COMBINAISON DE MÉTHODES FORMELLES POUR LA SPÉCIFICATION DE SYSTÈMES INDUSTRIELS

par

Thomas Fayolle

Thèse présentée au Département d'informatique
en vue de l'obtention du grade de philosophiæ doctor (Ph.D.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE
Sherbrooke, Québec, Canada

FACULTÉ DES SCIENCES
UNIVERSITÉ PARIS EST CRÉTEIL
Créteil, France,

8 mai 2017

Table des matières

Table des matières **iii**

Introduction **1**

1 Contexte et état de l'art **7**

1.1 Les langages formels de spécification 7

1.1.1 Les ASTD 7

1.1.2 La méthode B 9

1.1.3 La méthode Event-B 12

1.2 Le raffinement 13

1.2.1 Les spécifications basées sur les événements 13

1.2.2 Les spécifications basées sur les états 15

1.2.3 Discussion et positionnement 17

1.3 Les combinaisons de méthodes formelles 18

1.3.1 iUML-B 18

1.3.2 CSB||B 19

1.3.3 CSP2B 21

1.3.4 Circus 22

1.4 Utilisation des méthodes formelles pour la spécification de systèmes ferroviaires 24

1.4.1 Test de méthodes 24

1.4.2 Modélisation de système 25

2 Définition formelle de la méthode de spécification 29
2.1 Présentation de la méthode 29
2.1.1 La spécification ASTD 30
2.1.2 La spécification Event-B 30
2.1.3 La spécification B classique 31
2.1.4 Un exemple : Modélisation d'un système de contrôle de température avec la méthode 32
2.2 Vérification formelle de la cohérence 34
2.2.1 Objectif 34
2.2.2 Traduction formelle 35
2.2.3 Preuve de correction de la méthode 42
2.3 Raffinement 47
2.3.1 Définition du raffinement combiné 47
2.3.2 Utilisation du raffinement pour simplifier la preuve 60
2.4 Outillage de la méthode 67
2.4.1 Un éditeur d'ASTD 67
2.4.2 Un traducteur d'ASTD 68
2.4.3 Atelier B et Rodin 69

3 Correction de la traduction 71
3.1 Introduction 71
3.2 Présentation de Coq 73
3.3 Formalisation 74
3.3.1 Sémantique opérationnelle des automates 75
3.3.2 Sémantique opérationnelle pour un sous-ensemble du langage B 81
3.3.3 Règles de traduction 86
3.3.4 Théorème à prouver 91
3.4 Preuve 94
3.4.1 Induction mutuelle 95
3.4.2 Lemmes 96
3.4.3 Preuve du théorème 97
3.5 Conclusion et perspectives 100

4 Règles de simplification des ASTD **103**
4.1 Introduction . . . 103
4.1.1 Généralités . . . 105
4.1.2 Notations . . . 107
4.2 Suppression des transitions arrivant à un état initial . . . 109
4.2.1 Présentation de la règle . . . 109
4.2.2 Formalisation de la règle . . . 110
4.2.3 Esquisse de la preuve . . . 111
4.3 Fermeture de Kleene . . . 113
4.3.1 Présentation de la règle . . . 113
4.3.2 Formalisation de la règle . . . 115
4.3.3 Esquisse de la preuve . . . 116
4.4 Choix . . . 119
4.4.1 Présentation de la règle . . . 119
4.4.2 Formalisation . . . 120
4.4.3 Esquisse de la preuve . . . 121
4.5 Automates . . . 125
4.5.1 Présentation de la règle . . . 125
4.5.2 Formalisation . . . 125
4.5.3 Esquisse de la preuve . . . 129
4.6 Conclusion . . . 132

5 Cas d'étude **133**
5.1 Modélisation d'un système de contrôle automatique des trains . . . 134
5.1.1 Notions spécifiques au cas d'étude . . . 134
5.1.2 Présentation . . . 136
5.1.3 Spécifications . . . 137
5.1.4 Comparaison des modélisations . . . 149
5.2 Modélisation d'une machine à hémodialyse . . . 155
5.2.1 Notions spécifiques . . . 156
5.2.2 Spécifications . . . 157
5.2.3 Conclusion . . . 164

Conclusion **167**

A Sémantique opérationnelle des ASTD **171**

A.1 Automates 171
A.2 Sequence 173
A.3 Choice 173
A.4 Kleene closure 174
A.5 Parameterized synchronization 175
A.6 Quantified choice 176
A.7 Quantified Synchronization 176
A.8 Guard 177

Introduction

Contexte

Dans un contexte industriel, un système automatique est généralement plus efficace qu'un opérateur humain. Il est capable d'effectuer plus rapidement des tâches répétitives et, s'il est bien conçu, il permet d'éviter les erreurs d'inattention de l'humain. Afin de préciser cette affirmation, elle est illustrée ici par l'exemple des systèmes ferroviaires.

Les systèmes ferroviaires (métro, trains, etc...) sont indispensables pour fluidifier les déplacements dans les zones urbaines et entre les zones urbaines. Dans des zones urbaines denses, ces systèmes sont souvent proches de la saturation. Afin d'éviter les aléas de la conduite humaine, la solution de l'automatisation permet d'augmenter significativement la fréquence des trains.

L'automatisation de l'exploitation d'un système ferroviaire consiste à confier à un logiciel (ou à un système électronique) une partie ou la totalité du fonctionnement de ce système. L'automatisation partielle consiste à confier le pilotage du train au logiciel, mais un conducteur reste présent, notamment pour l'échange voyageur. C'est par exemple le cas du PA135 [1] et des systèmes de type *CBTC* [2] pour les métros ou du

1. Le système PA135 est un système électronique de contrôle des trains qui utilise un tapis de pilotage (appelé *grecque*). Le tapis permet de contrôler la vitesse du train en respectant la signalisation.

2. Le *CBTC* (*Communication Based Train Control System*) est un système de contrôle automatique des trains basé sur la communication entre un sous-système à bord du train qui envoie sa position à un sous-système au sol qui fixe une limite de déplacement en fonction des positions de tous les trains. Ce système est donc plus souple que le **PA**135

système *ERTMS*[3] pour les trains. Des systèmes d'automatisation totale permettent quand à eux de supprimer complètement le conducteur. C'est par exemple le cas du système *VAL*, premier système au monde de pilotage automatisé, déployé d'abord à Lille (France) puis sur plus de dix lignes dans le monde depuis, ou du système *SAET* (*Système Automatisé d'Exploitation des Trains*, système basé également sur la norme *CBTC*) des lignes 1 et 14 du métro parisien.

L'automatisation de l'exploitation d'un système de trains permet entre autres d'optimiser l'utilisation d'une ligne ferroviaire en augmentant la fréquence des trains. Mais elle permet également d'éviter l'erreur humaine et donc d'augmenter la sécurité du système. L'accident de Saint-Jacques-de-Compostelle[4] en 2013, par exemple, aurait pu être évité avec un système de contrôle des trains.

Le raisonnement tenu ici peut être étendu à beaucoup de systèmes industriels, surtout ceux pour lesquels la présence d'erreur serait critique. Les systèmes concernés ne manquent pas : on peut parler de l'aéronautique, mais également des systèmes automobiles qui commencent à prendre en charge des fonctions parfois critiques ou des systèmes médicaux dont les dysfonctionnements peuvent évidemment être critiques pour les patients.

Problématique

Lors de l'automatisation des systèmes, le problème de l'erreur humaine est alors déplacé au moment de la conception du système de contrôle. En effet, il faut s'assurer au moment de la spécification que le système utilisé vérifie les propriétés de sécurité nécessaires. Une solution pour aider à garantir la sécurité d'un système au moment de sa conception est d'utiliser des méthodes formelles. Par exemple, la norme EN50128 [CEN01] qui régit les processus de développement de la partie logicielle des

3. Le système *ERTMS* (*European Rail Traffic Management System*) est un système de gestion du trafic ferroviaire européen dont l'objectif est d'harmoniser les systèmes de signalisation. La norme *ERTMS* contient également des spécifications pour permettre l'automatisation partielle des trains.

4. Le 24 juillet 2013, près de Saint-Jacques-de-Compostelle, un train à grande vitesse aborde une courbe à une vitesse de 195 km/h alors qu'elle était limitée à 80 km/h et déraille causant la mort de 79 personnes. Cet accident a eu lieu quelques kilomètres après la fin de couverture par le système *ERTMS* qui aurait pu éviter cet accident.

systèmes de signalisation ferroviaire recommande fortement l'utilisation des méthodes formelles.

Cependant, les méthodes formelles utilisent des notations mathématiques parfois complexes qui nécessitent la collaboration entre des ingénieurs connaissant les langages formels et des ingénieurs qui connaissent le fonctionnement du système modélisé (ingénieurs aéronautiques, ingénieurs ferroviaires, etc...). Nous supposons que cette collaboration peut être facilitée par des langages de spécification graphiques, qui permettent généralement de faciliter la compréhension des spécifications.

Objectifs

L'objectif de ce travail de thèse est de définir une méthode de spécification formelle utilisant un langage graphique et qui permet la spécification de systèmes industriels. Les langages graphiques permettent de spécifier facilement le comportement d'un système mais sont moins puissants pour spécifier facilement les évolutions des données. La méthode définie dans cette thèse doit donc selectionner une méthode de spécification graphique et définir comment cette méthode de spécification doit être complétée pour permettre la spécification complète d'un système.

Les systèmes industriels sont des systèmes souvent complexes. La mise en œuvre de la méthode ne doit donc pas générer une augmentation trop forte de la complexité. De plus, la spécification par étapes du système est une solution pour gérer la complexité. Pour permettre la spécification incrémentale, les spécifications réalisées à l'aide de la méthode définie dans cette thèse doivent donc pouvoir être raffinées. Enfin, puisque la méthode de spécification doit pouvoir modéliser des systèmes critiques, la sûreté du système doit être garantie. Elle doit permettre de prouver les propriétés de sécurité du système et assurer à chaque étape de la mise en œuvre que la méthode n'introduit pas d'incohérence.

Méthodologie

Les ASTD sont un langage de spécification développé en 2008 par Fraikin, Frappier, Gervais et Laleau [FGL+08]. Ils sont basés sur la combinaison des diagrammes d'état

à la Harel [Har87] avec des opérateurs de l'algèbre de processus EB^3 [FSD03]. Les diagrammes d'état permettent de spécifier graphiquement le comportement du système et les opérateurs des algèbres de processus permettent d'étendre le comportement à plusieurs processus et de factoriser certaines spécifications. La spécification graphique permet donc de faciliter la compréhension des spécifications par des non spécialistes et les opérateurs des algèbres de processus permettent de modéliser un système comportant plusieurs entités. Les ASTD permettent de décrire un ordonnancement d'actions, mais ne permettent pas de décrire les changements d'état des variables du système. Pour décrire le comportement complet d'un système, il est donc nécessaire de les combiner avec une méthode de spécification basée sur les états.

La méthode définie dans cette thèse pour permettre la spécification complète d'un système, combine les ASTD avec les langages B et Event-B. Dans cette méthode, les ASTD sont utilisés pour décrire l'ordonnancement des actions du système et le langage Event-B est utilisé pour décrire les variables d'état du système et leur évolution. Les deux spécifications sont ensuite traduites dans le langage B classique, afin de vérifier la cohérence du système. Le choix du langage Event-B pour la spécification du modèle de données et de son évolution se justifie principalement parce que la définition du raffinement en Event-B est plus proche de la définition du raffinement des ASTD. Le langage B permet d'utiliser les obligations de la méthode B pour montrer la cohérence de la spécification. Pour permettre la spécification par étapes de systèmes complexes tout en préservant la cohérence des spécifications, un raffinement combiné des spécifications est défini.

La vérification de la propriété de cohérence nécessite de plonger les deux langages dans une même sémantique. Ici, les deux langages sont traduits dans le langage B. Cependant, puisque cette traduction est utilisée dans le cadre d'une méthode formelle, et que cette méthode doit permettre la spécification de systèmes critiques, il est important de s'assurer que les traductions sont correctes.

De plus, la spécification écrite en B classique, qui permet de prouver la cohérence de la spécification, génère un nombre important d'obligation de preuves qui croît avec la taille du modèle. Puisque les systèmes industriels sont parfois complexes, il est donc très important de limiter ce nombre d'obligations de preuve.

Enfin, le meilleur moyen d'évaluer une méthode de modélisation est de l'utiliser pour modéliser des systèmes. Afin de tester la méthode, d'évaluer son apport et de déceler ses points forts et les points qu'il faut améliorer, il est important de présenter des cas d'études de la méthode.

Contributions

La principale contribution présentée dans ce mémoire est la définition d'une méthode de spécification combinant les ASTD avec les langages B et Event-B. Cette méthode permet de spécifier un système en séparant l'aspect de l'ordonnancement des actions de l'aspect de la spécification du modèle de données. L'idée de cette méthode est de définir l'ordonnancement des actions dans le langage ASTD et le modèle de données et son évolution dans le langage Event-B. Le langage B permet de vérifier la cohérence de la spécification. Le chapitre 2 contient la définition de cette méthode. Il contient également une preuve de la cohérence de la méthode, c'est-à-dire une preuve qu'un système spécifié en utilisant la méthode exécutera ses actions dans l'ordre défini en ASTD et que l'évolution de son modèle de données vérifie la spécification définie en Event-B. Enfin, pour permettre la spécification de systèmes complexes, le chapitre 2 contient également une définition de raffinement combiné.

Pour vérifier la cohérence de la spécification, les ASTD doivent être traduits dans le langage B. Pour que le méthode soit formelle, cette traduction doit être validée. Le chapitre 3 contient une preuve de la correction de la traduction des ASTD vers le langage B réalisée à l'aide de l'assistant de preuve Coq. Pour réaliser cette traduction, les sémantiques des ASTD et du sous-ensemble du langage B nécessaire à cette traduction ont été traduites en Coq. De plus, une implémentation des règles de traduction décrites dans la thèse de Milhau [Mil11] a été réalisée dans cet assistant de preuve, qui a ensuite été utilisé pour prouver la correction des règles de traduction. Ce travail aurait également pu être réalisé avec l'assistant de preuve Isabelle/HOL.

La preuve de cohérence des spécifications du système est souvent complexe. Des solutions doivent être mises en place pour simplifier cette preuve. Dans le chapitre 4, des règles de simplification des ASTD sont proposées, ainsi qu'une esquisse de la preuve de ces règles de simplification.

Enfin, la modélisation d'un cas d'étude ferroviaire a permis de mettre au point la méthode. Ce cas d'étude a également permis de mesurer l'impact des différentes améliorations de la méthode proposées dans cette thèse. Il a enfin fait l'objet d'une publication [FFLG16]. De plus, pour valider la méthode, nous avons proposé une modélisation du cas d'étude proposé lors de la conférence ABZ2016. Il s'agissait alors de spécifier une machine à Hémodialyse [FFGL16, Mas15]. Ces deux cas d'étude sont présentés dans le chapitre 5.

Chapitre 1

Contexte et état de l'art

1.1 Les langages formels de spécification

1.1.1 Les ASTD

La notation ASTD [FGL+08] (*Algebraic State Transition Diagram*) est une notation graphique formelle combinant le formalisme des diagrammes d'état de Harel et le formalisme de l'algèbre de processus EB^3 [FSD03]. Elle est caractérisée par une structure et un état. La structure graphique modélise la sémantique et l'état représente les évolutions de l'ASTD au cours de son exécution. Des règles de sémantique formelle ont été définies pour les ASTD. Un ASTD comporte un état initial, et peut avoir plusieurs états finaux. La structure des ASTD est décrite inductivement. Les différents types d'ASTD sont les suivants :

Le type Élémentaire : Le type élémentaire correspond aux places d'un automate. Il ne peut être présent que dans les ASTD de type automate.

Le type Automate : Le type automate reprend le formalisme des diagrammes d'états. Il est constitué d'un ensemble d'états, qui peuvent être des états élémentaires ou un autre ASTD. Un ensemble de transitions permet de parcourir les états. Les transitions peuvent être gardées. Elles peuvent aussi être franchies depuis ou vers un état d'un automate de niveau immédiatement inférieur à celui de l'ASTD courant. Enfin il existe des états historiques qui permettent de retrouver le dernier état visité dans un ASTD.

Le type Synchronisation paramétrée : Ce type d'ASTD permet de synchroniser deux ASTD. Il a comme paramètre un ensemble d'événements. Les événements ne faisant pas partie de cet ensemble sont exécutés en entrelacement. Les événements de l'ensemble doivent être exécutés en même temps dans les deux ASTD. Si l'ensemble de synchronisation est vide, c'est un entrelacement.
Le type Synchronisation quantifiée : La synchronisation quantifiée prend comme paramètre un ASTD, un ensemble de synchronisation, une variable de quantification et un ensemble de valeurs correspondant aux valeurs possibles pour cette variable. On exécute donc autant d'ASTD qu'il y a de valeurs dans l'ensemble. Comme pour la synchronisation paramétrée, les événements n'appartenant pas à l'ensemble de synchronisation sont exécutés en entrelacement, les autres sont exécutés en même temps pour l'ensemble des ASTD. Cette notation peut être étendue : on ajoute comme paramètre un prédicat ; seuls les ASTD dont les variables de quantification vérifient le prédicat se synchronisent.
Le type Séquence : L'opérateur de séquence prend en paramètre deux ASTD. Le deuxième ASTD ne peut commencer son exécution que lorsque le premier ASTD est dans un état final.
Le type Choix : Le type choix prend comme paramètre deux ASTD et en exécute un au choix de l'utilisateur.
Le type Choix quantifié : Le choix quantifié prend comme paramètre un ASTD, une variable de quantification et un ensemble de valeurs possibles pour la variable de quantification. Il permet d'exécuter l'ASTD en paramètre pour une des instanciations de la variable de quantification.
Le type Garde : Un ASTD de type garde ne peut être exécuté qui si sa garde est vérifiée.
Le type Appel d'ASTD : Le type appel exécute l'ASTD appelé, auquel on peut passer des paramètres.
Le type Fermeture de Kleene : Un ASTD de type fermeture de Kleene spécifie le fait qu'on puisse exécuter plusieurs fois un ASTD, y compris zéro fois.

1.1.2 La méthode B

La méthode B [Abr96] est une méthode de spécification formelle du logiciel. Elle permet de décrire formellement un système de sa spécification abstraite jusqu'à son implémentation. Son formalisme repose sur des notions mathématiques de la théorie des ensembles et de la logique du premier ordre. Elle se base également sur une relation qui permet de relier une machine abstraite à une machine concrète qui détaille la spécification de la machine abstraite. Cette relation, appelée raffinement, doit être prouvée par des obligations de preuves, afin que les propriétés des machines soient conservées. La méthode est outillée notamment grâce à l'atelier B [1] qui permet d'éditer la spécification, qui génère les obligations de preuves et qui permet de prouver ces obligations à l'aide de différents prouveurs.

Une spécification B contient plusieurs **MACHINES**. Une machine spécifie des opérations qui modifient les données. Elle contient également une partie statique qui contient les propriétés du système. Les propriétés sont écrites sous la forme d'invariant ou d'assertion. Les invariants sont des propriétés sur les variables (incluant leur typage). La méthode requiert de prouver que les invariants ne sont pas violés par l'exécution de chaque opération. Les assertions sont des lemmes facilitant la preuve de invariants et qui doivent être prouvées en utilisant les invariants.

