

Nous allons maintenant étudier le raffinement des structures de données et des opérations générées par la traduction en B des ASTD et présentées dans le chapitre 4.

Les ensembles de places Les ensembles de places sont codés en utilisant les entiers naturels pour représenter chaque place. Ainsi l'ensemble $NAME$ codant l'ensemble des noms des ASTD est représenté par l'ensemble $1..card(NAME)$. Chaque élément de $NAME$ devient donc un entier naturel de $1..card(NAME)$. L'ensemble $AutState_a \subset NAME$ codant l'ensemble des noms des places de l'ASTD **a** est représenté par une fonction totale de $1..card(NAME)$ dans les booléens. Chaque image d'un entier correspondra au booléen qui indique si l'élément appartient à $AutState_a$.

Si nous prenons comme exemple l'ASTD de la Figure 7.6, l'ensemble $NAME$ de la traduction en B de cet ASTD est $NAME = \{E0, E1F0, E2F0, F1\}$. Dans l'implémentation, cet ensemble devient l'ensemble 1..4 et chaque entier est associé à un état abstrait par exemple $E0 \mapsto 1,\ E1F0 \mapsto 2,\ E2F0 \mapsto 3,\ F1 \mapsto 4$. Enfin, dans notre cas $AutState_G = NAME$. Ainsi, l'implémentation de $AutState_G$ est une fonction totale de 1..4 dans l'ensemble des booléens telle que chaque entier est associé à $true$.

Les ensembles spécifiques aux types ASTD Certains ASTD ont besoin d'ensembles spécifiques qui servent à coder l'état de chaque type d'ASTD. Ils ont été définis dans le chapitre 4 à la section 4.3. Nous détaillons leur implémentation dans la Table 7.1. Ce tableau présente pour chaque type d'ASTD l'ensemble B utilisé pour la modélisation de son état.

Type d'ASTD	Ensemble B			
Sequence	$SEQUENCE$	$fst \mapsto 1$	$snd \mapsto 2$	
Choice	$CHOICE$	$leftS \mapsto 1$	$rightS \mapsto 2$	$none \mapsto 0$
Closure	$KLEENE$	$started \mapsto true$	$notstarted \mapsto false$	
QChoice	$QCHOICE$	$chosen \mapsto true$	$notchosen \mapsto false$	
Guard	$GUARD$	$executed \mapsto true$	$notexecuted \mapsto false$	
Appel d'ASTD	$CALL$	$called \mapsto true$	$notcalled \mapsto false$	

TABLE 7.1 – Implémentation des ensembles spécifiques aux types d'ASTD

Puis la représentation de chaque élément de l'ensemble abstrait est donnée. Par exemple, l'ensemble $SEQUENCE$ utilisé pour coder l'état d'un ASTD de type séquence est représenté en utilisant l'ensemble $1..2$, 1 correspond à l'implémentation de l'état fst et 2 celle de snd.

Les ensembles de quantification Concernant les ensembles de quantification, chaque élément de l'ensemble est associé à un entier, par exemple l'ensemble de quantification T est représenté par $1..card(T)$.

Les fonctions de transition Les fonctions de transition sont modélisées par des fonctions totales retournant un booléen. Les entrées de la fonction sont l'état d'origine et l'état de destination. Le booléen indique si la transition est définie ou non. Si nous appliquons l'implémentation sur l'exemple présenté en Figure 7.6, voici la traduction B des fonctions de transitions de l'automate : $t_G_a = \{E0 \mapsto E1F0\}$, $t_G_b = \{E0 \mapsto E2F0\}$, $t_G_c = \{E1F0 \mapsto F1,\ E2F0 \mapsto F1\}$. L'implémentation de ces fonctions de transition est la représentation sous forme de tableau de fonctions totales suivantes, compte tenu de l'implémentation des places des automates définie précédemment :

$$t_G_a_imp \quad = \quad \lambda\, x,y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{1 \mapsto 2\}))$$

Cette fonction retourne $true$ lorsque le couple $(1 \mapsto 2)$ est passé en argument, $false$ sinon.

$$\begin{aligned} t_G_b_imp &= \lambda\, x,y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{1 \mapsto 3\})) \\ t_G_c_imp &= \lambda\, x,y\,.\,(x \in 1..4,\ y \in 1..4 \mid bool(x \mapsto y \in \{2 \mapsto 4,\ 3 \mapsto 4\})) \end{aligned}$$

Opérations

Pour chaque type de substitution utilisé dans la traduction, nous présentons leur raffinement. Ce raffinement n'est présenté que quand la substitution n'est pas une substitution d'implémentation. Les substitutions de type **IF** et les affectations (notées :=) sont utilisables directement dans une implémentation.

PRE C THEN S Une substitution précondition n'est pas autorisée dans l'implémentation. Lors du raffinement des machines B, les préconditions doivent être levées.

S1 || S2 Dans la traduction des ASTD en B, les ensembles de variables utilisées par $S1$ et $S2$ sont disjoints. Donc dans notre cas, une substitution simultanée est implémentée en substitution de type séquence. Deux raffinements sont possibles, selon l'ordre que l'on veut appliquer. Le premier opérande en premier ou en second. Nous choisissons d'appliquer la première substitution en premier. Ainsi $S1 \parallel S2$ devient $S1\ ;\ S2$.

SELECT Une substitution sélection s'écrit :

SELECT $C1$ **THEN** $S1$
WHEN $C2$ **THEN** $S2$
...
WHEN Cn **THEN** Sn **END**

Elle est raffinée en substitution de type **CASE** si l'ensemble des $C1 \ldots Cn$ sont disjoints. Dans le cas contraire, deux conditions au moins peuvent être vraies simultanément. À l'implémentation, il faut donc choisir d'appliquer une des substitutions correspondant à une des conditions qui sont vraies. Nous proposons implémenter une substitution **SELECT** en enchaînant des substitutions **IF THEN ELSIF**. Ainsi **SELECT C1 THEN S1 WHEN C2**

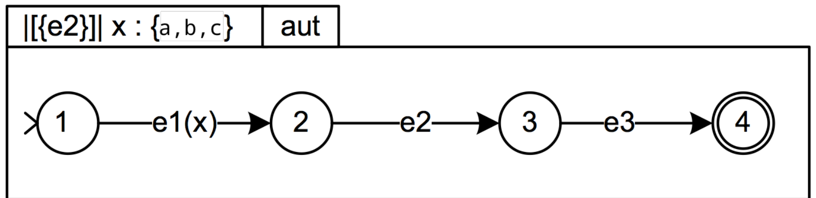


FIGURE 7.7 – Un ASTD de type synchronisation quantifiée

THEN S2 END devient **IF C1 THEN S1 ELSIF C2 THEN S2 END**. Si plusieurs conditions du **SELECT** sont vraies simultanément, alors seule la substitution correspondant à la première condition vérifiée sera exécutée. Le choix de la substitution à exécuter peut également être laissé au concepteur.

ANY x WHERE P IN S Nous savons que la substitution **ANY** n'est utilisée que dans la traduction des synchronisations quantifiées. Dans notre cas, P est un prédicat de typage de x (de la forme $x \in T_synch$) en conjonction éventuellement avec une condition sur la valeur de x. De plus, nous avons raffiné les ensembles de quantification en des ensembles d'entiers. Nous pouvons donc itérer sur l'ensemble T_synch des valeurs possibles de la variable quantifiée x. Une substitution de type choix non borné est alors dans notre cas implémentée par une substitution de type boucle. Nous itérons ainsi sur les éléments de l'ensemble des valeurs possibles de la variable de la quantification et dès que l'on trouve un élément satisfaisant P, on applique S pour cette valeur. Cette substitution utilise donc un variant qui contient la taille de l'ensemble restant à parcourir.

Exemple d'implémentation

La Figure 7.7 présente un ASTD de type synchronisation quantifiée composé d'un ASTD de type automate. Les trois cas détaillés dans la subsection 4.3.8 sont présents. L'évènement $e1(x)$ représente un évènement non synchronisé dont la variable de quantification apparaît dans les paramètres. L'évènement $e2$ est un évènement synchronisé. L'évènement $e3$ est un évènement non synchronisé qui n'utilise pas la variable de quantification. La traduction en B de l'évènement $e3$ de l'ASTD présenté dans la Figure 7.7 contient donc une substitution **ANY**. La traduction complète de l'ASTD est disponible dans la subsection 4.3.8.

```
SETS
    AutState_aut = {s1, s2, s3, s4} ;
    T_synch = {aa, bb, cc}
    /* l'usage de termes B comportant un seul caractère n'est
    pas autorisé par Atelier B. Ainsi 'aa' représente 'a'.*/

INVARIANT
    State_aut : T_synch --> AutState_aut

OPERATION
e3 =
PRE
    ∃ vv.( vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ))
THEN
    ANY vv WHERE vv ∈ T_synch ∧ State_aut(vv) ∈ dom ( t_aut_e3 ) THEN
        State_aut(vv) := t_aut_e3 ( State_aut(vv) )
    END
END
```

L'évènement $e3$ est implémenté de la manière suivante :

```
CONSTANTS
    C_AutState_aut = 0..3 ;
    C_T_synch = 0..2

OPERATION
e3 =
    VAR compteur IN
        compteur := 0 ;
        WHILE (compteur < 3 )
        DO
            IF C_State_aut(compteur) ∈ dom ( C_t_aut_e3 )
            THEN
                C_State_aut(compteur) := C_t_aut_e3 ( C_State_aut(compteur) ) ;
                compteur := 3
            ELSE
                compteur := compteur + 1
            END
```

```
        INVARIANT
            compteur : 0..3
        VARIANT
            3 - (compteur)
        END
    END
```

Nous avons implémenté l'ensemble de quantification $T_synch = \{aa, bb, cc\}$ par l'ensemble $C_T_synch = 0..2$. La valeur 0 correspond à aa, 1 à bb et 2 à cc. De même, $AutState_aut = \{s1, s2, s3, s4\}$ est implémenté par $C_AutState_aut = 0..3$. Enfin, la substitution **ANY** est implémentée par une boucle basée sur une variable nommée *compteur*. Cette variable représente la valeur courante de C_T_synch qui est testée pour vérifier que le prédicat du **ANY** est vrai. Ce prédicat se retrouve dans la condition du **IF**. La substitution du **ANY** n'est appliquée que quand ce prédicat est vrai, et la variable compteur prend alors une valeur qui permet d'interrompre la boucle. La précondition de l'opération $e3$ abstraite garantit qu'une valeur du compteur permettra d'appliquer la substitution.