Les opérations sont spécifiées par des substitutions sur les variables. Il existe une substitution particulière, la substitution "précondition". Cette substitution ne peut être utilisée qu'au niveau supérieur d'une opération. Elle est constituée d'un prédicat et d'une substitution. Le prédicat est appelé *précondition*, la substitution *postcondition*. Elle signifie que le comportement de la machine n'est connu que si la *précondition* est vraie. L'ensemble des substitutions utilisables dans la *postcondition* est défini dans [Abr96]. Nous ne détaillerons ici que les substitutions utilisées le plus fréquemment dans nos travaux :

— La substitution "affectation" qui affecte une valeur à une variable ;

— La substitution *"devient tel que"*, une substitution non déterministe qui spécifie la modification d'une variable de telle manière qu'elle vérifie une propriété écrite sous la forme d'un prédicat B ;

1. www.atelierb.eu

— La substitution "select", qui permet de définir un ensemble de conditions pour la modification des variables ;

— La substitution "any", qui donne le choix à l'utilisateur de la variable avec laquelle il veut exécuter la substitution

Pour garantir la préservation des propriétés, la méthode demande de prouver des obligations de preuves. Ces obligations sont définies dans le manuel de référence du langage B [Cle] et sont générées automatiquement par l'outil de développement (l'atelier B). Les obligations concernent deux éléments principaux : la préservation des invariants (et la preuve des assertions) et la preuve des raffinements. La preuve de préservation des invariants consiste à vérifier que chaque opération et l'initialisation ne contredisent pas les invariants. La preuve des raffinements vérifie que la spécification concrète d'une opération ne contredit pas sa spécification abstraite. Ces obligations peuvent être prouvées à l'aide des outils de l'atelier B, qui contient plusieurs prouveurs automatiques ainsi qu'un prouveur interactif manuel.

La génération des obligations de la preuve de préservation des invariants se fait à l'aide de la théorie des substitutions généralisées. Cette théorie définit une sémantique propre à chaque substitution. Dans le cas général, si on veut prouver la préservation d'un invariant I par une substitution S, il faut prouver le prédicat J défini par $J \triangleq I \Rightarrow [S]I$. Nous donnons dans la suite les règles de calcul pour les substitutions définies précédemment.

La substitution "Précondition". Elle se note **PRE** P **THEN** S **END**, où P est un prédicat et S une substitution. Elle signifie que si la précondition P est vraie, on applique la substitution S. Si la précondition n'est pas vraie, le comportement de la machine n'est pas connu, et l'état est modifié de manière quelconque. La sémantique de cette substitution est définie de la manière suivante :

$$[\textbf{PRE } P \textbf{ THEN } S \textbf{ END}]I \triangleq P \wedge [S]I$$

La substitution "Affectation" : Elle affecte une valeur à une variable. Elle s'écrit $x := e$. L'obligation de preuve pour la préservation des invariants est alors $[x := e]I$.

$[x := e]I$ signifie qu'on remplace dans le prédicat I l'ensemble des occurrences libres de x par e.

La substitution *"Devient tel que"* : Elle s'écrit $X : (P)$, où X est un ensemble de variables et P est un prédicat. L'ensemble des variables de X est alors remplacé par des valeurs qui satisfont le prédicat P. Le calcul de l'obligation de preuve associée à la préservation de l'invariant donne le résultat suivant : $\forall X.([X, x\$0 := Y, x]P \Rightarrow [X := Y]I)$, où x est une variable de X et $x\$0$ représente la valeur de la variable avant l'application de la substitution.

La substitution "Select" : Elle permet de spécifier un ensemble de cas. Elle s'écrit **SELECT** P_1 **THEN** S_1 **WHEN** P_2 **THEN** S_2 **END**. Elle signifie que si la condition P_1 est vraie, la substitution S_1 peut s'appliquer si la condition P_2 est vraie, la substitution S_2 peut s'appliquer. Si les deux conditions ne s'excluent pas mutuellement, la substitution est non déterministe et nous ne pouvons pas savoir quelle substitution s'applique. Si I est la conjonction de tous les invariants, l'obligation de preuve associée à la préservation des invariants est la suivante :

$$[\textbf{SELECT } P_1 \textbf{ THEN } S_1 \textbf{ WHEN } P_2 \textbf{ THEN } S_2]I \triangleq (P_1 \Rightarrow [S_1]I) \wedge (P_2 \Rightarrow [S_2]I)$$

Pour des raisons de clarté et de concision la substitution est expliquée avec deux prédicats et deux substitutions mais le nombre de cas n'est pas limité à deux. S'il y a plus de cas implémentés, le comportement est étendu pour l'ensemble de cas.

La substitution Any : Elle s'écrit **ANY** X **WHERE** P **THEN** S **END**. La substitution S est appliquée après avoir choisi des variables X qui vérifient P. L'obligation de preuve pour la conservation de l'invariant est la suivante : $\forall X.(P \Rightarrow [S]I)$.

Enfin, les obligations de preuves liées au raffinement permettent de prouver que le comportement d'une machine concrète est cohérent avec le comportement de la machine abstraite raffinée. Le langage B impose que chaque opération soit raffinée par une et une seule opération. Il y a autant d'opérations dans la machine concrète que dans la machine abstraite. Pour qu'une opération concrète raffine une opération

abstraite, il faut que la précondition de l'opération concrète soit plus faible que l'opération abstraite et que la postcondition de l'opération concrète soit plus forte. Le raffinement des ASTD est détaillé dans la section 1.2.

1.1.3 La méthode Event-B

La méthode Event-B est une méthode dérivée de la méthode B, mais utilisée pour une spécification événementielle. Elle permet de spécifier un système d'événements. Une partie statique, appelée contexte, définit les constantes et des propriétés sur ces constantes. Une partie dynamique, appelée machine, définit les variables, les invariants et variants sur ces variables, et les événements qui peuvent les modifier. Elle utilise un langage et des clauses très proches de la méthode B. La différence réside essentiellement dans la méthode de spécification.

Comme une machine B, une machine Event-B peut être raffinée. La méthode du raffinement en Event-B suit le "paradigme du parachute". Selon ce paradigme, le système est spécifié comme si une personne le décrivait pendant une descente en parachute. Le niveau abstrait du système décrit un système très simple et les détails sont ajoutés au fur et à mesure des raffinements.

La machine Event-B contient des événements, dont un événement d'initialisation, qui remplacent les opérations de la machine B. Les valeurs d'initialisation doivent établir les invariants. Un événement peut être paramétré, à l'aide d'une clause **ANY**. Une clause **WHEN** permet d'associer une garde à l'événement. Un événement ne peut être déclenché que si toutes ses gardes sont vraies. Les actions sont décrites dans une clause **THEN**. Contrairement au B classique, on ne peut écrire pas écrire de substitutions imbriquées, telles que les **SELECT**. On ne peut écrire que des affectations et des substitutions *"Devient tel que"*.

Le calcul des obligations de preuve n'utilise pas la théorie des substitutions généralisées. Les clauses de la machine (invariants, gardes, substitutions, etc...) sont exprimées par des prédicats liant les différents éléments de la machine (variables, constantes, paramètres, etc...). Les obligations de preuves demandent de déduire certains de ces prédicats à partir d'autres, suivant les règles définies dans [Abr07]. La preuve des obligations de preuves permet de garantir, comme pour le B classique, la

préservation des invariants et les raffinements. Les obligations de preuve de l'Event-B demandent à prouver en plus la faisabilité de la substitution. Pour une substitution de type *"Devient tel que"*, cette obligation consiste à montrer que pour chaque variable modifiée, il existe une valeur qui satisfait le prédicat.

1.2 Le raffinement

Les systèmes ferroviaires sont des systèmes complexes. Pour faciliter la spécification, nous souhaitons pouvoir modéliser les systèmes en plusieurs étapes. Les premières spécifications sont dites abstraites et ne modélisent pas tous les détails du fonctionnement du système. Ces détails de fonctionnement sont spécifiés dans des modélisations ultérieures. Une relation de raffinement permet de garantir la cohérence entre deux niveaux de modélisation. Cette relation est définie par un ensemble de propriétés qui doivent être prouvées pour prouver le raffinement d'une spécification par une autre.

1.2.1 Les spécifications basées sur les événements

Dans les algèbres de processus, comme CSP [Hoa78], les observations sur le système sont faites à l'aide de séquences d'événements admis ou non par la spécification. On observe plusieurs ensembles de séquences d'événements, telles que les traces, les séquences menant à un deadlock, les séquences divergentes, etc... Dans ce type de spécification, un raffinement est défini par des relations d'inclusion ou d'égalité entre ces ensembles. On parle de sémantique dénotationnelle [RHB97]

Ces ensembles définis ci-dessus correspondent à une modélisation des comportements admissibles du système. L'idée général du raffinement de système décrit par un paradigme basé sur les événements est la suivante. Il ne faut pas qu'une machine concrète C qui raffine une machine abstraite A admette des comportements que n'admet pas A. Il faut donc que les ensembles décrivant le comportement de la machine C soient au moins admis par la machine A. On peut par exemple comme le font

Schneider et al. [STW14] définir le raffinement en CSP de la manière suivante :

$$\begin{aligned} \mathsf{traces}(C) &\subseteq \mathsf{traces}(A) \\ \mathsf{divergences}(C) &\subseteq \mathsf{divergences}(A) \\ \mathsf{inifinites}(C) &\subseteq \mathsf{infinites}(A) \end{aligned}$$

Ces ensembles sont des ensembles de séquences d'événements. Les traces sont les séquences finies d'événements admissibles par une machine. L'ensemble $\mathsf{infinites}(A)$ (resp. C) correspond à l'ensemble des séquences infinies d'événements exécutables par la machine A (resp. C). Enfin les divergences sont les séquences d'événements menant à un état chaotique. Dans le cas de ce raffinement, le comportement de la spécification abstraite est restreint par la machine concrète.

Les ASTD sont aussi un système de spécification dont le paradigme est basé sur les états. Une définition du raffinement pour les ASTD est donnée dans [FGLM14]. Elle nécessite de prouver les assertions suivantes :

$$\begin{aligned} \alpha(A) - \alpha(C) &\neq \varnothing \\ \mathsf{traces}(A^{\setminus \bar{C}}) &= \mathsf{traces}(C^{\setminus \bar{A}}) \\ \mathsf{deadlock}(A^{\setminus \bar{C}}) &= \mathsf{deadlock}(C^{\setminus \bar{A}}) \\ \tau\text{-}\mathsf{enabled}(A^{\setminus \bar{C}}) \subseteq \tau\text{-}\mathsf{enabled}(C^{\setminus \bar{A}}) &\neq \varnothing \end{aligned}$$

Pour un ASTD A, $\alpha(A)$ est l'ensemble des étiquettes dénotant les transitions de l'ASTD A. $A^{\setminus \bar{C}}$ est l'ASTD A dans lequel toutes les transitions présentes dans l'ASTD A et pas dans l'ASTD C sont remplacées par des transitions silencieuses (c'est-à-dire $A^{\setminus \bar{C}} = A \setminus (\alpha(A) - \alpha(C))$ où $\setminus$ est l'opérateur *hide* de CSP). Les transitions silencieuses sont notées τ. L'ensemble $\tau - \mathsf{enabled}$ est l'ensemble des séquences d'événements après lesquels il est possible d'exécuter une transition silencieuse. L'ensemble $\mathsf{deadlock}$ est l'ensemble des séquences d'événements après lesquels on ne peut plus effectuer d'événements.

D'après cette définition du raffinement pour les ASTD, un ASTD concret doit avoir le même comportement qu'un ASTD abstrait, si on ne compare que les opérations communes. C'est une définition plus forte que le raffinement défini en CSP.

1.2.2 Les spécifications basées sur les états

Les méthodes de spécification basées sur la représentation des états des variables définissent des types de donnée abstraits (*Abstract Data Types - ADT*). Un *ADT* [DB01] est défini par une initialisation et un ensemble d'opérations. L'initialisation permet de définir l'état initial des variables et les opérations peuvent être vues comme des relations entre des états des variables. Dans ce système de représentation, les raffinements sont exprimées à l'aide d'une relation de simulation. La relation de simulation lie les états de la machine concrète à ceux de la machine abstraite. Si on effectue une même opération dans des états liés par la relation de simulation, les nouveaux états obtenus doivent être liés par la relation de simulation. C'est le cas pour la définition du raffinement en B et Event-B : le raffinement n'est pas défini par une relation sur les comportements globaux du système, mais par une relation liant la transformation abstraite des états à la transformation concrète.

Le Raffinement de la méthode B

Nous l'avons vu dans la section 1.1.2, dans une machine B sont définies des variables. Des opérations permettent de modifier l'état de ces variables tout en ne violant pas les invariants. Soit une machine abstraite A comprenant des variables $v_{a_1}, v_{a_2}, ..., v_{a_n}$, un invariant $I_a(v_{a_1}, v_{a_2}, ..., v_{a_n})$ (on considère un invariant global qui est la conjonction de tous les invariants), et des opérations $op_1, op_2, ..., op_k$. Soit une machine concrète C raffinant la machine A.

Le raffinement défini par la méthode B ne permet pas de modifier le nombre d'opérations. La machine C contient donc les même opérations $op_1, op_2, ..., op_k$ dont on peut modifier la substitution. La machine C définit de nouvelles variables $v_{c_1}, v_{c_2}, ..., v_{c_m}$ et un invariant $I_c(v_{a_1}, v_{a_2}, ..., v_{a_n}, v_{c_1}, v_{c_2}, ..., v_{c_m})$ qui lie les variables concrètes aux variables abstraites. Les invariants qui permettent de lier l'état d'une variable concrète à l'état d'une variable abstraite sont appelés invariants de collage. Ce sont ces invariants qui définissent la relation de simulation.

Une machine B raffine une autre machine B si chaque opération concrète raffine l'opération abstraite correspondante. Informellement, une opération concrète op_{i_c} raffine une opération abstraite op_{i_a} si on peut remplacer l'opération op_{i_a} par l'opération

op_{i_c} sans qu'un observateur ne s'en rende compte. Supposons que les deux opérations sont des substitutions "précondition" de la forme **PRE** Pr_a **THEN** Su_a **END** pour op_{i_a} et **PRE** Pr_c **THEN** Su_c **END** pour op_{i_c} (on peut toujours se ramener à une substitution précondition en prenant $Pr = TRUE$). Pour que l'opération op_{i_c} raffine l'opération op_{i_a} il faut que la précondition Pr_c soit moins restrictive que la précondition Pr_a (*i.e.* $Pr_a \Rightarrow Pr_c$) et que la substitution concrète ne contredise pas la substitution abstraite. Si on prend un prédicat R, cela signifie, dans le calcul des plus faibles préconditions que la plus faible précondition qui permet de satisfaire le prédicat R après l'application de la substitution abstraite Su_a implique la plus faible précondition qui permet de satisfaire R après l'application de la précondition concrète Su_c (*i.e.* $[Su_a]R \Rightarrow [Su_c]R$). Dans la pratique, cela signifie qu'on réduit le non déterminisme de l'opération, c'est-à-dire que les comportements prévus par la machine concrète sont des comportements prévus par la machine abstraite.

Le Raffinement de la méthode Event-B

Contrairement à la méthode B, la méthode Event-B permet l'ajout d'événements dans une machine concrète. On peut donc avoir plusieurs événements qui en raffinent un et des événements qui ne raffinent aucun événement (Ces événements raffinent la substitution qui ne fait rien : **skip**). Il existe alors une fonction qui lie les événements des deux machines. Cette fonction part de l'ensemble des événements concrets vers l'ensemble des événements abstraits. C'est une fonction partielle et surjective, ce qui signifie que chaque événement concret raffine au plus un événement abstrait et que tout événement abstrait est raffiné par au moins un événement concret.

Soit une machine Event-B M. Cette machine définit un ensemble de variable v, des invariants sur les variables $I(v)$, et des événements qui peuvent modifier l'état des variables. Un événement ev_i s'écrit $ev_i =$ **when** $G(v)$ **then** $BA(v, v')$ **end**. $G(v)$ est appelé la garde de l'événement : l'événement ne peut être déclenché que quand elle est vraie. $BA(v, v')$ est un prédicat avant/après : il lie l'état des variables avant l'exécution de l'événement à l'état des variables après l'exécution de l'événement.

Soit une machine M' raffinant M. M' contient de nouvelles variables d'état w, des invariants liant ces variables aux variables abstraites $J(v, w)$ et de nouveaux événements ev'_j. Chaque nouvel événement ev'_j est de la forme $ev'_j =$ **when** $H(v)$ **then**

$BA(v, v')$ **end**. Pour prouver le raffinement de l'événement ev_i par le raffinement ev'_j, il faut prouver :

— que la garde est renforcée : si l'événement est nouveau, c'est évident, sinon on montre que $H(w) \Rightarrow G(v)$

— que le nouvel événement simule bien l'ancien : on montre donc que $I(v) \wedge J(v, w) \wedge H(w) \wedge BA'(w, w') \Rightarrow \exists v'.(BA(v, v') \wedge J(v', w'))$ (on note que si l'événement est nouveau, il raffine **skip** et donc que $BA(v, v') \equiv v = v'$).

1.2.3 Discussion et positionnement

Pour exprimer la relation de raffinement dans le cadre d'une spécification basée sur les événements, les observations sont faites sur des comportements globaux du système (ensemble de séquences d'événements acceptables, etc...). La définition du raffinement impose donc la vérification de propriétés sur ces comportements globaux du système. Lorsque les spécifications sont basées sur les états, les observations sont faites sur des opérations modifiant les états du système. Le raffinement impose de vérifier que les modifications qu'effectue une opération concrète sur l'état des variables n'introduit pas d'incohérence avec les modifications qu'effectue l'opération abstraite.

Mais la sémantique utilisée pour décrire le comportement ne change pas le sens donné au raffinement. Schneider et al. [STW14] ont montré par exemple qu'il est possible d'exprimer le raffinement Event-B en matière de traces, divergences et traces infinies. Le choix d'une méthode de spécification et de raffinement par rapport à une autre se fait surtout en fonction du type de comportement que nous souhaitons représenter.

Certaines différences techniques cependant permettent de faire des choix. Le raffinement des ASTD permet l'ajout d'opérations pendant le raffinement (et l'oblige même d'après [FGLM14]). Nous souhaitons combiner cette méthode à une méthode permettant une expression de la modification des états des variables du système. La méthode B classique ne permettant pas l'ajout d'opération pendant le raffinement, nous privilégierons la méthode Event-B.

Enfin, les ASTD sont un langage de spécification basé sur les états. Les obligations de preuve du raffinement sont exprimées en utilisant la sémantique dénotationnelle,

c'est-à-dire à l'aide de propriétés sur les observations. Pour garantir la sécurité des systèmes ferroviaires, le comportement des entités du système est souvent contraint par un contrôleur. La spécification d'un système ferroviaire pour le cas d'étude [Fay14] a montré que la définition du raffinement donnée dans [FGLM14] est trop restrictive pour permettre de contraindre le fonctionnement d'un système par l'ajout d'un contrôleur.

1.3 Les combinaisons de méthodes formelles

1.3.1 iUML-B

iUML-B est issu de la notation UML-B [SB06] qui combine l'UML et la méthode B. La version actuellement développée utilise la méthode Event-B.

Description de la méthode

iUML-B est la combinaison de la notation UML et de la méthode Event-B. Il est intégrée à Rodin, l'outil de spécification en Event-B. Lors de la spécification d'une machine ou d'un contexte Event-B, il est possible d'ajouter une ou plusieurs machines à états ainsi qu'un ou plusieurs diagrammes de classes. Ces spécifications en UML sont ensuite traduites en Event-B et permettent de contraindre l'exécution de la machine. La partie de la spécification Event-B résultant directement de la traduction ne peut être modifiée, mais d'autres spécifications peuvent être ajoutées.

Les diagrammes de classes permettent de spécifier les structures de données de la même manière que dans la programmation orientée objet. Il est possible de définir des classes et de leur associer des attributs, des méthodes et des contraintes. On peut également hiérarchiser les types et définir des associations entre des classes. Dans la traduction en Event-B, ces associations sont modélisées par des relations sur les ensembles et les méthodes de chaque classe sont des événements qui permettent de modifier ces ensembles.

Les machines à états s'inspirent des diagrammes d'états de Harel [Har87]. Ils permettent de définir un ensemble d'états et de les relier par des transitions. Dans la traduction en Event-B, les transitions sont traduites par des événements. Les états

peuvent être codés de deux manières différentes. Dans la première, ils sont définis comme des constantes, et une variable d'état prend la valeur de l'état courant. Dans ce cas, la garde de la transition vérifie que le système est dans l'état précédant la transition et la postcondition met à jour l'état. Dans la seconde, les états sont définis comme des variables booléennes, qui prennent la valeur vraie s'ils sont l'état courant de l'automate. Les machines à états peuvent être hiérarchiques, c'est-à-dire que l'état de chaque machine à états peut à son tour être une machine à états. Ces machines à états peuvent également avoir plusieurs instances, ce qui permet de spécifier un entrelacement quantifié, mais uniquement au plus haut niveau.

Il n'existe pas de documentation sur les règles de traduction implémentées dans l'outil UML-B. Il est donc difficile de connaître avec précision la sémantique formelle formelle d'une spécification UML. Il faut utiliser l'outil pour voir le code Event-B généré.

Positionnement

La méthode iUML-B est une méthode de spécification très proche des ASTD. Elle a l'avantage d'être bien outillée et intégrée à Rodin. Cependant, le manque de documentation sur la sémantique formelle ne permet pas de commenter avec précision les différences exactes avec les ASTD.

Certains opérateurs des ASTD ne sont néanmoins pas inclus dans le langage iUML-B. Il est ainsi impossible de spécifier une synchronisation quantifiée, et l'entrelacement quantifié n'est utilisable qu'au plus haut niveau de spécification.

1.3.2 CSB||B

Description de la Méthode

La méthode de spécification CSP||B [ST02] utilise les paradigmes de la méthode B et ceux de l'algèbre de processus CSP. Une machine B permet de décrire les modifications apportées par le système aux variables d'état. Le processus CSP, appelé *contrôleur d'exécution*, est utilisé pour contraindre l'exécution des opérations de la

machine B. Chaque machine B est associée à un processus CSP. Un système peut contenir plusieurs paires de spécifications machine B/Processus CSP.

La spécification en B est modélisée dans une machine B standard, sans restriction sur le langage. Cette partie permet de vérifier des propriétés d'invariance sur les états des variables, comme pour toute spécification B.

Le contrôleur CSP utilise un sous ensemble du langage CSP. Il permet d'utiliser l'opérateur préfixe, les opérateurs de choix externes ou internes, les canaux de communication et les gardes et assertions des événements. Les opérateurs de synchronisation et d'entrelacement ne peuvent être utilisés dans les contrôleurs. Cependant, si on considère une spécification en CSP||B comme un ensemble de couple $(P_i, M_i)_{i\in[1..n]}$, où P_i est un processus CSP et M_i une machine B, le comportement global du système est modélisé par la composition parallèle de l'ensemble de ces systèmes (c'est-à-dire $P_1||...||P_n$).

La difficulté de cette combinaison de spécification est de s'assurer de la cohérence du modèle. La combinaison de CSP et de B peut introduire des blocages de la spécification. Pour éviter ces blocages, Schneider et Treharne [ST02] montrent qu'il faut prouver (1) pour chaque couple de spécification (P_i, M_i) où P_i est un processus CSP et M_i la machine B associée, que la spécification $M_i||P_i$ est sans divergence et (2) que la spécification $||_{i\in[1..n]}P_i$ est sans blocage.

Enfin, cette combinaison de méthode ne définit pas de raffinement. On peut cependant raffiner la machine de spécification B en utilisant le raffinement de la méthode B.

Positionnement

La composition CSP||B est une combinaison de méthodes formelles très proche de celle que nous souhaitons définir pour la spécification des systèmes ferroviaires. L'agencement des opérations du système est décrit à l'aide d'un paradigme basé sur les événements. La spécification des modifications effectuées sur les variables d'état est décrite à l'aide d'un paradigme basé sur les états.

Les ASTD apportent une spécification graphique qui permet une bonne lisibilité des spécifications tant pour des spécialistes que pour des non spécialistes. De plus, la spécification en ASTD ne restreint pas l'utilisation des opérateurs des algèbres de

processus. Enfin, la méthode ne définit pas spécifiquement de raffinement, alors que nous souhaitons pouvoir utiliser le raffinement pour spécifier nos systèmes.

1.3.3 CSP2B

Description de la Méthode

La méthode de spécification CSP2B [But00] utilise également les paradigmes de la méthode B et de l'algèbre de processus CSP. Une machine B spécifie les modifications apportées aux variables d'état et un processus CSP contraint l'exécution de la machine. Les outils de cette méthode permettent de traduire la spécification CSP en une machine B standard. Celle-ci est reliée à la machine spécifiant la modification des variables d'état grâce à la clause d'inclusion du langage B.

La spécification en B de la modification des variables d'état se fait dans une machine B, sans aucune restriction sur le langage.

La spécification en CSP est effectuée dans une machine B spéciale. Cette machine définit un alphabet et une description du processus à l'aide d'un sous-ensemble d'opérateurs CSP. Les opérateurs CSP utilisables sont le préfixe, le choix, la synchronisation, l'entrelacement et le processus *SKIP*, processus qui ne fait rien. Les processus peuvent également utiliser l'opérateur de condition **IF** c **THEN** P_1 **ELSE** P_2. Les opérateurs ne sont pas tous utilisables sans restriction : la synchronisation ne peut être utilisée que pour les expressions les plus à l'extérieur de la spécification. L'entrelacement ne peut concerner que plusieurs instances d'un même processus.

L'avantage de cette méthode de combinaison de CSP et B est que la spécification CSP est traduite en B. L'ordonnancement de la machine CSP est codée par des variables d'état en B. On peut donc vérifier des propriétés grâce aux clauses **ASSERTIONS** et **INVARIANTS** de la méthode B. Cette inclusion permet également de vérifier la cohérence de la machine définie. En revanche, les propriétés vérifiables sont des propriétés d'invariance sur les états. Les propriétés sur l'ordonnancement sont plus difficiles et parfois impossible à vérifier.

De plus la réunification dans le langage B des deux spécifications permet d'utiliser les méthodes de raffinement de la méthode B. La spécification des variables d'état est raffinée en utilisant la méthode B. La spécification CSP2B est raffinée à l'aide d'un

invariant d'abstraction qui lie les variables d'état de la machine de contrôle abstraite aux variables d'état de contrôle de la machine de contrôle concrète.

Positionnement

La méthode CSP2B combine les algèbres de processus et la méthode B et est donc très proche de la méthode que nous souhaitons utiliser. Elle n'utilise cependant pas de méthode de spécification graphique, que nous estimons intéressante pour la spécification de systèmes qui doivent être compris par des non spécialistes. De plus, le raffinement de la partie CSP est le raffinement défini pour la méthode B. Sa définition est donc beaucoup plus restrictive que certains raffinements que nous souhaiterions pouvoir utiliser dans notre méthode.

1.3.4 Circus

Le langage de spécification Circus [WC02] combine la notation CSP avec le langage Z. Avant de décrire Circus, nous présentons brièvement le langage Z.

Le langage Z

Le langage Z [Spi89] est basé sur la logique du premier ordre et la théorie des ensembles. Le schéma est la base de la notation de Z. Dans la partie statique, ils permettent de décrire les états du système. Dans la partie dynamique, ils permettent de spécifier les changements apportés aux états du système.

Un schéma statique, qui décrit les états du système, est composé de deux parties. La première partie contient la déclaration des variables et leur typage. La deuxième partie contient les propriétés d'invariance du système. Chaque schéma porte un nom.

Un schéma dynamique permet de décrire les modifications effectuées sur les états d'un système. Un schéma dynamique est également divisé en deux parties. La première partie permet de nommer les schémas qui seront modifiés et de définir les paramètres d'entrée et de sortie. Les schémas modifiés sont identifiés à l'aide de l'opérateur Δ. Les paramètres d'entrée sont suivis d'un "?", les paramètres de sortie d'un "!". La deuxième partie d'un schéma dynamique définit les propriétés que vérifient les variables au cours du changement. Ces propriétés sont définies en fonction des paramètres, des valeurs

des variables d'état avant l'exécution du schéma, et des valeurs des variables d'état après l'exécution du schéma. L'état d'une variable *var* après l'exécution est noté var'.

Description de la Méthode

Dans les notations CSP||B (1.3.2) et CSP2B (1.3.3), la spécification est en deux parties. Une partie en CSP spécifie l'agencement des processus et une partie en B spécifie les modifications apportées aux variables d'état. Dans la notation Circus, le système est modélisé par une unique spécification.

La notation Circus est constituée d'un ensemble de paragraphes. Ces paragraphes permettent de déclarer des canaux d'échanges ou des processus. Les canaux d'échanges ont la même fonction qu'en CSP et permettent d'échanger des variables entre plusieurs processus et de les synchroniser.

Un processus est séparé en deux parties : les déclarations et les actions. Dans la partie déclaration on trouve à la fois des schémas Z et des opérateurs de l'algèbre de processus CSP. Les schémas Z définissent les modifications apportées aux données. Les opérateurs CSP utilisables sont les opérateurs de choix interne et externe, les entrelacements, les synchronisations et les séquences. On peut également utiliser l'opérateur μ du langage Z. Cet opérateur permet de définir récursivement les actions. Il s'écrit de la forme $\mu N \bullet A(N)$, où N est un identifiant et A est une expression d'action qui dépend de N. Il signifie qu'on cherche le plus petit point fixe tel que $N = A(N)$. Dans la partie action, on trouve la définition des actions effectuées par le processus. Ces actions sont décrites à l'aide des opérateurs CSP et de l'opérateur μ.

La sémantique du raffinement en Circus (définie dans [CSW03]) est basée sur la Théorie Unifiée de la Programmation (*Unifying Theories of Programming - UTP* [HJ98]). La preuve du raffinement d'une spécification Circus se fait en exhibant une relation de simulation. Des méthodes de raffinement spécifiques sont définies par Cavalcanti et al. [CSW03] grâce notamment au raffinement de données de Z.

Positionnement

La méthode de spécification Circus combine un paradigme de spécification basé sur les événements (CSP) avec un paradigme de spécification basé sur les états (le lan-

gage Z). La méthode crée un nouveau langage contenant les deux méthodes, contrairement aux méthodes décrites précédemment. Elle permet l'utilisation de techniques de raffinement basées sur la théorie de la programmation unifiée. Il n'y a donc pas de restrictions importantes sur les raffinements utilisables dans la méthode.

Cependant, nous croyons que les spécifications graphiques, utilisées dans les spécifications des ASTD, permettent une meilleure lisibilité des spécifications. D'autant plus que les spécifications de systèmes ferroviaires doivent être relues et comprises par des ingénieurs non informaticiens dont le niveau d'expertise n'est pas suffisant pour saisir toutes les subtilités d'un langage mathématique complexe.

Enfin, pour pouvoir être utilisée dans le cadre d'un projet industriel, une méthode doit être correctement outillée. Un outil permet la traduction des modèles en langage Java [FC06], mais le Java n'est pas un bon candidat pour les spécifications formelles, notamment parce qu'il n'existe pas de machine virtuelle certifiée. De même, seul le prouveur ProofPower est disponible [Art], mais il n'est pas intégré à un outil de spécification. Il manque également des outils pour animer et vérifier les spécifications.

1.4 Utilisation des méthodes formelles pour la spécification de systèmes ferroviaires

L'utilisation des méthodes formelles pour la spécification de systèmes ferroviaires a fait l'objet de nombreux travaux. Ces travaux peuvent être classés en deux catégories. Dans la première catégorie, le but est de tester l'efficacité d'une méthode dans le cadre de la modélisation de systèmes ferroviaires. Dans la deuxième catégorie, le but des projets est de construire un système ou une spécification, en utilisant des méthodes formelles.

1.4.1 Test de méthodes

Dans l'Event-B book [Abr07], Abrial modélise un système ferroviaire en utilisant la méthode Event-B. Cette modélisation spécifie la partie *interlocking* d'un système ferroviaire. La voie sur laquelle circulent les trains est représentée par un ensemble de tronçons appelés CDV (Circuit De Voie), liés par une relation de proximité. Un

CDV “normal” a deux voisins, un aiguillage trois et un croisement quatre. Lorsqu'un train souhaite aller d'un point à un autre, il doit emprunter un ensemble de tronçons, qui constituent son itinéraire. Les tronçons de son itinéraire sont verrouillés et aucun autre train ne peut les emprunter. Pour verrouiller un itinéraire, il faut vérifier que les tronçons qui le constituent sont tous libres.

Positionnement : Cette spécification est effectuée grâce à la méthode Event-B. Elle est donc modélisée en plusieurs étapes, en utilisant le raffinement de la méthode Event-B. Elle permet de prouver des propriétés de non-collision. Mais elle ne permet pas de modéliser la présence de plusieurs trains. De plus elle se focalise sur les zones avec de nombreux aiguillages, mais ne permet pas de modéliser le fonctionnement complet d'un système.

Dans sa thèse [Sil12], Renato Silva modélise également un système ferroviaire en utilisant la méthode Event-B. Une nouvelle fois, la voie est divisée en tronçons, comprenant des aiguillages. La spécification est décomposée en plusieurs sous-systèmes (le train, la voie, et la machine de communication). Seul le sous-système du train est développé. Le raffinement de ce sous-système permet de spécifier le système d'ouverture des portes.

Positionnement : La spécification de Renato Silva modélise des aspects auxquels nous ne nous intéressons pas, et qui n'ont pas nécessairement besoin d'être modélisés par des ASTD. Mais sa modélisation permet de s'intéresser à la décomposition d'un système en plusieurs sous-systèmes, ce dont nous avons eu besoin pour la spécification de notre cas d'étude [Fay14].

1.4.2 Modélisation de système

La méthode B a souvent été utilisée pour la spécification de systèmes ferroviaires. Des industriels ont utilisé la méthode B pour spécifier des systèmes qui fonctionnent dans plusieurs pays. Le projet Météor [BBFM99] est un exemple de réussite de l'utilisation de la méthode B dans le domaine ferroviaire. Ce projet a permis de spécifier le ligne 14 du métro de Paris. Pour la réalisation de ce projet, une spécification système est écrite en langage naturel. À partir de cette spécification système, une spécification logicielle écrite elle aussi dans un langage naturel est dérivée. C'est cette

spécification qui est traduite dans une spécification formelle en B. Cette spécification est ensuite formalisée en B puis raffinée jusqu'à obtenir une implémentation en ADA (certains raffinements sont automatisés grâce à l'outil propriétaire Edith B de Siemens [Dol06]). L'expérience a depuis été répétée sur plusieurs projets de systèmes ferroviaires en France (Paris Ligne 1) et dans le monde (New-York, Sao Paulo, Etc...)
Positionnement : Ces projets ont permis de montrer que le langage B peut être utilisé dans le cadre de projets réels allant de la spécification à l'implémentation. Il est donc possible d'utiliser cette méthode pour des projets ambitieux. Mais dans cet exemple, les méthodes formelles ne sont pas présentes au cours de toute la spécification. Dans nos travaux, nous modéliserons le système à un niveau plus abstrait, mais nous souhaitons réduire au maximum la part des spécifications écrites en langage non-formel.

Le CBTC *Communication Based Train Control System* est un standard définissant un système de contrôle automatique de métro. Il est défini par des normes (IEEE 1474.1 [IEE] et IEC 62290 [IEC14]). Dans [FSMM14], Ferrari et al. partent de ces normes et des documents commerciaux des principaux industriels pour définir dans un langage semi-formel les exigences d'un CBTC. Le document définit les principaux éléments d'un CBTC et spécifie les liens qui les lient. Il définit également un ensemble de scénarios et les exigences de réponses attendues. Ces scénarios sont écrits dans un langage naturel contraint.
Positionnement : Lors de la spécification d'un système, il n'est pas possible de couvrir l'ensemble du processus avec des méthodes formelles. Ces travaux utilisent des méthodes de spécifications semi-formelles pour couvrir la partie la plus abstraite de la spécification qui est souvent écrite en langage naturel (comme c'est le cas par exemple dans le projet Météor). Les ASTD sont une méthode de spécification plus concrète mais formelle qui pourrait faire le lien entre ce type de spécifications abstraites et semi-formelles et des spécification logicielles encore plus concrètes.

Enfin, Haxthausen et al. [HPK11] définissent une méthode de spécification de systèmes ferroviaires. Cette spécification se fait à l'aide d'un langage spécifique au domaine. La spécification dans ce langage est traduite en *SystemC* qui est ensuite compilé. Chaque étape de spécification et de traduction est accompagnée de vérifications qui valident la syntaxe et la sémantique des spécifications. Dans le cas de la

traduction et de la compilation, elles vérifient également que la traduction est bien conforme à la spécification. Le développement du langage spécifique au domaine est fait en utilisant le dogme *TripTych* de Bjørner [Bjø06]. Un modèle du domaine a été spécifié avant tout développement en utilisant le langage formel RAISE [GH08]. Le but de cette spécification est d'arriver à une implémentation exécutable. La modélisation est donc une modélisation concrète du système. Le langage spécifique utilisé permet par exemple de modéliser concrètement la voie, c'est-à-dire les tronçons qui la découpent et les relations de proximité entre les tronçons.

Positionnement : Haxthausen et al. définissent une méthode de développement de systèmes ferroviaires qui permet en partant d'une spécification du modèle d'obtenir un exécutable. Mais le modèle et le langage spécifique utilisés sont des modélisations concrètes, qui spécifient directement un plan de voie et les règles qui s'appliquent pour contrôler ce plan de voie. Notre méthode de spécification permet de modéliser le système à un niveau plus abstrait, ce qui permet de spécifier le système dans son ensemble, et non une partie localisée du système.

- Utilisation de méthodes formelles dans le ferroviaires :

Résumé des différents projets ferroviaire dans lesquels des méthodes formelles ont été utilisées, à la fois dans la recherche et dans l'industrie

- Combinaison de méthodes formelles :

Quelles sont les méthodes combinant des méthodes formelles ? Comment sont combinées les méthodes formelles ? Comment la cohérence du système est garantie ?

- Raffinement :

Quelles sont les différents types de raffinement ? Y a-t-il des raffinements équivalents ?

Chapitre 2

Définition formelle d'une méthode de spécification couplant ASTD et B

Dans ce chapitre, nous présentons la méthode utilisée pour la combinaison des ASTD et des langages B et Event-B. La section 2.1 présente la méthode et les différents langages utilisés. Les sections 2.2 et 2.3 définissent formellement respectivement la méthode et le raffinement. Enfin, la section 2.4 présente les outils nécessaires à la mise en œuvre de la méthode et leur état de développement actuel.

2.1 Présentation de la méthode

La méthode de spécification combine les langages B et Event-B et la notation graphique ASTD. La méthode est résumée à la figure 2.1. Les ASTD permettent de modéliser l'ordonnancement des actions grâce à une notation graphique et aux opérateurs des algèbres de processus (rectangles de gauche sur la figure 2.1). Cependant, cette modélisation ne permet pas de spécifier les modifications apportées par chaque transition au modèle de données. Ces modifications sont spécifiées en utilisant le langage Event-B (rectangles de droite sur la figure 2.1). Enfin, pour vérifier la cohérence de la spécification, les deux spécifications sont traduites en langage B (rectangles centraux sur la figure 2.1). Le choix des différents langages sera discuté dans la section 2.2. Le raffinement combiné des deux méthodes est décrit dans la section 2.3.