7.4 Vers une traduction BPEL

Dans le cadre de leurs travaux, Embe-Jiague *et al.* ont proposé dans [19] une traduction BPEL des ASTD. BPEL (*Business Process Execution Language*) [73] est un langage d'expression de processus métiers utilisant la syntaxe de XML. Il permet de définir des activités, la façon dont elles s'enchaînent, le tout dans le cadre d'une architecture orientée services. Les activités BPEL sont décomposées en deux types. Les activités basiques décrivent les étapes élémentaires du processus. Un exemple d'activité basique est l'affection d'une valeur à une variable. Les activités structurées permettent de composer plusieurs activités ensemble. Elles sont en ce sens similaires aux opérateurs des algèbres de processus. Les types des variables sont ceux disponibles avec XML notamment les types numériques et les chaînes de caractères, les types énumérés et les listes.

Dans son implémentation, une machine B modélisant le comportement dynamique d'une politique de contrôle d'accès ne comporte plus que certains types de substitutions. Comme détaillé dans la section précédente, les seules substitutions disponibles dans une

implémentation sont l'affectation, la séquence, la définition d'une variable locale, la structure conditionnelle **IF**, la boucle **WHILE** et les appels d'opérations externes.

Nous pensons que ces substitutions peuvent être traduites assez intuitivement en activités BPEL. Nous n'avons cependant pas formellement défini de règles de traduction mais les types de données et les substitutions des implémentations B obtenues nous laissent penser que cette transformation est possible.

- L'affectation B peut être traduite par une balise `<assign>` modélisant une activité d'assignation (*assignment*) en BPEL.
- La séquence B peut être traduite par une balise `<sequence>` modélisant une activité de type séquence en BPEL.
- La définition d'une variable locale B peut être traduite par une balise `<variable>` pour définir une variable dont la portée peut être limitée par un `<scope>`.
- Le **IF THEN ELSE** en B peut être traduit par les balises `<if>` `<then>` `<elseif>` `<else>` en BPEL. Les conditions sont alors exprimées dans la balise `<condition>`.
- La boucle **WHILE** en B peut être traduite par une balise `<while>` dans laquelle on ajoute également une balise `<condition>`.

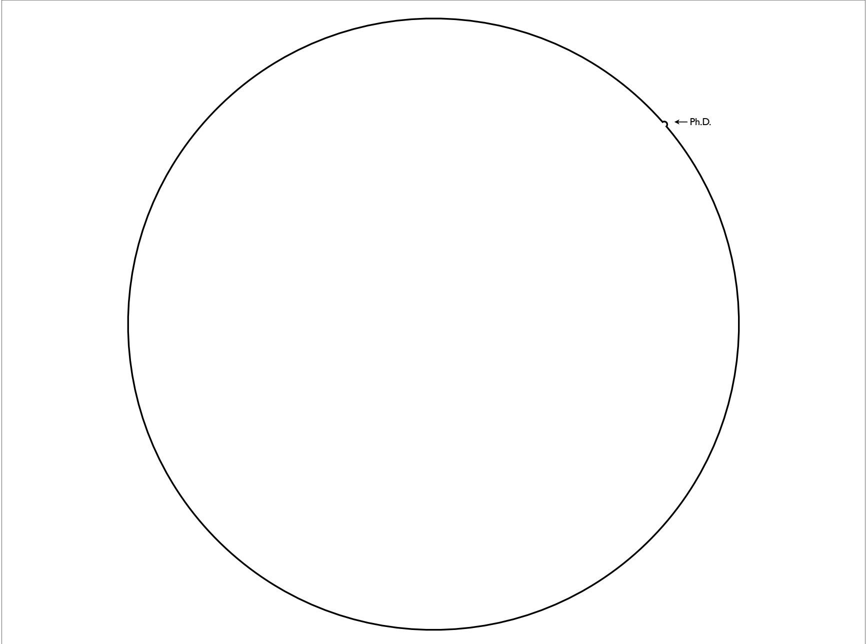
Concernant les types de données des variables de l'implémentation B, les types booléens, entiers, tableaux d'entiers, tableaux de booléens peuvent être codés en XML et sont donc définis en BPEL. Nous ne voyons donc pas, *a priori*, de restriction pour la traduction des implémentations B modélisant l'aspect dynamique d'une spécification de contrôle d'accès en BPEL. Cette traduction nous permettrait de comparer les différentes implémentations des politiques de contrôle d'accès exprimées à l'aide de la notation ASTD.

7.5 Conclusion

Dans ce chapitre, nous avons fait le lien entre les classes du méta-modèle MMB_{sec} présenté dans le chapitre précédent et le méta-modèle MM_{imp} introduit par Embe Jiague *et al.* dans [20]. Nous avons également décrit le diagramme de séquence décrivant le processus d'interrogation du filtre de contrôle d'accès lors d'une requête de l'utilisateur. Nous avons également présenté la façon dont sont implémentées les machines B obtenues par la traduction de spécifications ASTD ainsi que le filtre de contrôle d'accès. Puisque ces machines utilisent seulement un sous-ensemble des types B disponibles et que leur structure

est connue d'avance, un raffinement automatique vers une implémentation peut être envisagé. De plus, une étude rapide du langage BPEL montre qu'une traduction vers BPEL depuis l'implémentation B semble possible. Il faudra tout de même valider cette hypothèse si l'on souhaite utiliser cette approche. Il sera alors intéressant de comparer la traduction d'un ASTD vers BPEL avec la traduction passant par B et son raffinement avant d'être traduite en BPEL.

Conclusion



[65] Ph.D. étape 8 – Mais il ne faut pas oublier d'où l'on vient et prendre du recul. Et continuer à pousser !

L'objectif de cette thèse est de proposer une approche permettant l'implémentation dans un SI d'une politique de contrôle d'accès exprimée en utilisant des notations formelles. Nous pensons que les méthodes formelles permettent de garantir des propriétés par le biais de la vérification de modèles (*model checking*) ou de preuves et permettent également une validation de la spécification avant son implémentation par le biais d'outils d'animation.

Nous avons choisi d'exprimer les politiques de contrôle d'accès avec deux approches. (i) Le langage EB^{3}SEC, d'une part, qui permet d'exprimer les règles de façon unitaire ou sous la forme de patrons puis de les combiner en utilisant des opérateurs issus des algèbres de processus. (ii) La notation ASTD et SecureUML/B, d'autre part, qui permettent de modéliser la politique de contrôle d'accès de manière graphique en séparant les aspects dits statiques des règles dynamiques modélisables plus aisément avec la notation ASTD. Quel que soit le choix fait par le concepteur, la spécification est traduite en B ce qui permet la vérification de propriétés sur la politique ou sur le système combiné avec sa politique de contrôle d'accès. De plus, en utilisant les techniques de raffinement de B, nous pouvons obtenir une implémentation de la politique de sécurité qui respecte certaines contraintes permettant d'inclure cette implémentation dans une architecture orientée services.

Contributions

La Figure c.1 synthétise l'ensemble des contributions de la thèse. Celles-ci sont indiquées par de la couleur par opposition avec les éléments grisés, hors du cadre de cette thèse. Les contributions de cette thèse sont : (i) un état de l'art des traductions entre langages formels ; (ii) les règles de traduction de EB^3 vers ASTD ; (iii) les règles de traduction de ASTD vers Event-B ; (iv) les règles de traduction de ASTD vers B ainsi que leur preuve ; (v) la définition d'un méta-modèle B pour l'expression d'un filtre de contrôle d'accès intégrant l'aspect fonctionnel du système ; (vi) une étude de cas suivant ce méta-modèle dans le domaine médical ; (vii) l'implémentation en utilisant une stratégie de raffinement B de la politique de sécurité.

Nous pensons que ces contributions répondent aux besoins définis par les projets Selkis et EB^{3}SEC dans le cadre de l'implémentation des politiques de contrôle d'accès décrites à l'aide de méthodes formelles. En effet, la solution proposée peut s'intégrer dans un système bancaire ou hospitalier existant, qu'il ait ou non été développé en utilisant une approche

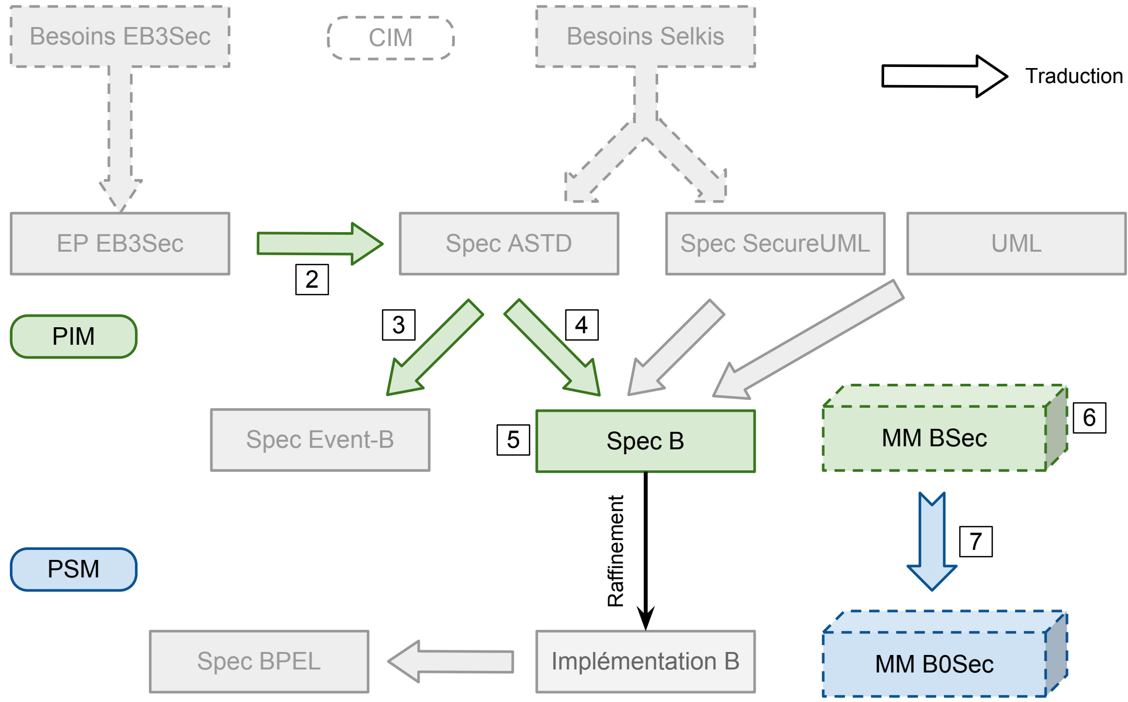


FIGURE c.1 – Contributions de la thèse

formelle. Cependant, l'intégration nous paraît plus simple si le système est développé en partant de zéro. En effet, les systèmes que nous avons pu étudier possèdent déjà une politique de contrôle d'accès intégrée au code source fonctionnel. Il serait donc complexe de retirer ce code éparpillé dans toutes les parties du système pour intégrer le filtre de contrôle d'accès.