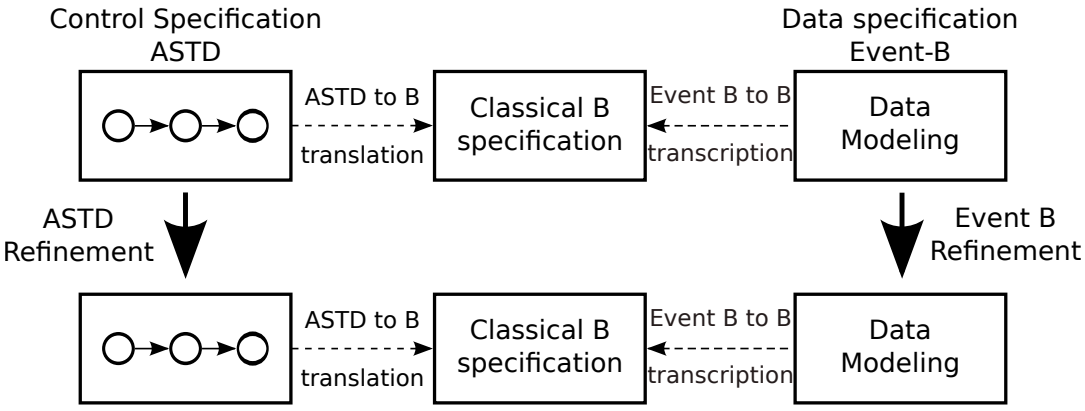


figure 2.1 – Résumé de la méthode

2.1.1 La spécification ASTD

La spécification ASTD permet de modéliser l'agencement des actions. Puisque les ASTD sont basés sur les diagrammes de transition d'états, ils permettent d'écrire une spécification graphique qui peut être relue, comprise et validée facilement. De plus, les opérateurs des algèbres de processus permettent non seulement d'ajouter de la lisibilité en factorisant certaines spécifications (fermeture de Kleene, opérateur de séquence), mais également d'ajouter de la puissance d'expression en étendant un comportement à un ensemble de processus (synchronisation et choix quantifiés). Enfin, les ASTD sont définis grâce à une sémantique formelle (définie dans [FGLF08]), qui permet d'éviter les ambiguïtés.

2.1.2 La spécification Event-B

La spécification ASTD ne permet de spécifier qu'un agencement d'actions. Pour que la méthode puisse décrire le fonctionnement d'un système, il est nécessaire de pouvoir décrire l'effet de ces actions sur les données du système. Cet effet est décrit par le langage Event-B, qui est un langage de spécification basé sur les états. La principale raison du choix d'Event-B, par rapport au langage B classique, est de faciliter le raffinement des spécifications. En effet, le raffinement Event-B permet l'ajout d'événements, comme le raffinement ASTD permet l'ajout de transition.

La machine Event-B associée à un ASTD contient un ensemble de variables d'état et contient un événement pour chaque étiquette de transition de la spécification ASTD. L'événement associé à une étiquette de transition permet de décrire l'effet de cette transition sur les variables d'état du système. C'est généralement dans cette partie de la spécification que seront traduites les propriétés que doit vérifier le système, généralement au moyen d'invariants. Ces invariants ne permettent pas d'écrire tous les types de propriétés, même si certaines (comme certaines propriétés utilisant des opérateurs de la logique temporelle) peuvent être encodées au moyen de théorèmes du langage Event-B.

2.1.3 La spécification B classique

La spécification ASTD spécifie l'agencement des actions, c'est-à-dire qu'elle spécifie le moment où doivent être exécutées les transitions. La spécification Event-B spécifie les changements apportés aux données du système lors de chaque transition. Cependant, pour permettre la préservation des propriétés de sûreté écrites sous la forme d'invariants, les événements Event-B ont une garde qui doit être vraie au moment d'exécuter l'événement. La spécification en B classique permet de vérifier que lorsqu'une action peut être exécutée selon la spécification ASTD, la garde correspondante en Event-B est vraie.

Pour vérifier cela, les deux spécifications sont traduites en B classique. L'idée de la traduction des ASTD est la suivante : les états des ASTD sont codés par des variables d'état en B. Une opération B est créée pour chaque étiquette de la spécification ASTD. La précondition de l'opération vérifie que les variables d'état correspondent à l'état précédant la transition. La postcondition met à jour les variables pour qu'elles correspondent à l'état qui suit la transition.

De même, la spécification Event-B est traduite en B classique. Les variables et invariants de typage sont conservés. Les gardes de chaque événement sont transformées en précondition et les postconditions restent inchangées. Chaque opération résultant de la traduction de la machine Event-B est appelée dans l'opération qui résulte de la traduction en B de l'étiquette de transition correspondante. Les obligations preuves

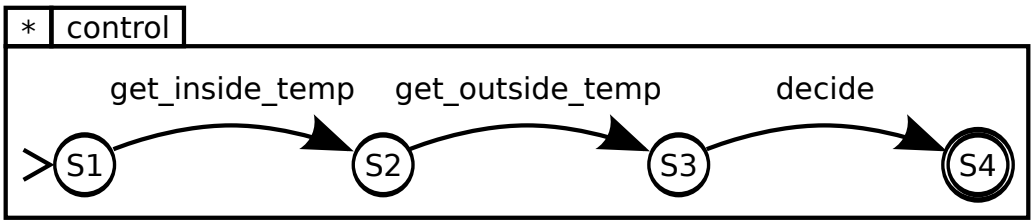


figure 2.2 – Contrôle d'une climatisation : spécification abstraite

prévues dans ce cas par la méthode B permettent de vérifier que la précondition est vraie lorsque l'opération est appelée.

2.1.4 Un exemple : Modélisation d'un système de contrôle de température avec la méthode

Dans cet exemple, nous souhaitons modéliser le contrôle de la température d'une voiture par un système de climatisation. L'utilisateur peut entrer une température souhaitée, dont le système doit se rapprocher, mais pour éviter que la différence de température avec l'extérieur soit trop importante, il peut décider de définir un écart maximal avec la température extérieure. Avant de décider s'il doit mettre en route ou non la climatisation, le système doit donc connaitre les deux températures. Pour cela, il dispose de deux mesures de température : la première est une mesure de la température à l'intérieur et la seconde une mesure de la température à l'extérieur. Le système relève d'abord la température de l'intérieur de l'habitacle, puis la température de l'extérieur. Il décide ensuite s'il doit déclancher la ventilation chaude ou froide ou s'il ne doit rien faire.

La spécification ASTD du système comporte trois transitions. Une transition *get_inside_temp* permet de mesurer la température intérieure, une transition *get_outside_temp* mesure la température extérieure et une transition *decide* décide s'il faut déclancher ou non la ventilation. La spécification du système est présentée à la figure 2.2. Lorsque le système a décidé s'il doit déclancher la ventilation, il mesure à nouveau les températures intérieure puis extérieure, ce qui est spécifié par la fermeture de Kleene ($*$ en haut à gauche de la spécification de la figure 2.2).

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

when

gu1 : $IN \in dom(temp)$

gu2 : $OUT \in dom(temp)$

then

act1 : $temp := \varnothing$

act2 : $state :: STATES$

end

figure 2.3 – Définition de l'événement *decide_act*

Comme indiqué dans la description de la méthode, cette spécification ASTD est accompagnée d'une spécification en Event-B qui décrit les modifications apportées aux données du système. Le système contient un ensemble $MESURES = \{IN, OUT\}$ de deux capteurs. Une fonction *temp* associe la température aux mesures, et une variable *state* contient l'état de la climatisation (*HEAT* s'il faut chauffer, *COOL* s'il faut refroidir, *OFF* si la température ne doit pas changer). Un événement Event-B est créé pour chaque étiquette de transition de la spécification ASTD. L'événement *get_inside_temp_act* associe une température au capteur *IN* et l'événement *get_outside_temp_act* associe une température au capteur *OUT*. L'événement *decide_act* change la valeur de la variable *state* en fonction de la température et de l'écart avec l'extérieur voulu par l'utilisateur. Cette opération est gardée par le fait qu'une valeur de température doit être associée aux deux capteurs et vide la variable *temp* afin de s'assurer que les valeurs des températures soient relues avant de décider à nouveau. La spécification de l'événement *decide_act* est donnée à la figure 2.3. L'action act2 permet de décider la nouvelle valeur de l'état de la climatisation. À ce niveau d'abstraction, le mécanisme qui décrit la manière dont est décidé l'état de la ventilation n'est pas détaillé. On sait simplement que c'est un état choisi dans l'ensemble des états.

Afin de garantir la cohérence horizontale du système, l'ASTD de la figure 2.2 est traduit en B et les événements correspondants sont appelés dans le corps de l'événement traduit. La traduction en B de la transition *decide* est donné à la figure 2.4. Puisque l'événement *decide_act* est appelé, les obligations de preuve générées par

decide =
PRE
State_clim = *S*3
THEN
decide_act||
SELECT
State_clim = *S*3
THEN
State_clim := *S*4
END
END

figure 2.4 – Traduction en B de la transition *decide*

l'atelier B nécessitent de prouver que quand la transition *decide* est appelée (c'est-à-dire que l'ASTD est dans l'état S_3), la garde de *decide_act* est vraie (c'est-à-dire que $IN \in dom(temp)$ et $OUT \in dom(temp)$). On sait alors que lorsque l'ASTD est dans l'état S_3, la garde de l'événement *decide_act* est vrai, et que l'événement peut donc être exécuté.

2.2 Vérification formelle de la cohérence

2.2.1 Objectif

Dans la méthode que nous avons définie, nous utilisons les ASTD pour modéliser l'agencement des actions et Event-B pour modéliser les changements que les actions apportent aux données du système. Cependant, pour permettre la vérification des invariants, certains événements de la modélisation Event-B sont gardés. Pour que l'agencement des actions soit modélisé uniquement par la spécification ASTD, il ne faut pas que cette garde bloque l'exécution de l'événement. En langage naturel, nous voulons alors vérifier la propriété suivante :

Propriété 1. *Si une transition peut être exécutée selon la spécification* ASTD*, alors l'événement correspondant peut être exécuté selon la spécification Event-B.*

Cette proposition signifie inductivement que si une transition est accessible depuis l'état initial d'un ASTD, alors l'événement correspondant doit être accessible depuis l'état initial de la machine Event-B et que si une transition est accessible après une séquence de transitions dans un ASTD, l'événement correspondant en Event-B doit être accessible après la séquence correspondante, c'est-à-dire la séquence des événements correspondants. Ceci signifie alors que si une séquence de transitions peut être executée dans un ASTD, alors la séquence des événements correspondants doit pouvoir être exécutée dans la machine Event-B. L'objectif de cette partie est de traduire cette propriété dans un langage formel, et de montrer que la méthode exposée dans la section 2.1 permet de conserver cette propriété.

2.2.2 Traduction formelle

Sémantique des ASTD

Pour rappel, la sémantique opérationnelle des ASTD est définie par un système de transitions étiquettées. Soit State l'ensemble des états admissibles pour un ASTD, et Event l'ensemble des événements, le système de transitions étiquettées est un sous ensemble de State $\times$ Event $\times$ State. Il s'écrit $s \xrightarrow{\sigma}_A s'$, ce qui signifie que dans l'ASTD A, il est possible de passer de l'état s à l'état s' par l'événement σ. Cependant, il est possible de décrire le système de transitions sans l'indice A si le contexte permet de déduire l'ASTD dont il est question. L'ensemble des règles d'inférence de la sémantique est défini dans le rapport technique [FGLF08]. Afin de formaliser la **Propriété 1**, nous cherchons à étendre cette sémantique aux séquences de transitions.

Soit EventSeq l'ensemble des séquences d'événements. L'ensemble des traces correspond à l'ensemble des séquences d'événements qu'un ASTD est capable d'accepter. Pour décrire cet ensemble, il est nécessaire d'étendre le système de transitions défini dans [FGLF08]. Le système de transitions décrivant l'acceptation d'une séquence d'événements par un ASTD A est un sous ensemble de State $\times$ EventSeq $\times$ State, et s'écrit $s_0 \xrightarrow{t}_{seqA} s$, ce qui signifie que l'ASTD A peut passer de l'état s_0 à l'état s par la séquence d'événements t. Comme pour le système de transition précédent, l'indice

peut être omis si l'ASTD dont il est question peut être déduit du contexte. Les règles d'inférence qui définissent la sémantique de ce système de transitions pour un ASTD A sont les suivantes :

$$empty\ \frac{}{init(A) \xrightarrow{[]}_{seqA} init(A)} \qquad cons\ \frac{s_0 \xrightarrow{t}_{seqA} s' \qquad s' \xrightarrow{\sigma}_{A} s}{s_0 \xrightarrow{t;\sigma}_{seqA} s}$$

La règle *emtpy* signifie qu'une trace vide ne peut partir que de l'état initial (la fonction *init* est la fonction qui à tout ASTD associe son état initial) et ne modifie pas l'état. La règle *cons* signifie que s'il est possible de rejoindre un état s' à partir d'un état s_0 par la séquence d'événements t et qu'il existe une transition σ entre l'état s' et l'état s alors il est possible de rejoindre l'état s à partir de l'état s_0 par la séquence d'événements $t;\sigma$. L'opérateur ; est l'opérateur de concaténation : $t;\sigma$ représente donc la séquence d'événements constituée par la séquence d'événements t suivie de l'événement σ.

En utilisant cette définition, nous définissons la fonction traces, qui à partir d'un ASTD A donne l'ensemble des séquences d'événements que cet ASTD est capable d'accepter. Cette fonction est définie de la manière suivante :

Définition 1. ***Traces d'un*** ASTD

$$\mathit{traces}(A) \triangleq \{t \in \mathit{EventSeq} \mid \exists (s_0, s) \in \mathit{State} \cdot s_0 \xrightarrow{t}_{seq} s\}$$

Sémantique d'Event-B

Event-B [Abr07] est une méthode de spécification basée sur les états. Une machine est définie par un ensemble de variables d'état, un ensemble d'événements qui permettent de modifier ces variables et un invariant que les variables doivent vérifier à tout moment. La sémantique du langage est donnée en associant des obligations de preuve aux machines.

Soit une machine M. Cette machine est constituée d'un ensemble de variables d'état $v \in \mathsf{State}_{Evb}$ qui représentent l'état de la machine et d'un ensemble de $n \in \mathbb{N}$ événements $Ev_i \in \mathsf{Event}_{Evb}, i \in 1..n$ (Event_{Evb} est l'ensmble des événements d'une machine Event-B), ainsi que d'un événement d'initialisation. Les obligations de preuve associées à la machine permettent de montrer que les variables v vérifient un invariant

$I(v)$. Un événement Ev_i est constitué d'une garde $G_i(v)$ et d'un prédicat $BA_i(v, v')$ (pour *Before/After*) qui lie l'état v des variables avant l'exécution de l'événement à l'état v' des variables après l'exécution de l'événement. Dans la syntaxe d'Event-B, l'événement s'écrit :

$$
\begin{array}{l}
Ev_i \triangleq \\
\quad \textbf{when} \\
\quad\quad G_i(v) \\
\quad \textbf{then} \\
\quad\quad BA_i(v, v') \\
\quad \textbf{end}
\end{array}
$$

Un événement peut aussi avoir des paramètres, mais les paramètres ne modifient pas les raisonnements effectués. Dans la suite, nous ne traiterons que d'événements non paramétrés, mais les raisonnements peuvent étendus aux événements paramétrés.

L'événement d'initialisation n'est pas gardé. De plus, puisque c'est un événement d'initialisation, les variables n'ont pas d'état avant l'événement. Il est donc constitué d'un unique prédicat $P_0(v_0)$ qui contraint l'état des variables après l'événement d'initialisation.

Dans le langage Event-B, une obligation de preuve requiert de prouver la faisabilité des événements. Pour l'événement d'initialisation, il s'agit de montrer qu'il existe un état des variables satisfaisant le prédicat d'initialisation. Pour les autres événements, il s'agit de prouver qu'il existe un état des variables qui satisfait le prédicat *Avant/Après* pour tout état des variables vérifiant la garde. Ces obligations de preuve se traduisent de la manière suivante :

$$
\begin{array}{ll}
\exists v_0 \cdot (P_0(v_0)) & \textbf{Initialisation} \\
\forall v \cdot (I(v) \wedge G_i(v) \Rightarrow \exists v' \cdot BA_i(v, v')) & \textbf{Événements}
\end{array}
$$

Pour traduire la propriété 1, nous cherchons à étendre la sémantique aux séquences d'événements en étendant la notion de prédicat *Before/After* aux séquences d'événements Event-B. Pour une séquence d'événements $t \in \mathsf{EventSeq}_{Evb}$, soit $BA_t(v_0, v)$

le prédicat qui lie l'état des variables avant l'exécution de la séquence à l'état des variables après l'exécution de la séquence. Ce prédicat est défini inductivement de la manière suivante :

$$BA_{[]}(v_0, v) \triangleq (v_0 = v \wedge P_0(v_0)) \tag{2.1}$$

$$BA_{t;Ev_i}(v_0, v) \triangleq \exists v' \cdot (BA_t(v_0, v') \wedge G_i(v') \wedge BA_i(v', v)) \tag{2.2}$$

$$\text{avec } Ev_i \triangleq \textbf{when } G_i(v') \textbf{ then } BA_i(v', v) \textbf{ end}$$

La définition 2.1 signifie que l'exécution d'une séquence d'événements vide laisse la machine dans son état inital. La définition 2.2 signifie que la séquence d'événements $t; Ev_i$ (qui est la séquence d'événements constitué de la séquence d'événements t suivi de l'événement Ev_i) peut faire passer les variables de la machine de l'état v_0 à l'état v si il existe un état v' des variables tel que la séquence d'évènents t permet de passer de l'état v_0 à v' ($BA_t(v_0, v')$), que la garde de l'événement Ev_i est vraie dans l'état v' ($G_i(v')$) et que l'événement Ev_i permet de passer de l'état v' à l'état v ($BA_i(v', v)$).

À partir de ce prédicat, nous pouvons définir l'ensemble des traces de la machine Event-B, c'est-à-dire l'ensemble des séquences d'événements telles que le prédicat *Before/After* associé peut être satisfait. Cet ensemble est défini de la manière suivante :

Définition 2. ***Traces d'une machine Event-B***

$$\mathsf{traces}(M) \triangleq \{t \in \mathsf{EventSeq}_{Evb} \mid \exists (v_0, v') \in \mathsf{State}_{Evb} \cdot BA_t(v_0, v')\}$$

Notons que puisque l'obligation de preuve de faisabilité est aussi calculée pour l'événement d'initialisation, toute machine prouvée accepte au moins la trace vide.

Formalisation de la propriété

La propriété 1 impose que lorsqu'il est possible d'exécuter un événement dans un ASTD, l'événement correspondant doit pouvoir être exécuté dans la machine Event-B correspondante. Ce qui peut être exprimé en terme de séquence d'événements de la manière suivante :

Propriété 2. *Une machine Event-B correspondant à un* ASTD *doit pouvoir exécuter toute les séquences d'événements correspondant aux séquences de transitions que peut exécuter l'*ASTD

Soit un ASTD A. Selon la définition de la méthode détaillée à la section 2.1, pour chaque étiquette de l'ASTD, un événement et un seul est défini en Event-B. Soit une machine Event-B M_A^{act} qui contient ces événements. Soit $\alpha(A)$ la liste des étiquettes de transition de l'ASTD A et $\alpha(M_A^{act})$ la liste des étiquettes des événements de la machine M_A^{act}. Il existe une bijection g_1 entre $\alpha(A)$ et $\alpha(M_A^{act})$. Par extension de la notation, cette fonction g_1 peut s'appliquer aux séquences d'événements. Soit une séquence d'événements $t \triangleq t_1; ..., t_n$, la séquence $g_1(t)$ correspond à la séquence $g_1(t_1); ...; g_1(t_n)$. En utilisant la notion de traces, la propriété 2 peut alors être traduite formellement par la propriété suivante :

Propriété 3. ***Propriété formelle***

Soit A *un* ASTD*,*
Soit M_A^{act} *une machine Event-B correpondant à cet* ASTD ***et***
Soit g_1 *la bijection qui lie les transitions en* ASTD *aux événements Event-B*
$g_1(\mathit{traces}(A)) \subseteq \mathit{traces}(M_A^{act})$

Sémantique du langage B

La propriété 3 porte à la fois sur le langage Event-B et sur le langage ASTD. Pour pouvoir la garantir, il faut donc utiliser un langage et une sémantique communs. Pour des raisons pratiques, nous avons choisi ici d'utiliser le langage B classique. Ce langage est un langage de spécification basé sur les états. Une machine est décrite par des variables d'états qui peuvent être modifiées par des opérations et contraintes par des invariants. La première raison du choix du langage B est que ce langage permet l'appel d'opérations au sein d'autres opérations et qu'il génère dans ce cas une obligation de preuve, qui consiste à prouver que la précondition de l'opération appelée est vraie. C'est cette obligation qui va nous permettre de vérifier la propriété 3. La deuxième raison est que le langage permet la modularité, ce qui permet de garder les deux spécifications séparées et de maintenir la dualité ordonnancement/données. Enfin,

une troisième raison plus pragmatique est que le langage B est très bien outillé ce qui facilite l'utilisation de la méthode.

Comme pour le langage Event-B, une machine B est décrite par un ensemble de variables w et un ensemble de $n \in \mathbb{N}$ opérations $Op_i, i \in 1..n$. Ces variables sont initialisées par une opération *init* définie par un prédicat $Q_0(w)$ qui contraint l'état des variables après l'exécution de l'opération. Les variables doivent toujours vérifier un invariants $J(w)$. Une opération contient une précondition et une postcondition et le langage garantit que si la précondition est vraie quand l'opération est exécutée alors les variables sont modifiées selon la postcondition. La précondition est un prédicat $Pre(w)$ sur les variables et la postcondition est un prédicat $BA(w, w')$ qui lie l'état des variables avant l'exécution à l'état des variables après l'exécution.

Le langage B est utilisé comme langage commun à la spécification ASTD et à la spécification Event-B. Il est donc nécessaire de traduire ces deux spécifications dans le langage B. La traduction d'une machine Event-B en machine B est presque immédiate. Les variables sont copiées d'une machine à l'autre. Puisque les deux langages permettent d'écrire les même prédicats, le prédicat qui garde l'événement est recopié et transformé en précondition et la postcondition est également recopiée.

La traduction du langage ASTD en langage B est moins évidente. Des règles de traduction ont été écrites par Milhau dans sa thèse [MGLF12], mais n'ont pas été formellement prouvées. La preuve de ces règles de traduction sera partiellement traitée dans le chapitre 3. Dans ce chapitre, nous supposerons qu'il existe une fonction de traduction qui transforme un ASTD en une machine B et que cette traduction est correcte et complète. Pour exprimer cette fonction de traduction, nous supposons également qu'il existe un prédicat qui lie les états d'un ASTD aux variables d'état d'une machine B. Ce prédicat est vrai si les variables sont dans un état qui correspond à l'état de l'ASTD.

Formellement, soit A un ASTD qui contient un ensemble d'étiquettes $\{\sigma_i \mid i \in 1..n\}$. Soit $M_A = \langle w, \{Op_i | i \in 1..n\}\rangle$ la machine B telle que M_A est la traduction de l'ASTD A en B. Il existe une bijection g_2 entre l'ensemble des étiquettes $\{\sigma_i | i \in 1..n\}$ et l'ensemble des opérations $\{Op_i | i \in 1..n\}$. Soit s un état de A et w un état des variables de M_A, $E(s, w)$ est le prédicat qui est vrai si l'état des variables w de M_A

est équivalent à l'état s de A. Puisque nous supposons que la traduction est correcte et complète, nous supposons que les deux théorèmes suivants sont vrais :

Théorème 1. ***Correction de la traduction***

Initialisation

Soit un état w_0 de la machine M_A

$$Q_0(w_0) \Rightarrow E(init(A), w_0)$$

Opérations

Soient w et w' deux états de la machine M_A

Soit σ une étiquette de transition de A et soit $Op = \langle Pre(w), BA(w, w') \rangle$ l'opération correspondante (i.e. $Op = g_2(\sigma)$)

$$\begin{aligned} &Pre(w) \wedge BA(w, w') \Rightarrow \\ &\quad \forall s, E(s, w) \Rightarrow \\ &\quad\quad \exists s', E(s', w') \wedge s \xrightarrow{\sigma} s' \end{aligned}$$

Théorème 2. ***Complétude de la traduction***

Initialisation

$$\exists w_0 \cdot E(init(A), w_0) \wedge Q_0(w_0)$$

Opérations

*Soient s et s' deux états de l'*ASTD *A*

Soit σ une étiquette de transition de A et soit $Op = \langle Pre(w), BA(w, w') \rangle$ correspondante (i.e. $Op = g_2(\sigma)$)

$$
\begin{aligned}
& s \xrightarrow{\sigma} s' \Rightarrow \\
& \quad \forall w, E(s, w) \Rightarrow \\
& \qquad \exists w', E(s', w') \wedge Pre(w) \wedge BA(w, w')
\end{aligned}
$$

Une machine $M_A^{act} = \langle v, \{Op_{act_i}\}\rangle$ est associée à la machine M_A. Cette machine est la traduction de la machine Event-B qui modélise les actions exécutées sur les variables d'état lors de l'exécution des transitions. Puisque la spécification Event-B est très proche de la spécification B, les deux notations sont ici confondues. Comme pour la traduction, il existe une bijection g_3 qui lie l'ensemble des étiquettes de la machine M_A à l'ensemble des étiquettes de la machine M_A^{act}. Il est possible d'écrire un invariant $J(w, v)$ qui lie l'état des variables de la machine M_A^{act} à l'état des variables de la machine M_A. Chacune des opérations $Op_{act_i} = \langle G_{act_i}(v), BA_{act}(v, v')\rangle$ est appelée dans la substitution de l'opération Op_i correspondante. Dans ce cas, les obligations de preuves de la méthode B nécessite de prouver que $J(w, v) \wedge Pre_i(w) \Rightarrow G_{act_i}(v)$ pour tout $i \in 1..n$. De plus, lors de la transition, les variables de la machine M_A^{act} sont modifiées selon le prédicat $BA_{act_i}(v, v')$.

2.2.3 Preuve de correction de la méthode

Rappels des notations

La méthode définie ci-dessus relie quatre spécifications dans trois langages différents (ASTD, B et Event-B). Une fonction α associe à chacune des spécifications l'ensemble des étiquettes de transition/opération/événement qu'elle contient. Une spécification ASTD, généralement nommée A, spécifie le comportement de la machine. Une machine Event-B, en général appelée M_A^{act} spécifie les changements que chaque transition apporte aux données du système. Cette spécification contient donc un événement pour chaque étiquette de transition de la spécification ASTD (il existe une bijection g_1 entre $\alpha(A)$ et $\alpha(M_A^{act})$).

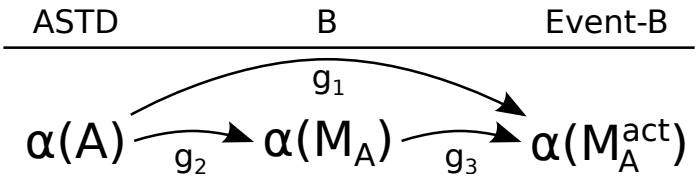


figure 2.5 – Résumé des Notations

La spécification ASTD est traduite en une machine B classique, généralement nommée M_A. Cette machine contient une opération pour chaque étiquette de transition des ASTD, il existe donc une bijection g_2 entre $\alpha(A)$ l'ensemble des étiquettes de transition de la spécification ASTD et $\alpha(M_A)$ l'ensemble des étiquettes des opérations de la machine M_A. De même, la machine M_A^{act} est traduite en B classique afin de pouvoir appeler dans chaque opération correspondant à une étiquette de transition la traduction de l'événement correspondant. Il existe donc une bijection g_3 entre l'ensemble $\alpha(M_A)$ des étiquettes de la spécification B et l'ensemble $\alpha(M_A^{act})$ des étiquettes de la spécification Event-B. Cette bijection est liée aux deux autres par la relation suivante : $g_1 = g_2 \circ g_3$. L'ensemble de ces notations est résumé à la figure 2.5.

Les langages Event-B et B utilisent des notations similaires pour définir les prédicats. Les gardes et préconditions sont décrites dans un même langage de prédicat. Les deux ne sont différents que par le sens qu'on leur donne. Dans la suite, nous utilisons indifféremment les mêmes notations pour désigner la garde d'un événement Event-B et sa traduction en précondition dans une machine B, de même pour les prédicats *Before/After*.

Preuve

Soient un ASTD A spécifiant un ordonnancement, une machine Event-B M_A^{act} spécifiant les changements apportés aux données et une machine M_A qui contient la traduction de l'ASTD A et les appels aux opérations de M_A^{act}. On cherche à prouver le théorème suivant :

Théorème 3. ***Correction de la méthode***

$$g_1(\mathit{traces}(A)) \subseteq \mathit{traces}(M_A^{act})$$

En étendant le théorème de complétude de la traduction aux séquences d'événements, on peut montrer le lemme suivant :

Lemme 1. ***Extension du théorème de complétude aux traces***

Soit t une séquence d'événements

$$\forall(s_0, s) \cdot (s_0 \xrightarrow{t}_{seq} s \Rightarrow \exists(w_0, w) \cdot (E(s_0, w_0) \wedge E(s, w) \wedge BA_{g_2(t)}(w_0, w)))$$

Démonstration. La démonstration se fait par induction sur la structure de la séquence d'événements.

Cas de base :

La trace vide est acceptée par tout ASTD. Elle part de l'état initial. Soit $s_0 = \mathit{init}(A)$.

D'après la partie concernant l'initialisation du **Théorème 2** de complétude, on sait :

$$\exists w_0 \cdot (E(s_0, w_0) \wedge Q_0(w_0))$$

Donc, d'après la définition du prédicat *Avant/Après* pour les séquences d'événements (Définition 2.1)

$$s_0 \xrightarrow{[]}_{seq} s_0 \Rightarrow \exists(w_0) \cdot (E(s_0, w_0) \wedge BA_{[]}(w_0, w_0))$$

Cas inductif : Soient s_0 et s tels que $s_0 \xrightarrow{t;\sigma}_{seq} s$

$$\begin{array}{ll} s_0 \xrightarrow{t;\sigma}_{seq} s & \Rightarrow \text{ (Règle d'inférence } \textit{cons}) \\ \exists s' \cdot (s_0 \xrightarrow{t}_{seq} s' \wedge s' \xrightarrow{\sigma} s) & \end{array}$$

Or

$$\begin{array}{ll} \exists s' \cdot (s_0 \xrightarrow{t}_{seq} s') & \Rightarrow \text{ (Hypothèse d'induction)} \\ \exists (w_0, w') \cdot (E(s_0, w_0) \wedge E(s', w') \wedge BA_{g_2(t)}(w_0, w')) & \end{array}$$

Soient w_0 et w' tels que $E(s_0, w_0) \wedge E(s', w') \wedge BA_{g_2(t)}(w_0, w')$. On a alors :

$$\begin{array}{ll} s' \xrightarrow{\sigma} s \wedge E(s', w') & \Rightarrow \text{ (\textbf{Théorème 2 (de complétude)})} \\ \exists w \cdot (E(s, w) \wedge & \\ \quad Pre_{g_2(\sigma)}(w') \wedge BA_{g_2(\sigma)}(w', w)) & \end{array}$$

Donc finalement :

$$\begin{array}{ll} BA_{g_2(t)}(w_0, w') \wedge Pre_{g_2(\sigma)}(w') \wedge BA_{g_2(\sigma)}(w', w) \Rightarrow & \text{(Propriété 2.2)} \\ \qquad BA_{g_2(t;\sigma)}(w_0, w) & \end{array}$$

□

D'après la définition des traces d'un ASTD et d'une machine B, on peut déduire du **Lemme 1** le lemme suivant sur le lien entre les traces acceptées par un ASTD et les traces acceptées par la machine qui est sa traduction en B :

Lemme 2.

Soit A un ASTD *et M_A la machine obtenue par la fonction de traduction des* ASTD *vers B,*

$$g_2(\mathit{traces}(A)) \subseteq \mathit{traces}(M_A)$$

Notons que dans le cadre de la définition de cette méthode, nous voulons montrer que les traces acceptées par les ASTD sont aussi acceptées par la spécification Event-B. Nous n'avons donc besoin que de l'inclusion des traces. En étendant de

même le théorème de correction de la traduction, on pourrait cependant montrer que $g_2(\mathsf{traces}(A)) = \mathsf{traces}(M_A)$.

Ensuite, puisque la machine M_A^{act} est incluse dans la machine M_A, que les événements correspondants à chaque transition sont appelées dans les événements de la machine M_A et que l'obligation de preuve associée à l'appel d'une opération est prouvée, on peut montrer que s'il est possible de passer d'une variable w_0 à une variable w par une séquence d'événements t dans la machine M_A, alors il est possible de trouver v_0 et v, deux états des variables de la machine M_A^{act} tels que la séquence des événements correspondant $g_3(t)$ permet de passer de v_0 à v, soit le lemme suivant :

Lemme 3.

$$g_3(\mathit{traces}(M_A)) \subseteq \mathit{traces}(M_A^{act})$$

Démonstration. La preuve se fait par induction sur la structure de la séquence d'événements.

Cas de base : D'après l'obligation de preuve sur la faisabilité de l'initialisation, on sait que la machine M_A^{act} accepte la trace vide.

Cas inductif : Soit $t;\sigma \in \mathsf{traces}(M_A)$, on a

$$\begin{aligned} t;\sigma \in \mathsf{traces}(M_A) &\Rightarrow (\textbf{Définition 2}) \\ \exists(w_0, w) \cdot (BA_{t;\sigma}(w_0, w)) &\Rightarrow (\text{Définition 2.2}) \\ \exists(w_0, w', w) \cdot (BA_t(w_0, w') \wedge Pre_\sigma(w') \wedge BA_\sigma(w', w)) & \end{aligned}$$

Or

$$\begin{aligned} \exists(w_0, w') \cdot (BA_t(w_0, w')) &\Rightarrow (\textbf{Définition 2}) \\ t \in \mathsf{traces}(M_A) &\Rightarrow (\text{Hypothèse d'induction}) \\ g_3(t) \in \mathsf{traces}(M_A^{act}) &\Rightarrow (\textbf{Définition 2}) \\ \exists(v_0, v') \cdot BA_{g_3(t)}(v_0, v') & \end{aligned}$$

Soient $(w_0, w', v_0, v') \cdot BA_t(w_0, w') \wedge BA_{g_3(t)}(v_0, v')$

On sait de manière triviale $J(w', v')$ est vrai. Puisque l'obligation de preuve liée à l'appel des opération est prouvée, on sait alors que $Pre_\sigma(w') \wedge J(w', v') \Rightarrow G_{g_3(\sigma)}(v')$, donc $G_{g_3(\sigma)}(v')$ est vraie.

D'après l'obligation de preuve de la faisabilité des événement Event-B on sait alors que $\exists v \cdot BA_{g_3(\sigma)}(v', v)$.

Donc $g_3(t; \sigma) \in \mathsf{traces}(M_A^{act})$

□

Le lemme 2 permet de montrer que $g_2(\mathsf{traces}(A)) \subseteq \mathsf{traces}(M_A)$ et le lemme 3 que $g_3(\mathsf{traces}(M_A)) \subseteq \mathsf{traces}(M_A^{act})$. Donc $g_3(g_2(\mathsf{traces}(A))) = g_1(\mathsf{traces}(A)) \subseteq \mathsf{traces}(M_A^{act})$. On garantit donc que la preuve des machine B et Event-B permet de vérifier qu'un événement Event-B peut toujours être appelé lorsque la transition qui correspond doit être exécutée et donc que notre méthode est correcte.

2.3 Raffinement

Pour permettre la spécification par étapes d'un système, la méthode doit permettre le raffinement. Dans cette section, nous définissons les règles de raffinement qui permettent de raffiner une spécification modélisée en utilisant la méthode définie dans la section 2.1. L'objectif de ces règles de raffinement est double : nous souhaitons d'une part qu'une spécification concrète soit cohérente avec la spécification abstraite qu'elle raffine, et d'autre part que le raffinement permette de simplifier une partie des preuves des machines. Dans la section 2.3.1, nous définissons le raffinement combiné des deux spécifications et montrons qu'il permet de vérifier la cohérence de la spécification concrète avec la spécification abstraite. Dans la section 2.3.2, nous proposons une méthode de raffinement qui permet de simplifier la preuve de la cohérence de chaque niveau.

2.3.1 Définition du raffinement combiné

Des raffinements ont déjà été définis pour les ASTD et pour le langage Event-B. Dans cette section, nous rappelons les définitions formelles de ces deux raffinements

et nous explicitons les conditions dans lesquelles le raffinement combiné permet de maintenir la cohérence.

Le raffinement ASTD

Le raffinement pour les ASTD a été défini dans [FGLM14]. Dans cette partie, nous présentons la définition du raffinement telle qu'elle est définie dans cet article, c'est-à-dire sans prendre en compte le raffinement combiné. Le langage ASTD est un langage de spécification basé sur les événements. Le comportement d'un système spécifié à l'aide d'ASTD est décrit en terme de traces acceptables par cet ASTD. Le raffinement permet l'ajout et le retrait de transitions. Si on se contente d'observer les étiquettes de transition communes, un ASTD concret doit préserver les traces acceptées par un ASTD abstrait qu'il raffine.

Plus formellement, soit A un ASTD abstrait et C un ASTD concret. Soit α la fonction qui associe à un ASTD l'ensemble des étiquettes de transition utilisées dans cet ASTD. Le raffinement ASTD permet l'ajout et le retrait de transitions, donc $\alpha(A)$ n'est pas toujours égal à $\alpha(C)$. Pour définir le raffinement, nous utilisons la notation $A^{\backslash\bar{C}}$, qui désigne l'ASTD A restreint aux événements qui sont communs à A et C, soit plus formellement l'ASTD $A\backslash(\alpha(A) - \alpha(C))$ où l'opérateur $\backslash$ est l'opérateur *hiding* classique des algèbres de processus. Soit A un ASTD et E un ensemble d'étiquettes de transition, $A\backslash E$ est l'ASTD A dans lequel toutes les transitions dont l'étiquette est dans E ont été remplacées par des transitions silencieuses τ.

Le raffinement des ASTD est décrit à l'aide de trois ensembles de séquences d'événements caractérisant une machine : l'ensemble des traces, l'ensemble des deadlock et l'ensemble des τ-enabled. L'ensemble des traces d'un ASTD a été défini dans la partie précédente. L'ensemble des deadlock correspond à l'ensemble des traces d'un ASTD après lesquelles il n'est plus possible d'exécuter une transition. L'ensemble des τ-enabled correspond à l'ensemble des traces après lesquels il est possible d'exécuter une transition silencieuse. Pour un ASTD A, ces ensembles sont définis formellement de la manière suivante :

$$\begin{aligned}
\mathsf{traces}(A) &= \{t \in \mathsf{EventSeq} \mid \exists (s, s') \cdot s \xrightarrow{t}_{seq} s'\} \\
\mathsf{deadlock}(A) &= \{t \in \mathsf{traces}(A) \mid \forall \sigma \in \alpha(A) \cdot (t; \sigma) \notin \mathsf{traces}(A)\} \\
\tau\text{-}\mathsf{enabled}(A) &= \{t \in \mathsf{traces}(A) \mid (t; \tau) \in \mathsf{traces}(A)\}
\end{aligned}$$

On dit alors que C raffine A ($A \sqsubseteq C$) si les conditions suivantes sont vérifiées :

$$\mathsf{traces}(A^{\setminus \bar{C}}) = \mathsf{traces}(C^{\setminus \bar{A}}) \tag{2.3}$$

$$\mathsf{deadlock}(A^{\setminus \bar{C}}) = \mathsf{deadlock}(C^{\setminus \bar{A}}) \tag{2.4}$$

$$\tau\text{-}\mathsf{enabled}(A^{\setminus \bar{C}}) \subseteq \tau\text{-}\mathsf{enabled}(C^{\setminus \bar{A}}) \tag{2.5}$$

Notons que dans [FGLM14], la définitions du raffinement contient deux conditions supplémentaires :

$$\begin{aligned}
\alpha(C) - \alpha(A) &\neq \varnothing \\
\tau\text{-}\mathsf{enabled}(C^{\setminus \bar{A}}) &\neq \varnothing
\end{aligned}$$

Dans notre méthode, nous relachons la définition du raffinement et oublions ces deux conditions, notamment parce que nous souhaitons que notre relation de raffinement soit réflexive et qu'elle permette que deux spécifications équivalentes (c'est-à-dire deux spécifications différentes acceptant exactement les même traces et les même deadlock) se raffinent mutuellement.