Perspectives

Chacune des étapes de cette thèse offre ses propres perspectives d'évolution.

- La traduction de EB^3 vers ASTD (chapitre 2) peut être outillée en adaptant les outils déjà existant pour ces méthodes. Les outils OCamlPAI, une implémentation OCaml de EB^{3}PAI [26], et *i*ASTD [86] peuvent ainsi être fusionnés en un seul outil intégrant les algorithmes de traduction. Cette fusion permettrait de réutiliser les structures de données utilisées pour coder les expressions de processus EB^3 et les ASTD qui sont

extrêmement proches. Cela permettrait également de simplifier l'écriture de spécification ASTD pour *i*ASTD. Les entrées d'*i*ASTD utilisent une notation particulièrement lourde dans l'état actuel. L'utilisation de EB^3 pour l'expression de la spécification initiale serait plus simple dans l'état actuel des choses. Une preuve des règles de traduction de EB^3 vers ASTD est également souhaitable.

– La traduction des ASTD vers Event-B (chapitre 3) peut être intégrée dans eASTD [8], un éditeur graphique pour les ASTD développé en tant qu'extension Eclipse/Rodin. Ceci permettrait de pouvoir valider une spécification ASTD directement avec l'outil d'animation ProB [55].
– La traduction des ASTD vers B (chapitre 4) peut être outillée afin de générer automatiquement des machines B à partir d'une spécification ASTD. Cet outil pourrait être intégré à *i*ASTD. De plus, une étude de la qualité de la traduction doit être menée afin de déterminer si les machines obtenues sont utilisables dans le cadre de preuves automatiques et de vérification de propriétés par *model checking*.
– Afin de rendre encore plus formel le filtre de contrôle d'accès exprimé en B (chapitre 5), une méthode de spécification de son comportement doit être développée. Cette spécification doit définir : l'ordre dans lequel sont interrogées les machines incluses par le filtre ; l'algorithme de calcul de la réponse finale du filtre qui donne ou non l'autorisation pour l'exécution de l'action.
– La stratégie de raffinement (chapitre 7) peut être complétée par une traduction vers BPEL comme le suggère la Figure c.1 afin de comparer la traduction ainsi obtenue avec les travaux de Embe Jiague et al. [23].

Perspectives plus lointaines

Prenons du recul et considérons la thèse comme un ensemble. Imaginons une boîte noire prenant en entrée la spécification ASTD et SecureUML/B d'une politique de sécurité, la spécification UML/B du système, ainsi qu'une liste de propriétés LTL ou CTL à vérifier sur le système. Imaginons maintenant un bouton qui lance la « compilation » du système à partir de ces entrées. Le « compilateur » serait en mesure de rejeter les entrées en fonction des propriétés données, comme un compilateur C rejette le code source s'il ne respecte pas la syntaxe C. Une telle approche automatique est-elle possible ? Probablement, mais

elle nécessite encore beaucoup de travail pour correctement intégrer, vérifier, prouver et implémenter un filtre de contrôle d'accès.

Post Mortem

Je me souviens d'une discussion similaire dans [39], la première thèse que j'ai entièrement lu, et qui dans sa conclusion cite Tony Hoare [45]. Tony lançait alors le défi de la création d'un « compilateur vérificateur » dans les trente prochaines années. Ce défi nous a été lancé en 2003, il nous reste 22 années. Ensuite, nous pourrons partir à la retraite.

À moins que…

Annexes

Annexe A

Étude de cas : Traduction d'un ASTD en Event-B

Dans cette annexe, nous traduisons en Event-B l'ASTD présenté à la Figure A.1 qui décrit un SI de gestion des plaintes de clients dans une entreprise.

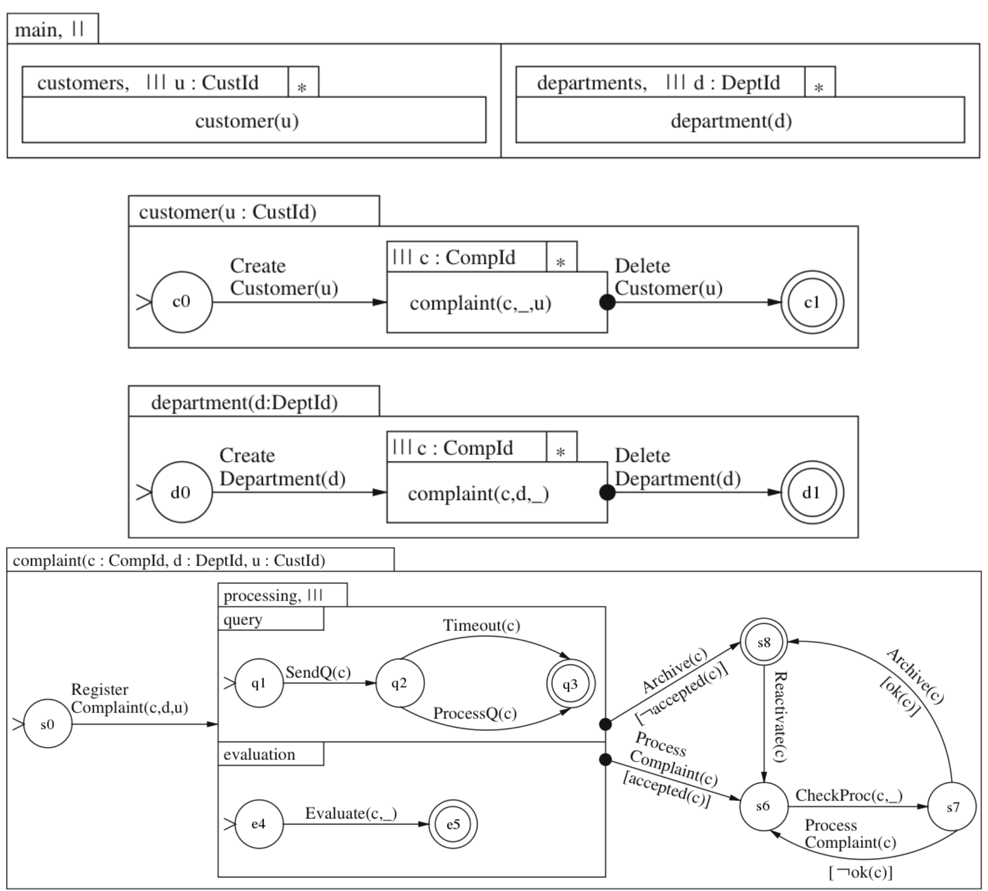


FIGURE A.1 – ASTD décrivant un système de gestion de plaintes de clients

A.1 Contexte commun aux ASTD

Ce contexte introduit des constantes et des ensembles Event-B. Ce contexte est le même pour toutes les traductions d'ASTD et est étendu pour prendre en compte les spécificités de l'ASTD traduit. Il décrit les éléments issus de la sémantique des ASTD et de leurs types.

CONTEXT astd

SETS

SCHOICE State of a choice

SSEQUENCE State of a sequence

SKLEENE State of a Kleene closure

SQCHOICE State of a quantified choice

CONSTANTS

none, right, left

fst, snd

qSome, qNone

isFinalSEQUENCE

isFinalKLEENE

isFinalSYNCHRO

isFinalCHOICE

AXIOMS

axm1 : $partition(SSEQUENCE, \{fst\}, \{snd\})$

A sequence is either in first state or second state

axm2 : $partition(SCHOICE, \{left\}, \{right\}, \{none\})$

A choice state is either not made (none), made on the left or right ASTD

axm3 : $partition(SKLEENE, \{TRUE\}, \{FALSE\})$

A Kleene closure state is either started or not

axm4 : $partition(SQCHOICE, \{qNone\}, \{qSome\})$

A quantified choice state is either not made (qNone), or made (qSome)

axm5 : $isFinalSEQUENCE = \{(fst \mapsto TRUE \mapsto TRUE) \mapsto TRUE, (fst \mapsto TRUE \mapsto FALSE) \mapsto FALSE, (fst \mapsto FALSE \mapsto TRUE) \mapsto FALSE, (fst \mapsto FALSE \mapsto FALSE) \mapsto FALSE, (snd \mapsto TRUE \mapsto TRUE) \mapsto TRUE, (snd \mapsto TRUE \mapsto FALSE) \mapsto FALSE, (snd \mapsto FALSE \mapsto TRUE) \mapsto TRUE, (snd \mapsto FALSE \mapsto FALSE) \mapsto FALSE\}$

This function takes as argument the state of the Sequence, plus left and right isFinal predicates

axm6 : $isFinalKLEENE = \{(FALSE \mapsto TRUE) \mapsto TRUE, (FALSE \mapsto FALSE) \mapsto TRUE, (TRUE \mapsto TRUE) \mapsto TRUE, (TRUE \mapsto FALSE) \mapsto FALSE\}$

This function takes Kleene closure state and isFinal predicate of sub state

axm7 : $isFinalSYNCHRO = \{(TRUE \mapsto TRUE) \mapsto TRUE, (TRUE \mapsto FALSE) \mapsto FALSE, (FALSE \mapsto TRUE) \mapsto FALSE, (FALSE \mapsto FALSE) \mapsto FALSE\}$

axm8 : $isFinalCHOICE = \{(left \mapsto TRUE \mapsto TRUE) \mapsto TRUE, (left \mapsto TRUE \mapsto FALSE) \mapsto TRUE, (left \mapsto FALSE \mapsto TRUE) \mapsto FALSE, (left \mapsto FALSE \mapsto FALSE) \mapsto FALSE, (right \mapsto TRUE \mapsto TRUE) \mapsto TRUE, (right \mapsto TRUE \mapsto FALSE) \mapsto FALSE, (right \mapsto FALSE \mapsto TRUE) \mapsto TRUE, (right \mapsto FALSE \mapsto FALSE) \mapsto FALSE, (none \mapsto TRUE \mapsto TRUE) \mapsto TRUE, (none \mapsto TRUE \mapsto FALSE) \mapsto TRUE, (none \mapsto FALSE \mapsto TRUE) \mapsto TRUE, (none \mapsto FALSE \mapsto FALSE) \mapsto FALSE\}$

END

A.2 Contexte

CONTEXT complaint_ctx
EXTENDS astd
SETS

QuerySTATE

EvalSTATE

ComplaintSTATE

CustomerSTATE

DepartmentSTATE

CONSTANTS

q1, q2, q3 // state in query

e4, e5 // state in evaluation

s0, sProcessing, s6, s7, s8 // state in complaint(c,d,u)

c0, c1, cc // state in customer

d0, d1, dc // state in department

// Transition functions

TransCreateCustomer, TransDeleteCustomer

TransCreateDepartment, TransDeleteDepartment

TransRegisterComplaint, TransSendQ, TransTimeout

TransProcessQ, TransEvaluate, TransArchive, TransReactivate

TransCheckProc, TransProcessComplaint

// Final States

isFinalQuery, isFinalEvaluate, isFinalProcessing

isFinalComplaint, isFinalCustomer, isFinalDepartment

// Initial States

InitQuery, InitEvaluation, InitComplaint, InitCustomer, InitDepartment

AXIOMS

axm1 : $partition(ComplaintSTATE, \{s0\}, \{sProcessing\}, \{s6\}, \{s7\}, \{s8\})$

axm2 : $partition(QuerySTATE, \{q1\}, \{q2\}, \{q3\})$

axm3 : $partition(EvalSTATE, \{e4\}, \{e5\})$

axm4 : $partition(CustomerSTATE, \{c0\}, \{c1\}, \{cc\})$

axm5 : $partition(DepartmentSTATE, \{d0\}, \{d1\}, \{dc\})$

// Transition function definitions

axm6 : $TransCreateCustomer = \{c0 \mapsto cc\}$

axm7 : $TransDeleteCustomer = \{cc \mapsto c1\}$

axm8 : $TransCreateDepartment = \{d0 \mapsto dc\}$

axm9 : $TransDeleteDepartment = \{dc \mapsto d1\}$

axm10 : $TransRegisterComplaint = \{s0 \mapsto sProcessing\}$

axm11 : $TransSendQ = \{q1 \mapsto q2\}$

axm12 : $TransTimeout = \{q2 \mapsto q3\}$

axm13 : $TransProcessQ = \{q2 \mapsto q3\}$

axm14 : $TransEvaluate = \{e4 \mapsto e5\}$

axm15 : $TransArchive = \{sProcessing \mapsto s8, s7 \mapsto s8\}$

axm16 : $TransProcessComplaint = \{sProcessing \mapsto s6, s7 \mapsto s6\}$

axm17 : $TransReactivate = \{s8 \mapsto s6\}$

axm18 : $TransCheckProc = \{s6 \mapsto s7\}$

// Final state functions

axm19 : $isFinalQuery = \{q1 \mapsto FALSE, q2 \mapsto FALSE, q3 \mapsto TRUE\}$

axm20 : $isFinalEvaluate = \{e4 \mapsto FALSE, e5 \mapsto TRUE\}$

axm21 : $isFinalComplaint = \{s0 \mapsto FALSE, sProcessing \mapsto FALSE, s6 \mapsto FALSE, s7 \mapsto FALSE, s8 \mapsto TRUE\}$

axm22 : $isFinalCustomer = \{c0 \mapsto FALSE, cc \mapsto FALSE, c1 \mapsto TRUE\}$

axm23 : $isFinalDepartment = \{d0 \mapsto FALSE, dc \mapsto FALSE, d1 \mapsto TRUE\}$

axm24 : $isFinalProcessing = isFinalSYNCHRO$

// Initial state definitions

axm25 : $InitQuery = q1$

axm26 : $InitEvaluation = e4$

axm27 : $InitComplaint = s0$

axm28 : $InitCustomer = c0$

axm29 : $InitDepartment = d0$

END

A.3 Machine

MACHINE mainComplaint
SEES complaint_ctx
VARIABLES

StateQuery

StateEval

StateComplaint
StateDepartment
StateCustomer
AssociationCustomer
AssociationDepartment

INVARIANTS

inv1 : $StateQuery \in 0\,..\,2 \rightarrow QuerySTATE$

inv2 : $StateEval \in 0\,..\,2 \rightarrow EvalSTATE$

inv3 : $StateComplaint \in 0\,..\,2 \rightarrow ComplaintSTATE$

inv4 : $StateDepartment \in 10\,..\,12 \rightarrow DepartmentSTATE$

inv5 : $StateCustomer \in 20\,..\,22 \rightarrow CustomerSTATE$

inv6 : $AssociationCustomer \in 0\,..\,2 \twoheadrightarrow\!\!\!\!\!\!\,+\; 20\,..\,22$

inv7 : $AssociationDepartment \in 0\,..\,2 \twoheadrightarrow\!\!\!\!\!\!\,+\; 10\,..\,12$

EVENTS

Initialisation

begin

act1 : $StateQuery := 0\,..\,2 \times \{q1\}$

act2 : $StateEval := 0\,..\,2 \times \{e4\}$

act3 : $StateComplaint := 0\,..\,2 \times \{s0\}$

act4 : $StateDepartment := 10\,..\,12 \times \{d0\}$

act5 : $StateCustomer := 20\,..\,22 \times \{c0\}$

act6 : $AssociationCustomer := \varnothing$

act7 : $AssociationDepartment := \varnothing$

end

Event $CreateCustomer \;\widehat{=}$

any

u

where

grd1 : $u \in 20\,..\,22$

grd2 : $StateCustomer(u) \in dom(TransCreateCustomer)$

then

act1 : $StateCustomer(u) := TransCreateCustomer(StateCustomer(u))$

end

Event $DeleteCustomer \;\widehat{=}$

any

u

where

grd1 : $u \in 20 \,..\, 22$

grd2 : $StateCustomer(u) \in dom(TransDeleteCustomer)$

grd3 : $\forall x \cdot x \in dom(AssociationCustomer \rhd \{u\}) \Rightarrow isFinalComplaint(StateComplaint(x)) = TRUE$

then

act1 : $StateCustomer(u) := TransDeleteCustomer(StateCustomer(u))$

act2 : $AssociationCustomer := AssociationCustomer \ntriangleright \{u\}$

end

Event $CreateDepartment \ \widehat{=}$

any

d

where

grd1 : $d \in 10 \,..\, 12$

grd2 : $StateDepartment(d) \in dom(TransCreateDepartment)$

then

act1 : $StateDepartment(d) := TransCreateDepartment(StateDepartment(d))$

end

Event $DeleteDepartment \ \widehat{=}$

any

d

where

grd1 : $d \in 10 \,..\, 12$

grd2 : $StateDepartment(d) \in dom(TransDeleteDepartment)$

grd3 : $\forall x \cdot x \in dom(AssociationDepartment \rhd \{d\}) \Rightarrow isFinalComplaint(StateComplaint(x)) = TRUE$

then

act1 : $StateDepartment(d) := TransDeleteDepartment(StateDepartment(d))$

act2 : $AssociationDepartment := AssociationDepartment \ntriangleright \{d\}$

end

Event $RegisterComplaint \ \widehat{=}$

any

c

d

u

where

grd1 : $c \in 0\,..\,2$

grd2 : $d \in 10\,..\,12$

grd3 : $u \in 20\,..\,22$

grd4 : $StateComplaint(c) \in dom(TransRegisterComplaint)$

grd5 : $StateCustomer(u) = cc$

grd6 : $StateDepartment(d) = dc$

then

act1 : $StateComplaint(c) := TransRegisterComplaint(StateComplaint(c))$

act2 : $StateEval(c) := InitEvaluation$

act3 : $StateQuery(c) := InitQuery$

act4 : $AssociationCustomer(c) := u$

act5 : $AssociationDepartment(c) := d$

end

Event $SendQ \mathrel{\widehat{=}}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $StateQuery(c) \in dom(TransSendQ)$

grd3 : $StateComplaint(c) = sProcessing$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateQuery(c) := TransSendQ(StateQuery(c))$

end

Event $Timeout \mathrel{\widehat{=}}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $StateQuery(c) \in dom(TransTimeout)$

grd3 : $StateComplaint(c) = sProcessing$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateQuery(c) := TransTimeout(StateQuery(c))$

end

Event $ProcessQ \;\widehat{=}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $StateQuery(c) \in dom(TransProcessQ)$

grd3 : $StateComplaint(c) = sProcessing$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateQuery(c) := TransProcessQ(StateQuery(c))$

end

Event $Evaluate \;\widehat{=}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $StateEval(c) \in dom(TransEvaluate)$

grd3 : $StateComplaint(c) = sProcessing$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateEval(c) := TransEvaluate(StateEval(c))$