Le raffinement Event-B

Le raffinement Event-B est défini dans l'Event-B Book [Abr07]. La méthode Event-B est une méthode de spécification basée sur les états. Une machine est décrite par des variables d'états et par un ensemble d'événements qui modifient ces variables. Le principe du raffinement Event-B est de s'assurer que les modifications des variables

d'état spécifiées dans la spécification concrète sont cohérentes avec les modifications des variables d'état spécifiées dans la spécification abstraite.

Soit une machine abstraite M_A constituée d'un ensemble de variables v_A vérifiant un invariant $I_A(v_A)$ et d'un ensemble d'événements $\{Ev_{A_i} = \langle G_{A_i}(v_A), BA_{A_i}(v_A, v'_A)\rangle, i \in 1..n\}$. Soit une machine concrète M_C qui raffine M_A, constituée d'un ensemble de variables v_C. L'invariant de cette machine $I_C(v_A, v_C)$ permet de relier les variables de la machine concrète aux variables de la machine abstraite. Un ensemble d'événements $\{Ev_{C_j} = \langle G_{C_i}(v_C), BA_{C_i}(v_C, v'_C)\rangle, j \in 1..m\}$ permet de modifier l'état de ces variables. Le raffinement Event-B permet l'ajout d'événements. Cet ajout peut se faire de deux façons différentes : soit l'événement ajouté est un événement réellement nouveau, dans ce cas il se comporte comme s'il raffinait l'événement *skip*, soit l'événement ajouté raffine un événement existant. Il existe donc une fonction partielle f qui associe à un événement concret l'événement abstrait qui est raffiné s'il y a lieu.

En Event-B, le raffinement est défini par les obligations de preuves qui lui sont associées. Pour chaque événement, ces obligations de preuves nécessitent de prouver deux choses : (1) si un événement concret peut être exécuté, alors l'événement abstrait qu'il raffine peut être exécuté (renforcement des gardes) et (2) les modifications apportées aux variables d'état par un événement concret doivent être compatibles avec les modifications apportées par un événement abstrait (simulation). Plus formellement, soient un événement abstrait $Ev_A \triangleq \langle G_A(v_A), BA_A(v_A, v'_A)\rangle$ et un événement concret $Ev_C \triangleq \langle G_C(v_C), BA_C(v_C, v'_C)\rangle$ tels que $f(Ev_C) = Ev_A$, les deux obligations de preuves liées au raffinement pour ces événements sont les suivantes :

$$\forall(v_A, v_C) \cdot (G_C(v_C) \wedge I_C(v_A, v_C) \Rightarrow G_A(v_A))$$
$$\forall(v_A, v_C, v'_C) \cdot (BA_C(v_C, v'_C) \wedge I_C(v_A, v_C) \wedge I_A(v_A) \wedge G_C(v_C) \Rightarrow$$
$$\exists v'_A \cdot (BA_A(v_A, v'_A) \wedge I_C(v'_C, v'_A)))$$

Raffinement de l'exemple

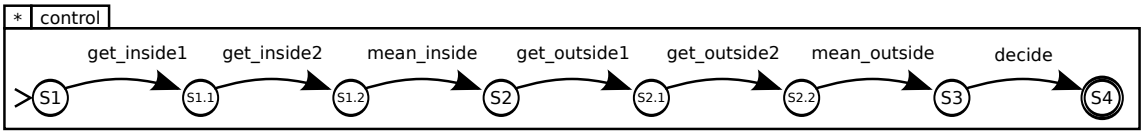


figure 2.6 – Contrôle d'une climatisation : raffinement

Pour rappel, le système modélisé est un système de gestion d'une climatisation. Le système relève les températures à l'intérieur et à l'extérieur d'un véhicule et décide s'il doit ou non chauffer ou refroidir le véhicule. Pour plus de précision du système, nous souhaitons que les températures ne soient plus relevées par un capteur, mais par une moyenne de deux capteurs. Avant d'assigner une température à l'intérieur et à l'extérieur du véhicule, le système relève donc deux capteurs et assigne la moyenne des deux valeurs relevées à la température. La nouvelle spécification est donnée à la figure 2.6.

Pour savoir si l'ASTD de la figure 2.6 raffine celui de la spécification de la figure 2.2, il faut remplacer dans chaque ASTD toute les transitions qui ne sont pas communes aux deux ASTD par des transitions silencieuses et s'assurer que les spécifications obtenues vérifient les propriétés 2.3 à 2.5. Ici, cela revient à masquer dans les deux ASTD toutes les transitions entre l'état S_1 et l'état S_3. Les seuls traces admissibles dans les deux ASTD sont toutes les successions arbitrairement longues de *decide*. Les deux ASTD n'ont aucun *deadlock*. Entre chaque *decide*, les deux ASTD peuvent exécuter une transition silencieuse. L'ASTD de la figure 2.6 raffine donc bien l'ASTD de la figure 2.2.

Dans la partie donnée, deux ensembles de capteurs sont ajoutés. L'ensemble $IN_SENSORS$ contient les deux capteurs IN_SENSOR_1 et IN_SENSOR_2 qui permettent de relever la température à l'intérieur de la pièce et l'ensemble $OUT_SENSORS$ contient les deux capteurs OUT_SENSOR_1 et OUT_SENSOR_2 qui permettent de relever les températures à l'extérieur. Une fonction *in_temp* associe une température aux capteurs intérieurs et une fonction *out_temp* associe une température aux capteurs extérieurs. Les événements associés aux transitions qui doivent relever la température à l'intérieur (get_inside_1 et get_inside_2) associent une valeur aux capteurs dans la fonction *in_temp*. De même, les fonctions qui doivent relever la température à l'extérieur ($get_outside_1$ et $get_outside_2$) associent une valeur aux capteurs dans la fonction *out_temp*. La fonction *temp* reste inchangée. L'événement *mean_inside_act*

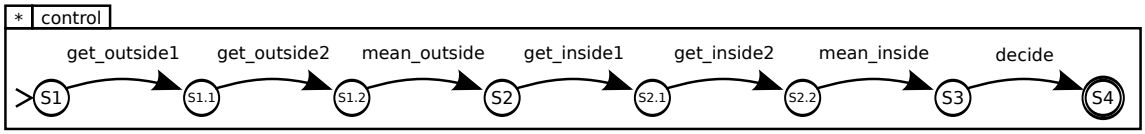


figure 2.7 – Contrôle d'une climatisation : mauvais raffinement

(resp. *mean_outside_act*) associe au capteur IN (resp. OUT) la valeur correspondant à la moyenne des deux capteurs de l'intérieur (resp. de l'extérieur) et raffine l'événément abstrait *get_inside_temp_act* (resp. *get_outside_temp_act*). On montre grâce aux obligations de preuve générées par l'outil Rodin que la spécification de données concrète raffine la spécification de données abstraite.

Cependant, vérifier ces deux raffinements indépendemment ne suffit pas. Reprennons l'ASTD de la figure 2.6 en inversant les transitions pour l'intérieur et l'extérieur comme sur l'ASTD de la figure 2.7 et en ne changeant pas la spécification Event-B. La spécification Event-B concrète raffine évidemment la spécification abstraite (puisqu'elle ne change pas). De même, les transitions qui sont inversées dans la figure 2.7 sont des transitions qui étaient masquées dans la vérification du raffinement ASTD. Donc l'ASTD de la figure 2.7 raffine l'ASTD de la figure 2.2. Cependant, le comportement n'a pas véritablement été préservé. Il est par exemple possible de montrer sur la spécification abstraite qu'une valeur est d'abord assignée à la température intérieur avant la température extérieur (par exemple en ajoutant l'invariant $card(temp) = 1 \Rightarrow IN \in \mathrm{dom}(temp)$). Cet invariant (implicite) est violé sur la spécification concrète. Nous souhaitons donc définir un raffinement plus strict qui permet d'éviter de valider ce type de raffinement.

Le raffinement combiné

Avant de définir le raffinement combiné des deux langages de spécification, il est nécessaire de définir les propriétés que le raffinement doit vérifier. Actuellement, aucune définition de raffinement précise et formelle ne fait consensus. Il faut donc choisir parmis les notions communément admises des propriétés du raffinement qui traduisent la sémantique de notre raffinement. Dans notre cas, nous souhaitons que le raffinement vérifie les propriétés appelées par Boiten [Boi14] *consistency* (cohérence du

raffinement) et *enableness* (accessibilité du raffinement). Ces propriétés sont décrites pour un raffinement dans lequel la spécification abstraite et la spécification concrète contiennent les même événements. De manière informelle, la cohérence se traduit par le fait que les effets d'un événement concret sont prévus par la spécification abstraite. L'accessibilité signifie que si un événement peut être exécuté dans la spécification abstraite, il doit pouvoir être exécuté dans la spécification concrète.

Notre méthode combine deux types de spécification. La spécification ASTD est basée sur les événements, et la spécification Event-B est basée sur les états. Pour traduire les propriétés, nous auront donc besoin de combiner les deux paradigmes. Pour les spécifications basées sur les événements, le raffinement s'exprime généralement en terme de séquences d'événements acceptées par la spécification, tandis que les spécification basées sur les états s'expriment en terme d'effet des événements sur les variables d'état (*cf.* section 1.2). Dans ce cas, les propriétés que nous voulons verifier et qui traduisent la cohérence et l'accessibilité, peuvent être exprimées informellement de la manière suivante :

— **cohérence du raffinement** : Si une séquence d'événements est acceptée par la spécification concrète et a un effet sur les variables d'état, alors une séquence d'événements correspondant est acceptée par la spécification abstraite et l'effet de cette séquence sur les variables d'état correspond.

— **accessibilité du raffinement** : Si un événement abstrait est autorisé dans un état, un événement concret correspondant doit être autorisé dans un état correspondant.

Les définitions de cohérence et d'accessibilité utilisent la notion d'*événements correspondants*. En effet, rien n'oblige une spécification ASTD concrète à utiliser le même ensemble d'étiquettes que la spécification abstraite qu'elle raffine. Cependant, il est possible (et souvent vrai) que les deux ensembles d'étiquettes contiennent des étiquettes en commun. On dit donc que les événements dont les étiquettes sont communes correspondent. De plus, une spécification Event-B est associée à la spécification ASTD, et les transitions modifient les variables d'état associées au système. Dans le raffinement Event-B, il est possible d'associer à un événement concret l'événement qu'il raffine, par le mot-clé *refines*. Si un événement concret raffine un événement abstrait, on dit alors que les événements correspondent. Puisqu'on souhaite que toutes

les séquences d'événements concrets soient conservées, chaque événement abstrait a au moins un événement concret qui lui correspond.

Cette précision va nous permettre de définir plus formellement les deux propriétés. Soient un ASTD A abstrait et un ASTD C concret et soient la machine Event-B M_A^{act} contenant la spécification de donnée associée à A et la machine Event-B M_C^{act} contenant la spécification de données associée à C. Puisqu'il est possible de définir une relation de raffinement en Event-B, il existe une fonction partielle f entre l'ensemble des étiquettes de la machine concrète M_C^{act} et l'ensemble des étiquettes de la machine abstraite M_A^{act} ($f \in \alpha(M_C^{act}) \nrightarrow \alpha(M_A^{act})$). Puisque tous les événements abstraits doivent au moins avoir un événement qui les raffine, on sait que $ran(f) = \alpha(M_A^{act})$. Ces deux spécifications sont écrites en utilisant la méthode définie à la section 2.1. Il existe donc une bijection g_{A_1} entre l'ensemble des étiquettes de A et l'ensemble des étiquettes de M_A^{act} et une bijection g_{C_1} entre l'ensemble des étiquettes de C et l'ensemble des étiquettes de M_C^{act}. Il existe de même les bijection g_{A_2} et g_{A_3} et les bijections g_{C_2} et g_{C_3}. On définit également la fonction f' qui transpose la fonction f sur les étiquettes des ASTD et l'étend pour en faire une fonction totale. Elle est définie par les deux équivalences suivantes :

$$e_C \mapsto e_A \in f' \Leftrightarrow g_{C_1}(e_C) \mapsto g_{A_1}(e_A) \in f \tag{2.6}$$

$$e_C \mapsto \tau \in f' \Leftrightarrow g_{C_1}(e_C) \notin \mathrm{dom}(f) \tag{2.7}$$

L'ensemble des notations est résumé à la figure 2.8

Dans la suite, la machine M_A^{act} contient un ensemble de variables v_A qui vérifient un invariant $I_A(v_A)$. De même la machine M_C^{act} contient un ensemble de variables v_C. Le raffinement Event-B permet de définir dans la spécification concrète un invariant (dit de collage) $I_C(v_A, v_C)$ liant les variables abstraites aux variables concrètes. C'est cet invariant qui permet de définir la notion de d'état correspondant des variables. Nous utilisons également les notations définies dans la section 2.2. Les deux propriétés peuvent alors être traduites formellement de la manière suivante :

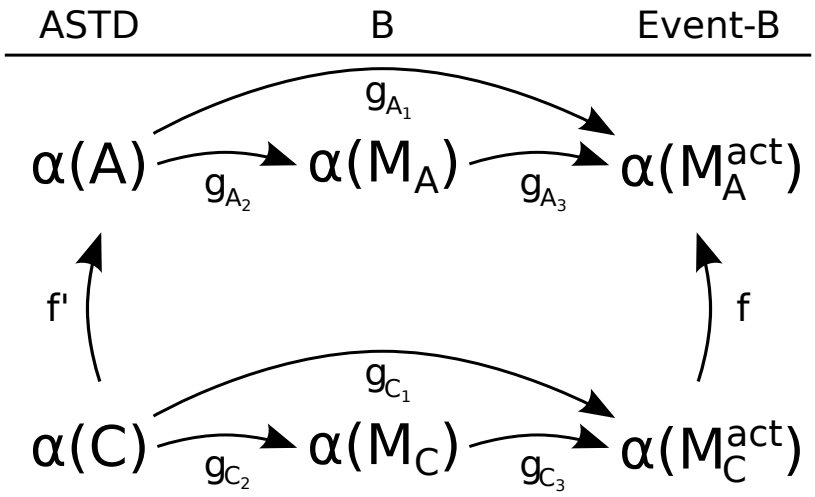


figure 2.8 – Résumé des notations pour le raffinement

— **cohérence**

$$\forall(t_C, v_{A_0}, v_{C_0}, v_C) \cdot (t_C \in \mathsf{traces}(C) \wedge BA_{g_{C_1}(t_C)}(v_{C_0}, v_C) \wedge I_C(v_{A_0}, v_{C_0}) \Rightarrow (f'(t_C) \in \mathsf{traces}(A) \wedge \exists v_A \cdot (BA_{g_{A_1}(f'(t_C))}(v_{A_0}, v_A) \wedge I_C(v_A, v_C)))) \tag{2.8}$$

— **accessibilité**

$$\forall t_A \cdot (t_A \in \mathsf{traces}(A) \Rightarrow f'(t_A) \in \mathsf{traces}(C)) \tag{2.9}$$

$$\mathsf{deadlock}(A) = f'(\mathsf{deadlock}(C))) \tag{2.10}$$

Les propriétés 2.8 et 2.9 permettent de garantir la préservation des traces. La figure 2.9 illustre l'importance de la propriété 2.10. Dans l'exemple de cette figure, le raffinement vérifie les propriétés 2.8 et 2.9, mais par la propriété 2.10. En effet, si la transition $trans_{1_2}$ raffine la transition $trans_1$, que la transition $trans_{1_1}$ raffine la transition $trans_1$ ou qu'elle ne raffine aucune transition, les traces des deux spécifications sont égales (à la fonction f' près). Cependant, ce type de raffinement n'est pas

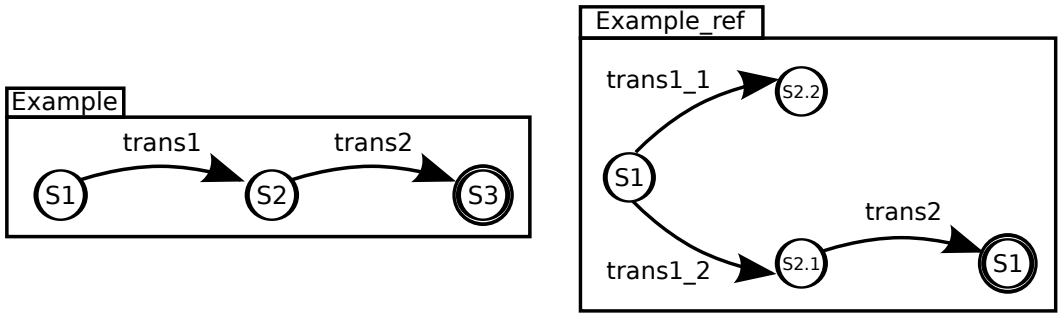


figure 2.9 – Un exemple de mauvais raffinement autorisé sans la propriété 2.10

souhaitable dans notre méthode. Mais la propriété 2.10 permet de rejeter ce type de cas.

Maintenant que nous avons défini les propriétés que nous voulons vérifier pour notre raffinement, nous pouvons définir un ensemble de conditions à vérifier pour qu'une spécification abstraite raffine une spécification concrète. Ces conditions sont données à la **Définition 3**. Nous montrons ensuite que si une spécification concrète définie selon la méthode décrite dans la section 2.1 raffine une spécification abstraite définie selon la méthode décrite dans la section 2.1 selon la définition 3, alors elle vérifie les propriétés de cohérence et d'accessibilité.