end

Event $Archive \;\widehat{=}$

any

c

where

grd1 : $c \in 0..2$

grd2 : $((StateComplaint(c) = sProcessing) \wedge (isFinalProcessing(isFinalQuery(StateQuery(c)) \mapsto isFinalEvaluate(StateEval(c))) = TRUE)) \vee (StateComplaint(c) = s7)$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateComplaint(c) := TransArchive(StateComplaint(c))$

end

Event $ProcessComplaint \mathrel{\widehat{=}}$

any

c

where

grd1 : $c \in 0..2$

grd2 : $((StateComplaint(c) = sProcessing) \wedge (isFinalProcessing(isFinalQuery(StateQuery(c)) \mapsto isFinalEvaluate(StateEval(c))) = TRUE)) \vee (StateComplaint(c) = s7)$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateComplaint(c) := TransProcessComplaint(StateComplaint(c))$

end

Event $CheckProc \mathrel{\widehat{=}}$

any

c

where

grd1 : $c \in 0..2$

grd2 : $StateComplaint(c) \in dom(TransCheckProc)$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateComplaint(c) := TransCheckProc(StateComplaint(c))$

end

Event *Reactivate* $\widehat{=}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $StateComplaint(c) \in dom(TransReactivate)$

grd5 : $StateCustomer(AssociationCustomer(c)) = cc$

grd6 : $StateDepartment(AssociationDepartment(c)) = dc$

then

act1 : $StateComplaint(c) := TransReactivate(StateComplaint(c))$

end

Event *lambdaComplaint* $\widehat{=}$

any

c

where

grd1 : $c \in 0\,..\,2$

grd2 : $isFinalComplaint(StateComplaint(c)) = TRUE$

then

act1 : $StateComplaint(c) := InitComplaint$

act2 : $AssociationDepartment := \{c\} \lhd\!\!\!- AssociationDepartment$

act3 : $AssociationCustomer := \{c\} \lhd\!\!\!- AssociationCustomer$

end

Event *lambdaCustomer* $\widehat{=}$

any

u

where

grd1 : $u \in 20\,..\,22$

grd2 : $isFinalCustomer(StateCustomer(u)) = TRUE$

then

act1 : $StateCustomer(u) := InitCustomer$

act2 : $AssociationCustomer := AssociationCustomer \,-\!\!\!\!\rhd \{u\}$

end

Event $lambdaDepartment \widehat{=}$

any

d

where

grd1 : $d \in 10 \,..\, 12$

grd2 : $isFinalDepartment(StateDepartment(d)) = TRUE$

then

act1 : $StateDepartment(d) := InitDepartment$

act2 : $AssociationDepartment := AssociationDepartment \rhd \{d\}$

end

END

Annexe B

Étude de cas : Traduction d'un ASTD en B

Dans cette annexe, nous traduisons en B l'ASTD d'un SI de gestion d'une bibliothèque.

B.1 SI de gestion d'une bibliothèques

L'ASTD présenté à la Figure B.1 est un extrait de l'ASTD qui décrit un SI de gestion de bibliothèque. La traduction de cet extrait en B est présentée ci-dessous. Nous avons choisi de ne présenter qu'un extrait du fait de la complexité de la traduction à la main de cet ASTD.

Cet extrait présente la traduction du processus **Book** qui décrit le cycle de vie d'un livre. La traduction a nécessité une légère modification des paramètres des évènements du processus **loan** en ajoutant *mid* à renew et return. Ceci reflète un problème bien connu en EB^3 et ASTD : les liens induits entre les variables de quantification. Il a été résolu par l'algorithme de la κ-optimisation présenté par Benoît Fraikin *et al.* [27]. Dans le cadre de notre traduction, deux options s'offrent à nous. Soit nous rajoutons manuellement les paramètres manquant de sorte que chaque évènement a l'ensemble des variables quantifiées ou paramètres d'un ASTD en tant que paramètres. Soit nous créons des fonctions permettant de calculer la valeur d'une variable en fonction des valeurs des autres variables. De telles fonctions peuvent se déduire du diagramme entités/relations du système. Nous avons ici choisi la première option.

Nous avons également procédé à une réécriture pour certains prédicats qui étaient trivialement vrais ou faux. Ce genre de réécriture pourrait être calculée par un outil afin de réduire la complexité de la traduction proposée.

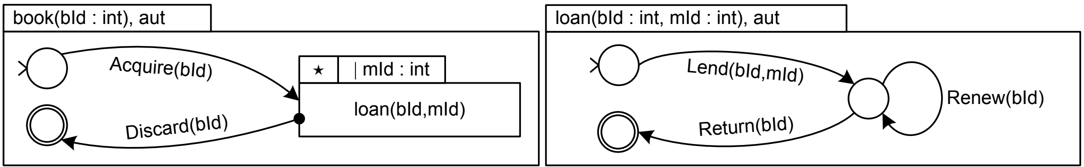


FIGURE B.1 – Extrait d'un ASTD modélisant un SI d'une bibliothèque

B.1.1 Traduction

```
MACHINE
    book(bid)

CONSTRAINTS
    bid : INT

SETS
    AutState_loan = {l0,l1,l2} ;
    AutState_book = {b0,b1,b2} ;
    CALL = {called, notcalled} ;
    QCHOICE = {chosen, notchosen} ;
    KLEENE = {started, notstarted}

CONSTANTS
    init_loan,
    t_loan_Lend,
    t_loan_Renew,
    t_loan_Return,
    SF_loan,
    init_call,
    init_kleene,
    init_Qchoice,
    init_book,
    t_book_Acquire,
    t_book_Discard,
    SF_book

PROPERTIES
    init_loan = l0 &
    t_loan_Lend = {l0 |-> l1} &
```

```
    t_loan_Renew = {l1 |-> l1} &
    t_loan_Return = {l1 |-> l2} &
    SF_loan = {l2} &
    init_call = notcalled &
    init_kleene = notstarted &
    init_Qchoice = notchosen &
    init_book = b0 &
    t_book_Acquire = {b0 |-> b1} &
    t_book_Discard = {b1 |-> b2} &
    SF_book = {b2}

VARIABLES
    State_loan,
    StateCall_loanFromBook,
    StateQchoiceMade_loanFromBook,
    StateQchoiceValue_loanFromBook,
    StateKleene_loanFromBook,
    State_book

INVARIANT
    State_loan : AutState_loan &
    StateCall_loanFromBook : CALL &
    StateQchoiceMade_loanFromBook : QCHOICE &
    StateQchoiceValue_loanFromBook : INT &
    StateKleene_loanFromBook : KLEENE &
    State_book : AutState_book

INITIALISATION
    State_loan := init_loan ||
    StateCall_loanFromBook := init_call ||
    StateQchoiceMade_loanFromBook := init_Qchoice ||
    StateQchoiceValue_loanFromBook :: INT ||
    StateKleene_loanFromBook := init_kleene||
    State_book := init_book

OPERATIONS

    Acquire(bid_book) =
    PRE
        bid_book : INT &
        ( bid_book = bid  & State_book : dom(t_book_Acquire) )
    THEN
        SELECT
            bid_book = bid  & State_book : dom(t_book_Acquire)
        THEN
            State_book := t_book_Acquire(State_book)
        END
    END ;
```

```
Discard(bid_book) =
PRE
    bid_book : INT &
    ( bid_book = bid  & State_book : dom(t_book_Discard) &
        ( StateKleene_loanFromBook = notstarted or
            (
                /* StateQchoiceMade_loanFromBook = notchosen =>
                # val . (val : INT & init_loan : SF_loan) & */
                StateQchoiceMade_loanFromBook = chosen =>
                State_loan : SF_loan
            )
        )
    )
THEN
    SELECT
        ( bid_book = bid  & State_book : dom(t_book_Discard) &
            ( StateKleene_loanFromBook = notstarted or
                (
                    /* StateQchoiceMade_loanFromBook = notchosen =>
                    (StateCall_loanFromBook = notcalled => false &
                    StateCall_loanFromBook = called =>
                    # val . (val : INT & init_loan : SF_loan) ) & */
                    StateQchoiceMade_loanFromBook = chosen =>
                    StateCall_loanFromBook = called =>
                    State_loan : SF_loan
                )
            )
        )
    THEN
        State_book := t_book_Discard(State_book)
    END
END;

Lend(bid_loan,mid_loan) =
PRE
    bid_loan : INT &
    mid_loan : INT &
    State_book = b1 &
    ( ((StateKleene_loanFromBook = notstarted or
      (StateQchoiceMade_loanFromBook = chosen =>
      StateCall_loanFromBook = called => State_loan : SF_loan ) ) &
      (init_Qchoice = notchosen &
      ((init_call = notcalled & init_loan : dom(t_loan_Lend))
      or (init_loan : dom(t_loan_Lend) & bid_loan = bid)) ) ) or
      ((StateQchoiceMade_loanFromBook = notchosen &
      ((init_call = notcalled & init_loan : dom(t_loan_Lend))
      or (init_loan : dom(t_loan_Lend) & bid_loan = bid)) ) or
      mid_loan = StateQchoiceValue_loanFromBook &
      ((StateCall_loanFromBook = notcalled & init_loan : dom(t_loan_Lend))
```

```
          or (State_loan : dom(t_loan_Lend) & bid_loan = bid)))
        )
    THEN
        StateKleene_loanFromBook := started ||
        SELECT
            ((StateKleene_loanFromBook = notstarted or
            (StateQchoiceMade_loanFromBook = chosen =>
            StateCall_loanFromBook = called => State_loan : SF_loan ))
            & (init_Qchoice = notchosen
            & ((init_call = notcalled & init_loan : dom(t_loan_Lend))
            or (init_loan : dom(t_loan_Lend) & bid_loan = bid)) ))
        THEN
            SELECT
                init_Qchoice = notchosen
                & ((init_call = notcalled & init_loan : dom(t_loan_Lend))
                or (init_loan : dom(t_loan_Lend) & bid_loan = bid))
            THEN
                StateQchoiceMade_loanFromBook := chosen ||
                StateQchoiceValue_loanFromBook := mid_loan ||
                SELECT
                    init_call = notcalled & init_loan : dom(t_loan_Lend)
                THEN
                    StateCall_loanFromBook := called ||
                    SELECT
                        (init_loan : dom(t_loan_Lend) & bid_loan = bid)
                    THEN
                        State_loan := t_loan_Lend(init_loan)
                    END
                WHEN
                    init_loan : dom(t_loan_Lend) & bid_loan = bid
                THEN
                    SELECT
                        (init_loan : dom(t_loan_Lend) & bid_loan = bid)
                    THEN
                        State_loan := t_loan_Lend(init_loan)
                    END
                END
            END
        WHEN
            ( (StateQchoiceMade_loanFromBook = notchosen &
            ((init_call = notcalled & init_loan : dom(t_loan_Lend)) or
            (init_loan : dom(t_loan_Lend) & bid_loan = bid)) ) or
                mid_loan = StateQchoiceValue_loanFromBook &
                 ((StateCall_loanFromBook = notcalled & init_loan : dom(t_loan_Lend)) or
                 (State_loan : dom(t_loan_Lend) & bid_loan = bid))
            )
        THEN
            SELECT
                StateQchoiceMade_loanFromBook = notchosen &
```

```
            ((init_call = notcalled & init_loan : dom(t_loan_Lend)) or
            (init_loan : dom(t_loan_Lend) & bid_loan = bid))
        THEN
            StateQchoiceMade_loanFromBook := chosen ||
            StateQchoiceValue_loanFromBook := mid_loan ||
            SELECT
                init_call = notcalled & init_loan : dom(t_loan_Lend)
            THEN
                StateCall_loanFromBook := called ||
                SELECT
                    (init_loan : dom(t_loan_Lend) & bid_loan = bid)
                THEN
                    State_loan := t_loan_Lend(init_loan)
                END
            WHEN
                init_loan : dom(t_loan_Lend) & bid_loan = bid
            THEN
                SELECT
                    (init_loan : dom(t_loan_Lend) & bid_loan = bid)
                THEN
                    State_loan := t_loan_Lend(init_loan)
                END
            END
        WHEN
            mid_loan = StateQchoiceValue_loanFromBook &
            ((StateCall_loanFromBook = notcalled & init_loan : dom(t_loan_Lend))
            or (State_loan : dom(t_loan_Lend) & bid_loan = bid))
        THEN
            SELECT
                StateCall_loanFromBook = notcalled & init_loan : dom(t_loan_Lend)
            THEN
                StateCall_loanFromBook := called ||
                SELECT
                    (init_loan : dom(t_loan_Lend) & bid_loan = bid)
                THEN
                    State_loan := t_loan_Lend(init_loan)
                END
            WHEN
                State_loan : dom(t_loan_Lend) & bid_loan = bid
            THEN
                SELECT
                    (State_loan : dom(t_loan_Lend) & bid_loan = bid)
                THEN
                    State_loan := t_loan_Lend(State_loan)
                END
            END
        END
    END
END
```

```
    /*
    Les traductions de Renew et Return sont similaires à celle de Lend.
    On notera qu'ici les substitutions et prédicats n'ont pas été simplifiés
    à des fins d'illustration de l'approche. Un résultat réduit est
    indispensable pour la lisibilité de l'approche.
    */

END
```