Définition 3. ***Raffinement combiné des*** ASTD ***et de la méthode B***

Soient un ASTD A *abstrait et un* ASTD C *concret.*
Soit une machine M_A^{act} *qui est la spécification de donnée de* A*.*
Soit une machine M_C^{act} *qui est la spécification de donnée de* C*.*
Soit f *la fonction de raffinement liant les étiquette de* M_C^{act} *aux étiquettes de* M_A^{act} *et* f' *sa fonction associée pour* C *et* A*.*
On dit que $A \sqsubseteq C$ *ssi*

$$M_A^{act} \sqsubseteq M_C^{act} \tag{2.11}$$

$$\mathsf{traces}(A) = f'(\mathsf{traces}(C)) \tag{2.12}$$

$$\mathsf{deadlock}(A) = f'(\mathsf{deadlock}(C)) \tag{2.13}$$

De manière immédiate, l'égalité 2.12 implique la propriété 2.9 et l'égalité 2.13 implique la propriété 2.10. Une preuve par induction permet de montrer que les propriétés 2.11 et 2.12 impliquent la propriété 2.8. Pour cela, il faut démontrer le théorème suivant :

Théorème 4. *Soit un* ASTD A *abstrait et un* ASTD C *concret,*
Soit une machine M_A^{act} *qui est la spécification de donnée de* A*,*
Soit une machine M_C^{act} *qui est la spécification de donnée de* C*,*
Soit f *la fonction de raffinement liant les étiquette de* M_C^{act} *aux étiquettes de* M_A^{act} *et* f' *sa fonction associée pour* C *et* A*,*

$$
\begin{aligned}
&M_A^{act} \sqsubseteq M_C^{act} \wedge \\
&\mathit{traces}(A) = f'(\mathit{traces}(C)) \Rightarrow \\
&\quad \forall (t_C, v_{A_0}, v_{C_0}, v_C) \cdot (t_C \in \mathit{traces}(C) \wedge BA_{g_{C_1}(t_C)}(v_{C_0}, v_C) \wedge \\
&\qquad I_C(v_{A_0}, v_{C_0}) \Rightarrow \\
&\qquad\quad f'(t_C) \in \mathit{traces}(A) \wedge \\
&\qquad\qquad \exists v_A \cdot (BA_{g_{A_1}(t_A)}(v_{A_0}, v_A) \wedge I_C(v_A, v_C))))
\end{aligned}
$$

Ce théorème est démontré en utilisant les deux lemmes suivant :

Lemme 4. *Soit un* ASTD A *concret et un* ASTD C *abstrait*
Soit une machine M_A^{act} *qui est la spécification de donnée de* A*,*
Soit une machine M_C^{act} *qui est la spécification de donnée de* C*,*
Soit f *la fonction de raffinement liant les étiquettes de* M_C^{act} *aux étiquettes de* M_A^{act} *et* f' *sa fonction associée pour* C *et* A*,*

$$
\begin{aligned}
&\mathit{traces}(A) = f'(\mathit{traces}(C)) \Rightarrow \\
&\quad \forall t_C \cdot (t_C \in \mathit{traces}(C) \Rightarrow \\
&\qquad f'(t_C) \in \mathit{traces}(A))
\end{aligned}
$$

Démonstration. La preuve de ce lemme est triviale :

Soit $t_C \in \mathsf{traces}(C)$, on a

$$\begin{array}{ll} t_C \in \mathsf{traces}(C) & \Rightarrow \\ f'(t_C) \in f'(\mathsf{traces}(C)) & \Rightarrow \\ f'(t_C) \in \mathsf{traces}(A) & \end{array}$$

□

Puis le lemme suivant :

Lemme 5. *Soit un* ASTD A *concret et un* ASTD C *abstrait*

Soit une machine M_A^{act} *qui est la spécification de donnée de* A*,*

Soit une machine M_C^{act} *qui est la spécification de donnée de* C*,*

Soit f *la fonction de raffinement liant les étiquettes de* M_C^{act} *aux étiquettes de* M_A^{act} *et* f' *sa fonction associée pour* C *et* A*,*

$$\begin{array}{l} M_C^{act} \sqsubseteq M_A^{act} \wedge \\ \mathsf{traces}(A) = f'(\mathsf{traces}(C)) \Rightarrow \\ \quad \forall (t_C, t_A, v_{C_0}, v_C, v_{A_0}) \cdot (t_C \in \mathsf{traces}(C) \wedge t_A \in \mathsf{traces}(A) \wedge t_A = f'(t_C) \wedge \\ \qquad\qquad BA_{g_{C_1}(t_C)}(v_{C_0}, v_C) \wedge I_C(v_{C_0}, v_{A_0}) \Rightarrow \\ \qquad\qquad\quad \exists v_A \cdot (BA_{g_{A_1}(t_A)} \wedge I_C(v_C, v_A))) \end{array}$$

Démonstration. La preuve se fait par induction sur la séquence d'événements.

Cas de base : $t_C = []$

Il existe un v_{C_0} tel que $I_C(v_{A_0}, v_{C_0})$ d'après la preuve de faisabilité de l'initialisation et la preuve de simulation de l'initialisation d'Event-B

Cas inductif : Soit $t_C; \sigma_C \in \mathsf{traces}(C)$

Soit $t_A \in \mathsf{traces}(A)$ tel que $t_A = f'(t_C; \sigma_C) = f'(t_C); f'(\sigma_C)$

Donc $t_A = t'_A; \sigma'_A$ avec $t'_A = f'(t_C)$ et $\sigma'_A = f'(\sigma_C)$

On sait

$$\begin{aligned} & BA_{g_{C_1}(t_C;\sigma_C)}(v_{C_0}, v_C) && \Rightarrow \text{ (Définition 2.2)} \\ \exists v'_C \cdot (& BA_{g_{C_1}(t_C)}(v_{c_0}, v'_C) \wedge \\ & BA_{g_{C_1}(\sigma_C)}(v'_C, v_C) \wedge G_{g_{C_1}(\sigma_C)}(v'_C)) \end{aligned}$$

Si $g_{C_1}(\sigma_C) \notin \text{dom}(f)$ ($g_{C_1}(\sigma_C)$ ne raffine pas d'événement abstrait dans la spécification Event-B)

$\sigma'_A = \tau$

Alors on a

$$\begin{aligned} & t_A = f'(t_C) && \Rightarrow \text{ Hypothèse d'induction} \\ \exists v_A \cdot (& BA_{g_{A_1}(t_A)}(v_{A_0}, v_A) \wedge I_C(v_A, v'_C)) \end{aligned}$$

Or si $g_{C_1}(\sigma_C) \notin dom(f)$, cela signifie que σ_C raffine *skip*

Donc

$$\forall (v'_C, v_C, v_A) \cdot (BA_{g_{C_1}(\sigma_C)}(v'_C, v_C) \wedge I_C(v_A, v'_C) \Rightarrow I_C(v_A, v_C))$$

Donc

$$BA_{g_{A_1}(t_A)}(v_{A_0}, v_A) \wedge I_C(v_A, v_C)$$

Si $g_{C_1}(\sigma_C) \in \text{dom}(f)$ ($g_{C_1}(\sigma_C)$ raffine un événement abstrait dans la spécification Event-B)

$t'_A = f'(t_C)$ et $\sigma'_A = f'(\sigma_c)$ et $g_{A_1}(\sigma'_A) = f(g_{C_1}(\sigma_C))$

On a

$$\begin{aligned} & t'_A = f'(t_C) \wedge BA_{g_{C_1}(t_C)}(v_{C_0}, v'_C) \wedge I_C(v_{C_0}, v_{A_0}) && \Rightarrow \text{ (Hypothèse d'induction)} \\ \exists v'_A \cdot (& BA_{g_{A_1}(t'_A)}(v_{A_0}, v'_A) \wedge I_C(v'_A, v'_C)) \end{aligned}$$

Donc

$$G_{g_{C_1}(\sigma_C)}(v'_C) \wedge I_C(v'_A, v'_C) \Rightarrow \quad \text{(Renforcement des gardes)}$$
$$G_{g_{A_1}(\sigma'_A)}(v'_A)$$

Et

$$G_{g_{A_1}(\sigma_A)}(v'_A) \wedge G_{g_{C_1}(\sigma_C)}(v'_C) \wedge$$
$$I_C(v'_A, v'_C) \wedge BA_{g_{C_1}(\sigma_C)}(v'_C, v_C) \quad \Rightarrow \quad \text{(Preuve de simulation)}$$
$$\exists v_A \cdot (BA_{g_{A_1}(\sigma'_A)}(v'_A, v_A) \wedge I_C(v_C, v_A))$$

□

On peut ensuite déduire le théorème 4 de ces deux lemmes. Donc si une spécification abstraite écrite en suivant la méthode définie dans la section 2.1 est raffinée par une spécification suivant la méthode définie dans la section 2.1 et que les deux spécifications suivent la **Définition 3**, alors le raffinement vérifie les propriétés de cohérence et d'accessibilité, ce qui se résume par le théorème suivant :

Théorème 5. ***Cohérence et accessibilité du raffinement***
Si une spécification abstraite est raffinée par une spécification concrète suivant la ***Définition 3****, alors les deux spécifications vérifient les propriétés d'accessibilité et de cohérence définies par Boiten (Propriétés (2.8), (2.9), (2.10))*

2.3.2 Utilisation du raffinement pour simplifier la preuve

La vérification de la cohérence de la méthode présentée dans la section 2.2 permet de montrer que les événements spécifiant les changements apportés aux données sont appelés dans des conditions où leur garde est vraie. Cependant, la vérification de la cohérence (c'est-à-dire la preuve de la spécification B) génère un très grand nombre d'obligations de preuve (des chiffres illustrant le nombre des obligations de preuves sont donnés dans le chapitre 5).

L'idée de la preuve de la cohérence est de montrer que dans l'état où est déclanchée une transition, la garde de l'événement associé est vraie. Or, au cours du raffinement,

nous constatons dans un grand nombre de cas que la garde d'un événement abstrait et la garde de l'événement concret qui le raffine sont équivalentes. Parfois, l'ajout de variables concrètes renforce la garde, mais la condition imposée par la garde aux variables communes est dans ce cas généralement équivalente. De plus, le raffinement des ASTD préserve les traces. Il semble donc intuitif de dire que si une transition concrète est permise, la transition abstraite correspondante doit être permise. Or la preuve de la cohérence au niveau abstrait montre que si la transition abstraite est permise, la garde de l'événement associé doit être vraie. Si la garde d'un événement abstrait est en partie équivalente à la garde concrète, il pourrait être intéressant d'utiliser ces preuves pour alléger la preuve de cohérence du niveau concret.

Retour à l'exemple

La remarque du paragraphe précédent concerne les événements concrets qui raffinent un événement abstrait. Dans notre exemple, l'événement *mean_inside* raffine l'événement *get_inside_temp*. Les éviments concrets et abstrait associés à ces deux transitions sont données à la figure 2.10. La variable *temp* est commne entre la spécification abstraite et la spécification concrète. On remarque dans les événements de la figure 2.10 que la garde à propos de la variable *temp* est identique dans les deux événements. Il serait donc souhaitable d'utiliser cette propriété pour ne pas devoir reprouver la cohérence pour ces gardes.

Formalisation

Soient deux spécifications ASTD A et C et soient M_A^{act} et M_C^{act}. La machine M_A^{act} contient un ensemble de variables v_A et la machine M_C^{act} contient un ensemble de variables v_C satisfiant l'invariant $I_C(v_A, v_C)$. Soient un événement $e_{A_{act}}$ et un événement $e_{C_{act}}$ tels que $e_{A_{act}} = f(e_{C_{act}})$ (donc $e_{C_{act}}$ raffine $e_{A_{act}}$). L'événement $e_{C_{act}}$ est gardé par une garde $G_C(v_C)$ et l'événement $e_{A_{act}}$ est gardée par une garde $G_A(v_A)$. On suppose alors qu'il existe $G_{C_C}(v_C)$ et $G_{C_A}(v_C)$ telles que $G_{C_C}(v_C) \wedge G_{C_A}(v_C) \Rightarrow G_C(v_C)$. L'objectif est de montrer que si on a $I_C(v_C, v_A) \wedge G_A(v_A) \Rightarrow G_{C_A}(v_C)$ et qu'on remplace la garde $G_C(v_C)$ de l'événement $e_{C_{act}}$ par la garde $G_{C_C}(v_C)$ dans la preuve de cohérence de la méthode, alors la méthode reste cohérente.

Event *get_inside_temp_act* $\widehat{=}$

when

gu1 : $IN \notin dom(temp)$

then

act1 : $temp : |(...)$

end

(a) Événement *get_inside_temp_act*

Event *mean_inside_act* $\widehat{=}$

when

gu1 : $IN \notin dom(temp)$

gu2 : $IN_SENSORS \subseteq dom(in_temp)$

then

act1 : $temp := (...)$

act2 : $in_temp := \varnothing$

end

(b) Événement *mean_inside_act*

figure 2.10 – Comparaison de l'évément *get_inside_temp_act* et de l'événement *mean_inside_act* qui le raffine

Dans la partie précédente, nous avons constaté que les gardes étaient souvent identiques pour les variables communes aux spécifications abstraite et concrète. Par variables communes, nous entendons les variables identiques mais également les variables concrètes liées aux variables abstraites par l'invariant. Dans ce cas, la garde G_{C_A} est la garde qui contient les conditions sur les variables liées aux variables abstraites et G_{C_C} les conditions sur les variables introduites dans la machine abstraite. Notons cependant qu'il est possible d'avoir une spécification de donnée concrète qui ne vérifie pas ce critère et qui pourtant raffine la spécification abstraite. Dans ce cas, la difficulté est de trouver la garde G_{C_A} vérifiant le critère $I_C(v_C, v_A) \wedge G_A(v_A) \Rightarrow G_{C_A}(v_C)$. Notons également qu'il est toujours possible de prendre la garde toujours vraie pour la garde G_{C_A}, ce qui nous ramène au raffinement précédent.

Dans l'exemple de la figure 2.10, on cherche à séparer la garde $IN \notin \text{dom}(temp) \wedge IN_SENSORS \subseteq \text{dom}(in_temp)$ en deux gardes, la première concernant les variable abstraites et la seconde les variables concrètes. On prend ici $G_{C_A}(v_C) = IN \notin \text{dom}(temp)$ et $G_{C_C}(v_C) = IN_SENSORS \subseteq \text{dom}(in_temp)$. On montre ensuite

trivialement que :

$$I_C(v_C, v_A) \wedge G_A(v_A) \Rightarrow G_{C_A}(v_C)$$
$$G_{C_C}(v_C) \wedge G_{C_A}(v_C) \Rightarrow G_C(v_C)$$

avec $G_A(v_A) = IN \notin \mathrm{dom}(temp)$ et $I_C(v_C, v_A) = \mathbf{True}$.

On appel $M_{c_c}^{act}$ la machine correspondant à la machine M_C^{act} dans laquelle les gardes des événements ont été remplacées par la garde G_{C_C} définie ci-dessus. On cherche alors à montrer la propriété suivante :

Propriété 4. ***Raffinement combiné pour simplifier la preuve***

*Si la machine $M_{c_c}^{act}$ et l'*ASTD *C vérifient la méthode définie à la section 2.2*

Si pour tout événement $Ev_C \in \mathrm{dom}(f)$ et tout événement $Ev_A = f(Ev_C)$

— $I_C(v_C, v_A) \wedge G_A(v_A) \Rightarrow G_{c_A}(v_C)$

— $G_{C_A}(v_C) \wedge G_{C_C}(v_C) \Rightarrow G_C(v_C)$

Alors $g_1(\mathsf{traces}(C)) \subseteq \mathsf{traces}(M_C^{act})$

Démonstration. Cette propriété est prouvée par induction sur la structure de la séquence d'événements.

Cas de base : $t_C = []$

La trace vide est une trace acceptée par toute machine Event-B prouvée

Cas inductif :

Soit $t_C; \sigma_C \in \mathsf{traces}(C)$

$$\begin{aligned}
t_C; \sigma_C \in \mathsf{traces}(C) &\Rightarrow (\textbf{Définition 1}) \\
\exists (s_{C_0}, s_C) \cdot (s_{C_0} \xrightarrow{t_C;\sigma_C}_{seq} s_C) &\Rightarrow (\text{Règle d'inférence } \textit{cons}) \\
\exists (s_{C_0}, s'_C, s_C) \cdot (s_{C_0} \xrightarrow{t_C}_{seq} s'_C \wedge s'_C \xrightarrow{\sigma_C} s_C) &\Rightarrow (\textbf{Définition 1}) \\
t_c \in \mathsf{traces}(C) &\Rightarrow (\text{Hypothèse d'induction}) \\
g_1(t_C) \in \mathsf{traces}(M_C^{act}) &\Rightarrow (\textbf{Définition 2}) \\
\exists (v_{C_0}, v'_C) \cdot (BA_{g_{C_1}(t_C)}(v_{C_0}, v'_C)) &
\end{aligned}$$

Si $g_1(\sigma_c) \notin \mathrm{dom}(f)$

Soit $Pre_{g_2(\sigma_C)}(w_C)$ la précondition de l'opération B $g_2(\sigma_C)$ qui est la traduction en B de σ_C

On a alors :

$$\begin{array}{rl} s_{c_0} \xrightarrow{t_C}_{seq} s'_C & \Rightarrow \ (\textbf{Lemme 1}) \\ \exists (w_{C_0}, w'_C) \cdot (BA_{g_{C_2}(t_c)}(w_{C_0}, w'_C) \wedge E_C(s_{C_0}, w_{C_0}) \wedge E_C(s'_C, w'_C)) & \end{array}$$

$$\begin{array}{rl} s'_C \xrightarrow{t_C}_{seq} s_C \wedge E_C(s'_C, w'_C) & \Rightarrow \ (\textbf{Théorème 1}) \\ \exists w_C \cdot (Pre_{g_{C_2}(e_C)}(w'_C) \wedge BA_{g_{C_2}(e_C)}(w'_C, w_C) \wedge E_C(s_C, w_C)) & \end{array}$$

Puisque $g_1(\sigma_C)$ est dans la machine $M_{c_c}^{act}$, et que la machine $M_{c_c}^{act}$ est liée à C par la méthode, on a montré que $Pre_{g_1(\sigma_C)}(w'_C) \Rightarrow G_{g_A(\sigma_C)}(v'_C)$. De plus, trivialement on a $I_C(v'_A, v'_C)$ (par définition de l'invariant).

$$\begin{array}{rl} I_c(v'_A, v'_C) \wedge G_{g_{C_1}(\sigma_C)}(v'_C) & \Rightarrow \ (\text{Faisabilité de l'événement}) \\ \exists v_C \cdot (BA_{g_{C_1}(\sigma_C)}(v'_C, v_C)) & \end{array}$$

Donc

$$\begin{array}{rl} \exists (v_{C_0}, v_C) \cdot (BA_{g_{C_1}(t_C;\sigma_c)}(v_{C_0}, v_C)) & \Rightarrow \ (\textbf{Définition 2}) \\ g_1(t_C; \sigma_C) \in \mathsf{traces}(M_C^{act}) & \end{array}$$

Si $g_1(\sigma_C) \in \mathrm{dom}(f)$

Soient $G_{g_1(\sigma_C)_C}(v'_C)$ et $G_{g_1(e_C)_A}(v'_C)$ telles que

$$G_{g_1(e_C)_C}(v'_C) \wedge G_{g_1(e_C)_A}(v'_C) \Rightarrow G_{g_1(e_C)}(v'_C)$$

On montre en utilisant le même raisonnement que précédemment que $G_{g_1(e_C)_C}(v'_C)$ (La garde $G_{g_1(e_C)_C}(v'_C)$ est la garde de l'événement $g_1(\sigma_C)$ dans la machine $M^{act}_{c_c}$. Or cette machine est couplée à l'ASTD C suivant la méthode. Donc s'il est possible de faire σ_C, la garde de l'événement associé est vraie)

Il reste à montrer que $G_{g_1(e_C)_A}(v'_C)$

$$
\begin{array}{rll}
t_C;\sigma_C \in \mathsf{traces}(C) & \Rightarrow & (\textbf{Définition 3 (du raffinement)}) \\
f'(t_C;\sigma_C) \in \mathsf{traces}(A) & \Rightarrow & (\textbf{Définition 1}) \\
\exists (s_{A_0}, s_A) \cdot (s_{0_A} \xrightarrow{f'(t_C;\sigma_C)}_{seq} s_A) & \Rightarrow & (\text{Règle d'inférence } \textit{cons}) \\
\quad \exists s'_A \cdot (s_{0_A} \xrightarrow{f'(t_C)}_{seq} s'_A \wedge s'_A \xrightarrow{f'(\sigma_C)} s_A) & &
\end{array}
$$

Or, d'après le lemme d'extension de la complétude de la traduction aux traces, on a :

$$
\begin{array}{rll}
s_{A_0} \xrightarrow{f'(t_C)}_{seq} s'_A & \Rightarrow & (\textbf{Lemme 1 }) \\
\exists (w_{A_0}, w'_A) \cdot (BA_{g_{A_2}(t_C)}(w_{A_0}, w'_A) \wedge E_A(s_{A_0}, w_{A_0}) \wedge E_A(s'_A, w'_A)) & &
\end{array}
$$

Et d'après le théorème de la correction de la traduction, on a

$$
\begin{array}{rll}
s'_A \xrightarrow{f'(\sigma_C)} s_A \wedge E_A(s'_A, w'_A) & \Rightarrow & (\textbf{Théorème 2}) \\
\exists w_A \cdot (Pre_{g_{A_1}(f'(\sigma_C))}(w'_A) \wedge BA_{g_{A_2}(f'(\sigma_C))}(w'_A, w_A) \wedge E_A(s_A, w_A)) & &
\end{array}
$$

Puisque la machine M^{act}_A a été associée à A en suivant la méthode, on sait que

$$Pre_{g_1(f'(\sigma_C))}(w'_A) \wedge J_A(w'_A, v'_A) \Rightarrow G_{g_1(f'(\sigma_c))}(v'_A)$$

Or, on sait que d'après la définition de la méthode de raffinement que :

$$G_{g_1(f'(\sigma_c))}(v'_A) \wedge I_C(v'_A, v'_C) \Rightarrow G_{g_1(\sigma_C)_A}(v'_C)$$

	Raffinement classique	Raffinement étendu
Nombre d'obligation	93	62

tableau 2.1 – Intéret du raffinement étendu

On a donc

$$
\begin{array}{lll}
G_{g_1(\sigma_C)_C}(v'_C) \wedge G_{g_1(\sigma_C)_A}(v'_C) & \Rightarrow & \text{(Définition de } G_{C_O} \text{ et } G_{C_N}\text{)} \\
G_{g_1(\sigma_C)}(v'_C) & \Rightarrow & \text{(Faisabilité d'un événement)} \\
\exists v_C \cdot (BA_{g_1(\sigma_C)}(v'_C, v_C)) & &
\end{array}
$$

Donc

$$
\begin{array}{lll}
\exists (v_{C_0}, v_C) \cdot (BA_{g_{C_1}(t_C;\sigma_C)}(v_{C_0}, v_C)) & \Rightarrow & \textbf{(Définition 2)} \\
g_{C_1}(t_c;\sigma_c) \in \mathsf{traces}(M_C^{act}) & &
\end{array}
$$

□

Pour illustrer l'intérêt de cette méthode de raffinement, nous avons modélisé l'exemple en utilisant le raffinement classique présenté dans la section 2.3.1 et le raffinement permettant de réduire le nombre d'obligations de preuve présenté dans cette section. Les obligations de preuves générées sont indiquées dans la tableau 2.1. On remarque que l'utilisation du raffinement étendu permet de supprimer environ un tiers des obligations de preuve dans le cas de notre exemple.