B.1.2 Statistiques

La machine B présentée a été *typecheck*ée par l'atelier B et l'outil génère 140 obligations de preuves pour la spécification incluant toutes les opérations c'est-à-dire *Renew* et *Return*. Ces obligations de preuves sont toutes prouvées automatiquement.

Annexe C

Règles d'inférence des ASTD

Dans cette annexe nous donnons la liste des règles d'inférences des ASTD.

Automates

Soit Ψ tel que :

$$\Psi \triangleq \big((\mathit{final}? \Rightarrow \mathit{final}_{\nu(n_1)}(s)) \wedge g \wedge \sigma' = \sigma \wedge h' = h \lhd\!\!- \{n_1 \mapsto s\}\big)[\Gamma]$$

$$\mathsf{aut}_1 \frac{\delta((\mathsf{loc}, n_1, n_2), \sigma', g, \mathit{final}?) \qquad \Psi}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', \mathit{init}(\nu(n_2)))}$$

$$\mathsf{aut}_2 \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, n_{2b}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} \notin \{\mathsf{H}, \mathsf{H^*}\} \\ \Psi\end{array}}{\begin{array}{l}(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} \\ \qquad (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2b}, h_{init}, \mathit{init}(\nu(n_{2b}))))\end{array}}$$

$$\mathsf{aut}_3 \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} = \mathit{name}(h(n_2)) \\ \Psi\end{array}}{\begin{array}{l}(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} \\ \qquad (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, \mathit{init}(\nu(n_{2_\flat}))))\end{array}}$$

$$\mathsf{aut}_4\ \frac{\delta((\mathsf{tsub},n_1,n_2,\mathsf{H}^*),\sigma',g,\mathit{final}?) \qquad \Psi}{(\mathsf{aut}_\circ,n_1,h,s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ,n_2,h',h(n_2))}$$

$$\mathsf{aut}_5\ \frac{\begin{array}{c}\delta((\mathsf{fsub},n_1,n_{1_\flat},n_2),\sigma',g,\mathit{final}?)\\ \mathit{name}(s)=n_{1_\flat}\\ \Psi\end{array}}{(\mathsf{aut}_\circ,n_1,h,s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ,n_2,h',\mathit{init}(\nu(n_2)))}$$

$$\mathsf{aut}_6\ \frac{s \xrightarrow{\sigma,\Gamma}_{\nu(n)} s'}{(\mathsf{aut}_\circ,n,h,s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ,n,h,s')}$$

ASTD **unaires**

$$\star_1\ \frac{(\mathit{final}_b(s)[\Gamma] \vee \neg \mathit{started}?) \qquad \mathit{init}(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ,\mathit{started}?,s) \xrightarrow{\sigma,\Gamma} (\star_\circ,\mathsf{true},s')} \qquad \star_2\ \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ,\mathsf{true},s) \xrightarrow{\sigma,\Gamma} (\star_\circ,\mathsf{true},s')}$$

$$|{:}_1\ \frac{\mathit{init}(b) \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b s' \qquad v \in T}{(|{:}_\circ,\bot,_) \xrightarrow{\sigma,\Gamma} (|{:}_\circ,v,s')} \qquad |{:}_2\ \frac{s \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b s' \qquad v \neq \bot}{(|{:}_\circ,v,s) \xrightarrow{\sigma,\Gamma} (|{:},v,s')}$$

$$\Rightarrow_1\ \frac{g[\Gamma] \qquad \mathit{init}(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ,\mathsf{false},\mathit{init}(b)) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ,\mathsf{true},s')} \qquad \Rightarrow_2\ \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ,\mathsf{true},s) \xrightarrow{\sigma,\Gamma} (\Rightarrow,\mathsf{true},s')}$$

$$|[]|{:}_1\ \frac{\alpha(\sigma) \notin \Delta \qquad f(v) \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b s'}{(|[]|{:}_\circ,f) \xrightarrow{\sigma,\Gamma} (|[]|{:}_\circ,f \lhd\!\!\!- \{v \mapsto s'\})}$$

$$|[]|{:}_2\ \frac{\alpha(\sigma) \in \Delta \qquad \forall v : T \cdot f(v) \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b f'(v)}{(|[]|{:}_\circ,f) \xrightarrow{\sigma,\Gamma} (|[]|{:}_\circ,f')}$$

ASTD **binaires**

$$|_1 \; \frac{init(l) \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \bot, \bot) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')} \qquad |_2 \; \frac{init(r) \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \bot, \bot) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$

$$|_3 \; \frac{s \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \mathsf{left}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')} \qquad |_4 \; \frac{s \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \mathsf{right}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$

$$\Rightarrow_1 \; \frac{s \xrightarrow{\sigma,\Gamma}_{fst} s'}{(\Rightarrow_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{fst}, s')} \qquad \Rightarrow_2 \; \frac{final_{fst}(s)[\Gamma] \qquad init(snd) \xrightarrow{\sigma,\Gamma}_{snd} s'}{(\Rightarrow_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{snd}, s')}$$

$$\Rightarrow_3 \; \frac{s \xrightarrow{\sigma,\Gamma}_{snd} s'}{(\Rightarrow_\circ, \mathsf{snd}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathsf{snd}, s')} \qquad |[]|_1 \; \frac{\alpha(\sigma) \notin \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s'_l}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s'_l, s_r)}$$

$$|[]|_2 \; \frac{\alpha(\sigma) \notin \Delta \qquad s_r \xrightarrow{\sigma,\Gamma}_r s'_r}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l, s'_r)} \qquad |[]|_3 \; \frac{\alpha(\sigma) \in \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s'_l \qquad s_r \xrightarrow{\sigma,\Gamma}_r s'_r}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s'_l, s'_r)}$$

Appel d'ASTD

$$\mathsf{cal}_1 \; \frac{init(b) \xrightarrow{\sigma,([\vec{x}:=\vec{v}])\lhd\Gamma}_b s'}{(\mathsf{cal}, \bot) \xrightarrow{\sigma,\Gamma} (\mathsf{cal}, s')} \qquad \mathsf{cal}_2 \; \frac{s \xrightarrow{\sigma,([\vec{x}:=\vec{v}])\lhd\Gamma}_b s'}{(\mathsf{cal}, s) \xrightarrow{\sigma,\Gamma} (\mathsf{cal}, s')}$$

Suivi des lecteurs

```
<script type="text/javascript">

  var _gaq = _gaq || [];
  _gaq.push(['_setAccount', 'UA-82049-28']);
  _gaq.push(['_setDomainName', '.jeremilhau.fr']);
  _gaq.push(['_trackPageview']);

  (function() {
    var ga = document.createElement('script');
    ga.type = 'text/javascript'; ga.async = true;
    ga.src = ('https:' == document.location.protocol ?
        'https://ssl' : 'http://www') + '.google-analytics.com/ga.js';
    var s = document.getElementsByTagName('script')[0];
    s.parentNode.insertBefore(ga, s);
  })();

</script>
```

Merci d'avoir lu cette thèse. Malheureusement les scripts disponibles sur Internet pour le suivi des visiteurs d'un site ne fonctionnement pas sur du papier. Passez donc me laisser un message sur cette page : http://jeremilhau.fr/phd/go.php

Bibliographie

[1] Y. Ait-Ameur, M. Baron, N. Kamel, et J.-M. Mota, « Encoding a process algebra using the Event B method, » *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 11, pp. 239–253, 2009.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, et L. Voisin, « An open extensible tool environment for Event-B, » *Lecture Notes in Computer Science*, vol. 4260, p. 588, 2006.

[3] J.-R. Abrial, *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010.

[5] H. Azkia, N. Cuppens-Boulahia, F. Cuppens, et G. Coatrieux, « Modèles formels pour l'analyse et l'expression des exigences de sécurité, » Projet SELKIS, Rapport technique, September 2010.

[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, et S. Weerawarana, « Business process execution language for Web services, » 2003.

[7] S. Act, « Public Law No. 107-204, » *Washington, DC : Government Printing Office*, 2002.

[8] P. Amar, M. Frappier, C. Lartaud, et J. Milhau, « Integrating ASTD in the Rodin Platform, » dans *Rodin User and Developer Workshop 2010*. University of Duesseldorf, September 2010.

[9] A. Allas, « Canada Health Infoway : EHRS Blueprint, » *Health Canada Infoway*, 2006.

[10] N. Benaissa, « La composition des protocoles de sécurité avec la méthode B événementielle, » THESE, Université Henri Poincaré - Nancy I, mai 2010. Disponible à http://tel.archives-ouvertes.fr/tel-00580145/en/

[11] M. Butler, « csp2B : A Practical Approach to Combining CSP and B, » *Formal Aspects of Computing*, vol. 12, pp. 182–198, 2000.

[12] Clearsy, « Atelier B http://www.atelierb.eu/. »

[13] F. Cuppens et A. Miège, « Modelling Contexts in the Or-BAC Model, » dans *Proceedings of the 19th Annual Computer Security Applications Conference*, série ACSAC '03. Washington, DC, USA : IEEE Computer Society, 2003, pp. 416–.

[14] T. O. Consortium, « ORKA — Organizational Control Architecture, » 2010. Disponible à http://www.organisatorische-kontrolle.de/index-en.htm

[15] A. Dardenne, S. Fickas, et A. van Lamsweerde, « Goal-directed concept acquisition in requirements elicitation, » dans *Proceedings of the 6th international workshop on Software specification and design*, série IWSSD '91. Los Alamitos, CA, USA : IEEE Computer Society Press, 1991, pp. 14–21.

[16] M. Delattre et E. Levy, *L'homme qui valait cinq milliards*. Editions Générales First, juin 2008.

[17] A. Dardenne, A. van Lamsweerde, et S. Fickas, « Goal-directed requirements acquisition, » *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3 – 50, 1993. Disponible à http://www.sciencedirect.com/science/article/pii/016764239390021G

[18] Eclipse Consortium, « Eclipse Graphical Modeling Framework (GMF). » Disponible à http://www.eclipse.org/modeling/gmp/?project=gmf

[19] M. Embe Jiague, M. Frappier, F. Gervais, R. Laleau, et R. St-Denis, « Enforcing ASTD access control policies to WS-BPEL processes deployed in a SOA environment, » *International Journal of Systems and Service-Oriented Engineering (IJSSOE)*, vol. 2, no. 2, pp. 37–59, 2011.

[20] M. Embe Jiague, M. Frappier, F. Gervais, R. Laleau, et R. St-Denis, « A Metamodel for the Design of Access-Control Policy Enforcement Managers :

Work in Progress, » dans *Foundations & Practice Of Security (FPS 2011)*, série Lecture Notes in Computer Science, vol. 6888. Springer Berlin / Heidelberg, 2011, pp. 218–226.

[21] M. Embe Jiague, M. Frappier, F. Gervais, P. Konopacki, R. Laleau, J. Milhau, et R. St-Denis, « A four-concern-oriented secure IS development approach, » dans *8th International Joint Conference on e-Business and Telecommunications (ICETE 2011), Seville, Spain, 18-21 July*, I. Press, éditeur, vol. SECRYPT 2011, 2011, pp. 464–471.

[22] A. A. El Kalam, S. Benferhat, A. Miège, R. El Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, et G. Trouessin, « Organization based access control, » dans *POLICY '03 : Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*. Washington, DC, USA : IEEE Computer Society, 2003, p. 120.

[23] M. Embe-Jiague, R. St-Denis, R. Laleau, et F. Gervais, « A BPEL Implementation of a Security Filter, » dans *PhD Symposium of 8th European Conference on Web Services*, 2010.

[24] G. Fey et R. Drechsler, « Minimizing the Number of Paths in BDDs, » dans *Proceedings of the 15th symposium on Integrated circuits and systems design*. Washington, DC, USA : IEEE Computer Society, 2002, pp. 359–. Disponible à http://dl.acm.org/citation.cfm?id=827246.827403

[25] M. Frappier, F. Diagne, et A. Amel Mammar, « Proving Reachability in B using Substitution Refinement, » dans *B 2011 Workshop*, série Electronic Notes in Theoretical Computer Science, Elsevier, éditeur, vol. to appear, 2011.

[26] B. Fraikin et M. Frappier, « EB^{3}PAI : an Interpreter for the EB^3 Specification Language, » dans *5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002), proceedings*, D. Haneberg, G. Schellhorn, et W. Reif, éditeurs, Reisensburg Castle, Günzburg, Germany, juin 2002.

[27] B. Fraikin et M. Frappier, « Efficient Symbolic Execution of Large Quantifications in a Process Algebra, » dans *Formal Methods and Software Engineering*, série Lecture Notes in Computer Science, M. Butler,

M. Hinchey, et M. Larrondo-Petrie, éditeurs. Springer Berlin / Heidelberg, 2007, vol. 4789, pp. 327–344, 10.1007/978-3-540-76650-6-19. Disponible à http://dx.doi.org/10.1007/978-3-540-76650-6_19

[28] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, et M. Ouenzar, « Comparison of Model Checking Tools for Information Systems, » dans *Formal Methods and Software Engineering*, série Lecture Notes in Computer Science, J. Dong et H. Zhu, éditeurs. Springer Berlin / Heidelberg, 2010, vol. 6447, pp. 581–596, 10.1007/978-3-642-16901-4-38. Disponible à http://dx.doi.org/10.1007/978-3-642-16901-4_38

[29] B. Fraikin, M. Frappier, et R. Laleau, « State-based versus event-based specifications for information systems : a comparison of B and EB^3, » *Software and Systems Modeling*, vol. 4, pp. 236–257, 2005.

[30] M. Frappier, B. Fraikin, R. Laleau, et M. Richard, « Automatic Production of Information Systems, » dans *AAAI Symposium on Logic-Based Program Synthesis*, Stanford University, Stanford, CA, mars 2002, p. 7. Disponible à http://www.dmi.usherb.ca/~frappier/SSSMFrappier602.pdf

[31] B. Fraikin, F. Gervais, M. Frappier, R. Laleau, et M. Richard, « Synthesizing Information Systems : the APIS Project, » dans *First International Conference on Research Challenges in Information Science (RCIS)*, C. Rolland, O. Pastor, et J.-L. Cavarero, éditeurs, Ouarzazate, Morocco, avril 2007, pp. 73–84.

[32] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, et R. St-Denis, « Extending statecharts with process algebra operators, » *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 285–292, October 2008.

[33] M. Frappier, F. Gervais, R. Laleau, et B. Fraikin, « Algebraic State Transition Diagrams, » Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, Rapport technique 24, juin 2008. Disponible à http://www.dmi.usherb.ca/~frappier/Papers/astd2008.pdf

[34] D. F. Ferraiolo, D. R. Kuhn, et R. Chandramouli, *Role-Based Access Control*. Norwood, MA, USA : Artech House, Inc., 2003.

[35] M. D. Fraser, K. Kumar, et V. K. Vaishnavi, « Informal and formal requirements specification laguages : Bridging the gap, » *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 454–465, 1991.

[36] M. Frappier et R. Laleau, « Proving Event Ordering Properties for Information Systems, » dans *ZB 2003 : Formal Specification and Development in Z and B*, série Lecture Notes in Computer Science, D. Bert, J. Bowen, S. King, et M. Waldén, éditeurs. Springer Berlin / Heidelberg, 2003, vol. 2651, pp. 628–644.

[37] M. Fujita, P. McGeer, et J.-Y. Yang, « Multi-Terminal Binary Decision Diagrams : An Efficient Data Structure for Matrix Representation, » *Formal Methods in System Design*, vol. 10, pp. 149–169, 1997.

[38] M. Frappier et R. St-Denis, « EB^3 : an entity-based black-box specification method for information systems, » *Software and Systems Modeling*, vol. 2, pp. 134–149, 2003.

[39] F. Gervais, « Combinaison de spécifications formelles pour la modélisation des systèmes d'information, » Thèse de doctorat, Conservatoire national des arts et métiers, 2006.

[40] F. Gervais, M. Frappier, et R. Laleau, « Generating relational database transactions from EB^3 attribute definitions, » *Software and Systems Modeling*, vol. 8, pp. 423–445, 2009.

[41] D. Harel, « Statecharts : A visual formalism for complex systems, » *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[42] P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, et B. Walters, « Questions and Answers about Ten Formal Methods, » dans *In Proc. 4th Int. Workshop on Formal Methods for Industrial Critical Systems*, 1999, pp. 179–203.

[43] J. Hopcroft, R. Motwani, et J. Ullman, *Introduction to automata theory, languages, and computation*. Addison-wesley Reading, MA, 1979.

[44] C. A. R. Hoare, « CSP–Communicating Sequential Processes, » *Prenctice Hall*, 1985.

[45] T. Hoare, « The Verifying Compiler : A Grand Challenge for Computing Research, » *Journal of the ACM*, vol. 50, p. 2003, 2003.

[46] A. 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. » Thèse, Université Joseph-Fourier - Grenoble I, 2006.

[47] A. Idani, M.-A. Labiadh, et Y. Ledru, « Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B, » *Ingénierie des Systèmes d'Information*, vol. 15, no. 3, pp. 87–112, 2010.

[48] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, et P. Valduriez, « ATL : a QVT-like transformation language, » dans *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, série OOPSLA '06. New York, NY, USA : ACM, 2006, pp. 719–720.

[49] F. Jouault et I. Kurtev, « Transforming Models with ATL, » dans *Satellite Events at the MoDELS 2005 Conference*, série Lecture Notes in Computer Science, vol. 3844. Springer Berlin / Heidelberg, 2006, pp. 128–138. Disponible à http://www.springerlink.com/content/7143g735r4j59463/

[50] J. Jürjens, « UMLsec : Extending UML for Secure Systems Development, » dans *«UML» 2002 — The Unified Modeling Language*, série Lecture Notes in Computer Science, J.-M. Jézéquel, H. Hussmann, et S. Cook, éditeurs. Springer Berlin / Heidelberg, 2002, vol. 2460, pp. 1–9.

[51] P. Konopacki, M. Frappier, et R. Laleau, « Modélisation de politiques de sécurité à l'aide d'une algèbre de processus, » dans *INFORSID*, 2009, pp. 295–310.

[52] P. Konopacki, M. Frappier, et R. Laleau, « Modélisation de politiques de sécurité à l'aide d'une algèbre de processus. Présentation de la méthode EB^{3}SEC, » *Ingénierie des Systèmes d'Information*, vol. 15, no. 3, pp. 113–136, 2010.

[53] P. Konopacki, M. Frappier, et R. Laleau, « Expressing Access Control Policies with an Event-Based Approach, » dans *Advanced Information Systems Engineering Workshops*, série Lecture Notes in Business Information

Processing, W. Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, C. Szyperski, C. Salinesi, et O. Pastor, éditeurs. Springer Berlin Heidelberg, 2011, vol. 83, pp. 607–621.

[54] A. L1110-4, « République Française - Code de la santé publique, » août 2004. Disponible à http://www.legifrance.gouv.fr/affichCodeArticle.do?idArticle=LEGIARTI000006685746&cidTexte=LEGITEXT000006072665&dateTexte=20090720

[55] M. Leuschel et M. Butler, « ProB : A Model Checker for B, » dans *FME 2003 : Formal Methods*, série Lecture Notes in Computer Science, K. Araki, S. Gnesi, et D. Mandrioli, éditeurs. Springer Berlin / Heidelberg, 2003, vol. 2805, pp. 855–874.

[56] M. Leuschel et M. J. Butler, « ProB : an automated analysis toolset for the B method, » *STTT*, vol. 10, no. 2, pp. 185–203, 2008.

[57] T. Lodderstedt, D. A. Basin, et J. Doser, « SecureUML : A UML-Based Modeling Language for Model-Driven Security, » dans *5th International Conference on The Unified Modeling Language (UML)*, série LNCS, vol. 2460. Springer, 2002, pp. 426–441.

[58] K. Lano, D. Clark, et K. Androutsopoulos, « UML to B : Formal Verification of Object-Oriented Models, » dans *Integrated Formal Methods*, série Lecture Notes in Computer Science, vol. 2999. Springer, 2004, pp. 187–206.

[59] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, et M.-A. Labiadh, « Taking into Account Functional Models in the Validation of IS Security Policies, » dans *Advanced Information Systems Engineering Workshops*, série Lecture Notes in Business Information Processing, C. Salinesi, O. Pastor, W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, et C. Szyperski, éditeurs. Springer Berlin Heidelberg, 2011, vol. 83, pp. 592–606.

[60] R. Laleau et A. Mammar, « An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations, » dans *Proceedings of the 15th IEEE international conference on Automated software engineering*, série ASE '00. Washington, DC, USA : IEEE Computer Society, 2000, pp. 269–272.

[61] A. Mammar, M. Frappier, et F. Diagne, « A proof-based approach to verifying reachability properties, » dans *Proceedings of the 2011 ACM Symposium on Applied Computing*, série SAC '11. New York, NY, USA : ACM, 2011, pp. 1651–1657.

[62] J. Milhau, B. Fraikin, et M. Frappier, « Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions, » dans *Integrated Formal Methods*, série Lecture Notes in Computer Science, M. Leuschel et H. Wehrheim, éditeurs. Springer Berlin / Heidelberg, 2009, vol. 5423, pp. 337–351.

[63] J. Milhau, M. Frappier, F. Gervais, et R. Laleau, « Systematic translation of EB3 and ASTD specifications in B and EventB, » Université de Sherbrooke, Rapport technique 30 v3.0, 2010.

[64] J. Milhau, M. Frappier, F. Gervais, et R. Laleau, « Systematic Translation Rules from ASTD to Event-B, » dans *Integrated Formal Methods*, série Lecture Notes in Computer Science, D. Méry et S. Merz, éditeurs. Springer Berlin / Heidelberg, 2010, vol. 6396, pp. 245–259.

[65] M. Might, « The illustrated guide to a Ph.D. » Texte librement adapté, Images sous licence Creative Commons Attribution-NonCommercial 2.5 - http://matt.might.net/articles/phd-school-in-pictures/.

[66] R. Milner, *Communication and concurrency*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1989.

[67] J. Milhau, « Génération automatique de messages d'erreurs pour l'exécution symbolique d'expressions de processus EB^3, » Mémoire de maîtrise, Université de Sherbrooke, novembre 2009.

[68] J. Milhau, A. Idani, R. Laleau, M. A. Labiadh, Y. Ledru, et M. Frappier, « Combining UML, ASTD and B for the Formal Specification of an Access Control Filter, » *In International NASA Journal on Innovations in Systems and Software Engineering (ISSE), Special Issue of UMLFM 2011 workshop*, to be published, 2011.

[69] A. Mammar et R. Laleau, « Implémentation JAVA d'une spécification B : Application aux bases de données, » *Technique et Science Informatiques*, vol. 27, no. 5, pp. 537–570, 2008.

[70] E. Meyer et J. Souquières, « A Systematic Approach to Transform OMT Diagrams to a B Specification, » dans *FM'99 — Formal Methods*, série Lecture Notes in Computer Science, J. Wing, J. Woodcock, et J. Davies, éditeurs. Springer Berlin / Heidelberg, 1999, vol. 1708, pp. 706–706.

[71] G. Neumann et M. Strembeck, « An approach to engineer and enforce context constraints in an RBAC environment, » dans *Proceedings of the eighth ACM symposium on Access control models and technologies*, série SACMAT '03. New York, NY, USA : ACM, 2003, pp. 65–79.

[72] OASIS, *eXtensible Access Control Markup Language (XACML) Version 2.0*. OASIS, 2005. Disponible à http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[73] OASIS, *Web Services Business Process Execution Language Version 2.0*. OASIS, 2007. Disponible à http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

[74] F. Paterno, *Model-Based Design and Evaluation of Interactive Applications*. London, UK : Springer-Verlag, 1999.

[75] S. Preda, N. Cuppens-Boulahia, F. Cuppens, J. Garcia-Alfaro, et L. Toutain, « Model-Driven Security Policy Deployment : Property Oriented Approach, » dans *International Symposium on Engineering Secure Software and Systems (ESSOS'10)*, série LNCS, vol. 5965. Springer, 2010, pp. 123–139.

[76] A. Pnueli, « The temporal logic of programs, » dans *Foundations of Computer Science, 18th Annual Symposium on*, 1977, pp. 46–57.

[77] J. Rumbaugh, I. Jacobson, et G. Booch, *The unified modeling language*. University Video Communications, 1996.

[78] P. Rao, D. Lin, E. Bertino, N. Li, et J. Lobo, « An algebra for fine-grained integration of XACML policies, » dans *Proceedings of the 14th ACM sympo-*

sium on Access control models and technologies, série SACMAT '09. New York, NY, USA : ACM, 2009, pp. 63–72.

[79] C. Snook et M. Butler, « U2B - A tool for translating UML-B models into B, » dans *UML-B Specification for Proven Embedded Systems Design*, J. Mermet, éditeur, 2004.

[80] C. Snook et M. Butler, « UML-B : Formal modeling and design aided by UML, » *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, pp. 92–122, 2006.

[81] C. Snook et M. Butler, « UML-B and Event-B : an integration of languages and tools, » dans *The IASTED International Conference on Software Engineering - SE2008*, February 2008.

[82] M. Said, M. Butler, et C. Snook, « Language and Tool Support for Class and State Machine Refinement in UML-B, » dans *FM 2009 : Formal Methods*, série Lecture Notes in Computer Science, A. Cavalcanti et D. Dams, éditeurs. Springer Berlin / Heidelberg, 2009, vol. 5850, pp. 579–595.

[83] R. Sandhu, E. Coyne, H. Feinstein, et C. Youman, « Role-based access control models, » *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[84] E. Sekerinski, « Verifying Statecharts with State Invariants, » dans *13th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE, 2008, pp. 7–14.

[85] R. Sandhu et Q. Munawer, « How to do discretionary access control using roles, » dans *Proceedings of the third ACM workshop on Role-based access control*, série RBAC '98. New York, NY, USA : ACM, 1998, pp. 47–54. Disponible à http://doi.acm.org/10.1145/286884.286893

[86] K. Salabert, J. Milhau, B. Fraikin, M. Frappier, F. Gervais, et R. Laleau, « iASTD : un interpréteur pour les ASTD, » dans *Actes AFADL 2010*, 2010, pp. 3–6.

[87] E. Sekerinski et R. Zurob, « Translating Statecharts to B, » dans *Integrated Formal Methods*, série Lecture Notes in Computer Science, vol. 2335. Springer Berlin / Heidelberg, 2002, pp. 128–144.

[88] M. Toahchoodee, I. Ray, K. Anastasakis, G. Georg, et B. Bordbar, « Ensuring spatio-temporal access control for real-world applications, » dans *Proceedings of the 14th ACM symposium on Access control models and technologies*, série SACMAT '09. New York, NY, USA : ACM, 2009, pp. 13–22.

[89] W. M. P. Van Der Aalst, « The Application of Petri Nets to Workflow Management, » *The Journal of Circuits, Systems and Computers*, vol. 8, no. 1, pp. 21–66, 1998.