Cependant, ce chiffre doit être nuancé. Dans le cas d'étude détaillé dans la section 5.1 par exemple, au cours de certains raffinement, des variables issues de variables abstraites sont utilisées dans des nouveaux événements, et dans ce cas, le gain en nombre d'obligations de preuve n'est pas significatif. En fait, le raffinement étendu défini dans cette section est véritablement efficace si les variables communes aux spécifications abstraite et concrète ne sont utilisées que dans des événements communs

aux spécifications abstraite et concrète. Par variables communes, nous entendons des variables liées par l'invariant de collage, et par événements communs, nous entendons événements liés par la fonction qui associe à chaque événement l'événement qu'il raffine (s'il existe).

2.4 Outillage de la méthode

Pour pouvoir être utilisée, notamment dans un contexte industriel, la méthode présentée ci-dessus doit être outillée. Dans cette section, nous présentons l'ensemble des outils nécessaires à la mise en place de la méthode, en présentant les fonctionnalités indispensables et les améliorations possibles pour avoir l'outil idéal. Nous donnons également un apperçu de l'état actuel de l'outil.

2.4.1 Un éditeur d'ASTD

Présentation de l'outil

Pour pouvoir spécifier en utilisant les ASTD, un outil d'édition d'ASTD est nécessaire. Cet outil doit pouvoir permettre de dessiner les diagrammes d'états et d'ajouter les opérateurs qui les encapsulent. Il doit également permettre de modifier les paramètres liés aux opérateurs (l'ensemble des transitions synchronisées pour un ASTD des ASTD en synchronisation par exemple). Il devrait également être capable vérifier si les ASTD sont syntaxiquement corrects

On pourrait également ajouter à cet outil un animateur, qui permettrait de visualiser l'exécution d'un ASTD. De plus, pour le moment aucun outil ne permet de montrer que le raffinement est correct. L'outil d'édition des ASTD pourrait permettre de faire cette vérification.

État de l'outil

Pour le moment, un éditeur a été commencé. Il a été réalisé à l'aide du *plug-in* grafiti d'éclipse. Il permet de spécifier quelques uns des opérateurs des ASTD, mais n'est pas complet, et la prise en main du *plug-in* n'est pas facilitée par la faible documentation. Les ASTD présentés dans cette thèse ont été dessinés à l'aide d'un logiciel

classique de dessin vectoriel, ce qui est suffisant pour les besoins de visualisation, mais ne permet pas une utilisation de la méthode dans le cadre de projets industriels.

2.4.2 Un traducteur d'ASTD

Présentation de l'outil

Des règles de traduction des ASTD vers le langage B ont été décrites dans la thèse de Milhau. L'objectif de cet outil est d'implémenter ces règles de traduction afin de rendre la traduction automatique.

De plus, les règles de traduction n'ont pas été prouvées. Il faut dont imaginer un moyen de valider la traduction des ASTD vers la méthode B. Pour valider une traduction, il existe deux méthodes classiques. La première consiste à prouver que le logiciel de traduction est correct, la deuxième consiste à confier à deux équipes la réalisation d'un outil de traduction et de vérifier ensuite que les traductions sont identiques.

État de l'outil

Pour le moment, il est possible de décrire les ASTD dans un langage textuel. La structure de ce langage est proche du langage XML. Un ASTD est une structure décrite sous forme de balises. Une balise correspond à chaque type d'ASTD. Dans le cadre d'un projet de réalisation d'un animateur d'ASTD, un parseur de ce langage a été défini. Ce parseur est écrit en CamL. Un outil utilisant ce parseur a été développé en CamL pour traduire les ASTD en langage B. Cet outil est disponnible en ligne [1]. De plus, la traduction des automates a également été réalisée en Coq, avec les preuves de complétude et de correction (*cf.* Chapitre 3). L'avantage de Coq est d'ailleurs que les spécifications peuvent être extraites en CamL, et donc pourraient réutiliser le parseur d'ASTD.

1. https://github.com/ThomasFay/ASTD2B

2.4.3 Atelier B et Rodin

Présentation des outils

Par rapport aux outils précédents, ces deux outils ne sont pas des outils que nous avons écrits, mais des outils existants.

L'atelier B [2] est l'outil de spécification de la méthode B. Il est développé par Clearsy. Il est nécessaire pour permettre de générer les obligations de preuves de cohérence de la méthode.

L'outil Rodin [3] est l'outil de spécification de la méthode Event-B. C'est un outil basé sur Eclipse. Il permet de générer les obligations de preuves liées aux spécifications Event-B et décrites dans la méthode. De plus, puisqu'il est basé sur éclipse il permet l'ajout de *plug-in*. Un *plug-in* a d'ailleurs été développé pour traduire les spécifications Event-B en langage B (il s'agit de remplacer les mots-clés et certains opérateurs qui diffèrent légèrement). Si le développement de l'outil de spécification utilisant graphiti pouvait être terminé, il serait intéressant d'utiliser le fait que les deux outils soient sous éclipse. On pourrait imaginer par exemple qu'on puisse extraire d'une spécification ASTD l'ensemble des étiquettes de transition qu'elle contient et qu'on puisse créer une machine contenant l'ensemble de ces étiquettes. On pourrait également imaginer qu'un *plug-in* permette de mettre en place la méthode de raffinement décrite dans la partie 2.3.2 et de générer les obligations de preuve associées.

2. www.atelierb.eu
3. www.event-b.org

Chapitre 3

Preuve de correction de la traduction des automates en langage B

3.1 Introduction

La méthode définie dans le chapitre 2 nécessite de traduire les ASTD dans le langage B. Dans sa thèse, Milhau [Mil11] a défini des règles de traduction systématique des ASTD vers le langage B, mais sans en fournir la preuve de correction. Sans preuve de ces règles de traduction, la confiance dans la méthode présentée diminue. Pour pouvoir utiliser notre méthode dans un cadre formel, il est important de valider la traduction des ASTD vers le langage B. Il s'agit ici de montrer que toute spécification ASTD traduite par notre outil est équivalente au sens de la bisimulation à la spécification en langage B obtenue après la traduction. Plusieurs méthodes permettent de valider une traduction d'un langage vers un autre.

La méthode la plus immédiate consiste à prouver la correction de l'outil de traduction. La preuve de l'outil de traduction consiste à montrer que la spécification obtenue après la traduction de n'importe quelle spécification valide est équivalente à

la spécification d'origine. C'est la méthode utilisée notamment dans le projet CompCert [Ler09]. Ce projet a pour objectif d'écrire un compilateur C réaliste et certifié (un compilateur C est en fait un traducteur du langage C vers le langage assembleur). Dans le projet CompCert, la preuve de l'outil de traduction a été réalisée en utilisant l'assistant de preuve Coq [Coq]. Le principal avantage de cette méthode est que l'utilisateur final peut utiliser l'outil de traduction en toute confiance. Cependant, à chaque changement d'un des langages ou du traducteur, l'outil de traduction doit être reprouvé intégralement.

Une deuxième méthode consiste à valider la traduction *a posteriori* [PSS98]. Dans cette méthode, l'outil de traduction est développé en utilisant des méthodes de développement classiques et seul un outil de validation de la traduction est prouvé formellement. Après chaque traduction, l'outil de validation vérifie que la traduction obtenue est correcte. L'avantage de cette méthode est que généralement, la preuve de l'outil de validation est beaucoup plus simple que la preuve de l'outil complet, ce qui réduit considérablement les coûts de développement de l'outil de traduction. De même, si les langages ou l'outil de traduction changent, seul l'outil de validation doit être corrigé. Cependant, pour faciliter le processus de traduction, il est important de trouver un moyen d'automatiser la preuve de validité. De plus, cette méthode déplace la découverte d'erreurs dans l'implémentation de la traduction au moment de l'utilisation de l'outil.

Enfin, une méthode non formelle consiste à faire écrire deux logiciels de traduction par deux équipes distinctes. À chaque traduction, on vérifie alors que les deux outils de traduction renvoient les mêmes résultats. Cette méthode permet de garantir plutôt facilement un niveau de confiance assez élevé. Cependant, pour pouvoir facilement comparer les spécifications obtenues après la traduction, les deux équipes doivent préalablement s'entendre sur des conventions d'écriture du langage cible. Et comme la méthode précédente, une traduction d'une spécification valide peut échouer et nécessiter des corrections de l'outil de traduction. C'est la méthode utilisée dans le projet METEOR d'automatisation de ligne 14 du métro parisien pour traduire les spécifications B en Ada [Dol06].

Dans le cadre de la méthode développée dans le chapitre précédent, nous souhaitons traduire les ASTD dans le langage B. La sémantique de ces deux langages

étant fixée, les règles de traduction ne sont pas amenées à changer. Il serait donc intéressant de développer un outil de traduction certifié. La certification d'un outil de traduction consiste à montrer que quelle que soit une spécification ASTD valide donnée en entrée du traducteur, cette spécification peut être traduite en langage B et la spécification obtenue en sortie est équivalente à la spécification d'entrée au sens de la bisimulation. La preuve de bisimulation avec un assistant de preuve est un travail long. Pour des raisons de temps, seule une preuve partielle de l'outil de traduction a été réalisée. Les ASTD sont des automates étendus par des opérateurs des algèbres de processus. Nous présentons la preuve que pour tout automate, si l'outil de traduction peut calculer une traduction, alors l'automate simule la spécification B obtenue par l'outil de traduction (ce qui correspond à la preuve de correction de la traduction). Les autres opérateurs des ASTD ne sont pas considérés. Pour réaliser cette preuve, les règles de traduction décrites dans la thèse de Milhau [Mil11] ont été traduites dans le langage de programmation de l'assistant de preuve Coq, qui a ensuite été utilisé pour effectuer la preuve de simulation. Un travail de preuve similaire aurait pu être réalisé avec l'outil Isabelle/HOL. Ici Coq a été choisi, notamment pour sa proximité avec le langage OCaml, dans lequel un analyseur syntaxique et un outil de traduction ont déjà été développés.

La suite du chapitre est organisée de la manière suivante. La section 3.3 présente la formalisation du problème. Nous y présentons la sémantique des automates des ASTD, la sémantique du sous-ensemble du langage B nécessaire pour la traduction, les règles de traduction des automates et la formalisation du théorème à prouver. Pour chaque point, nous présentons quelques intuitions sur la manière de traduire ces formalisations en Coq. Dans la section 3.4, nous présentons le schéma de la preuve. Mais auparavant, la section 3.2 présente l'assistant de preuve Coq et les notions importantes à la compréhension de ce chapitre.

3.2 Présentation de Coq

Coq est un assistant de preuve basé sur le calcul des constructions, un λ-calcul typé d'ordre supérieur basé sur la correspondance de Curry-Howard. Les termes du langage sont des fonctions typées et il est possible d'écrire dans le même langage un programme

et des preuves de propriété sur ce programme. Le langage de programmation de Coq est un langage fonctionnel proche du langage OCaml. Coq permet d'ailleurs d'extraire ses programmes vers OCaml (et vers d'autres langages fonctionnels).

Dans la pratique, le langage Coq permet de définir des types inductifs, comme les types inductifs de OCaml par exemple. Ces types inductifs sont introduits par le mot clé `Inductive`. Dans la pratique, nous distinguons les types inductifs, qui dans notre traduction permettent de définir les structures du langage (par exemple le type des automates) et les prédicats définis inductivement, appelés dans la suite prédicats inductifs. Les prédicats inductifs permettent de décrire des prédicats à partir d'un ensemble de règles. Chaque règle est écrite comme une implication : un ensemble de prémisses permettent de déduire un prédicat sur des termes du langage, à la manière des règles d'inférence. Ce sont donc les prédicats inductifs qui permettront de traduire les règles d'inférence des ASTD. Il faut noter qu'en Coq il n'y a pas de réelle différence de nature entre ces les types inductifs et les prédicats inductifs. La différence réside dans le fait que les prédicats inductifs sont ignorés lors de l'extraction des preuves vers un langage de programmation.

Le langage Coq permet également de définir des fonctions récursives. Ces fonctions sont introduites par le mot clé `Fixpoint`. Elles ont deux caractéristiques importantes : (1) toute fonction doit terminer : l'appel récursif doit porter sur un sous-terme structurel de l'argument initial (récursion structurelle) (2) toute fonction doit être totale : la définition d'une fonction récursive se fait généralement au moyen d'une recherche de motif (*pattern matching*) qui doit être exhaustif.

Enfin, le langage Coq dispose de plusieurs mots clés pour définir des théorèmes (`Theorem`, `Lemma` ou encore `Corollary`) qui doivent ensuite être prouvés. L'écriture de ces théorèmes est souvent technique et peu compréhensible. Dans ce chapitre, nous avons privilégié une écriture moins formelle qui permet de mieux comprendre les théorèmes prouvés.

3.3 Formalisation

L'objectif de cette section est de présenter la formalisation du problème ainsi qu'une description de la manière dont le problème a été traduit dans le langage Coq,

en décrivant les choix réalisés dans l'implémentation qui impactent les preuves et en explicitant certaines définitions implicites. Dans la section 3.3.1, nous présentons la sémantique formelle des automates dans le langage ASTD. Puisque les ASTD ont une sémantique opérationnelle, nous souhaitons avoir une sémantique opérationnelle du langage B. Cette sémantique est décrite dans la section 3.3.2. Les règles de traduction et une description de leur implémentation en Coq sont décrites dans la section 3.3.3. Enfin, le théorème que nous souhaitons prouver est décrit dans la section 3.3.4.

3.3.1 Sémantique opérationnelle des automates

Le principe des automates dans le langage ASTD est basé sur le principe des diagrammes d'états hiérarchiques de Harel [Har87]. Un automate est constitué d'un ensemble d'états et d'un ensemble de transitions entre ces états. Le fait que les diagrammes d'états soient hiérarchiques signifie que chaque état de l'automate peut être lui-même un automate. Dans les automates, les transitions peuvent être de trois types : (1) les transitions locales entre deux états d'un automate, (2) les transitions qui partent du sous-état d'un automate et (3) les transitions qui arrivent à un sous-état d'un automate. Dans ce dernier cas, la transition peut arriver à un état dit historique, c'est-à-dire que l'état après la transition est le dernier état visité. Les automates ont un état initial, marqué par un chevron ($>$), et un état final marqué par un double cercle.

Le type ASTD est le type des ASTD. L'état courant d'un ASTD est décrit par le type State. Enfin, l'ensemble des événements acceptés par le système est décrit par l'ensemble Event. La sémantique est décrite par un système de transitions étiquetées, soit un sous-ensemble de $\mathsf{State} \times \mathsf{Event} \times \mathsf{State}$. Une transition du système est un élément de cet ensemble. Elle est notée $s \xrightarrow{\sigma} s'$, ce qui signifie que dans l'état s, on peut exécuter l'événement σ et qu'après avoir exécuté cet événement, le système est dans l'état s'. Enfin, une fonction appelée $\mathit{final} \in \mathsf{State} \rightarrow \mathsf{Boolean}$ permet de dire si l'état d'un ASTD est final et une fonction $\mathit{init} \in \mathsf{ASTD} \rightarrow \mathsf{State}$ retourne l'unique état initial d'un ASTD.

Dans ce chapitre, nous nous intéressons uniquement à la traduction des automates, les autres opérateurs des ASTD ne sont pas considérés. Un ASTD peut être de deux

types : soit il est un état élémentaire, noté ASTDElem, soit il est un automate, noté Automaton.

État élémentaire

Le type $\mathsf{ASTDElem} \triangleq \langle \mathsf{elem} \rangle$ est le type des états élémentaires d'un automate. L'état courant d'un état élémentaire d'un automate est décrit par $\langle \mathsf{elem}_\circ \rangle \in \mathsf{State}$. Les fonctions *init* et *final* sont définies de la manière suivante :

$$\mathit{init}(\langle \mathsf{elem} \rangle) \triangleq (\mathsf{elem}_\circ)$$
$$\mathit{final}(\mathsf{elem}_\circ) \triangleq \mathbf{true}$$

Automates

Le type $\mathsf{Automaton} \triangleq \langle \mathsf{aut}, \mathit{name}, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$ est le type des automates, où :

- $\mathit{name} \in N$ est le nom de l'ASTD.
- $\Sigma \subset \mathsf{Event}$ est l'ensemble des événements de cet ASTD.
- $N \subset \mathsf{Name} - \{\mathsf{H}, \mathsf{H^*}\}$ est l'ensemble des noms des états de l'automate.
- $\nu \in N \rightarrow \mathsf{ASTD}$ est une fonction qui relie chaque nom d'état à l'ASTD qui lui correspond.
- $\delta \subset \langle \eta, \sigma, \phi, \mathit{final?} \rangle$ est l'ensemble des transitions, dans lequel :
  - η correspond à la flèche. Il y a trois types de flèches : les flèches locales, notées $\langle \mathsf{loc}, n_1, n_2 \rangle$, qui partent de l'état nommé n_1 vers l'état nommé n_2, les flèches qui partent du sous-état nommé $n_{1_\flat}$ d'un automate nommé n_1 vers l'état nommé n_2, notées $\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$, et les flèches qui, à partir d'un état nommé n_1 vont vers le sous-état nommé $n_{2_\flat}$ d'un automate nommé n_2 , notées $\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$. Dans ce dernier cas, la flèche peut arriver à un état historique H ou historique profond H*. Dans le cas de l'état historique, la transition arrive alors à l'état initial du dernier état visité et dans l'état historique profond, elle arrive au dernier état visité.

  - $\sigma \in \Sigma$ est l'événement.
  - $\phi \in \mathsf{Predicate}$ est la garde éventuelle de la transition. Dans le rapport technique décrivant les ASTD, il n'est pas précisé quels types de prédicat sont acceptés. Puisque nous traduisons les ASTD en B, nous supposons que les gardes acceptées sont les gardes acceptées par le langage B.
  - *final*? $\in \mathsf{Boolean}$ est un booléen qui est vrai si la transition est de type finale (c'est-à-dire qu'elle ne peut être exécutée que si l'ASTD de départ est dans un état final).
- $SF \subseteq \mathsf{Name}$ est l'ensemble des noms des états finaux non profonds.
- $DF \subseteq \mathsf{Name}$ est l'ensemble des noms des états finaux profonds.
- $n_0 \in N$ est le nom de l'état initial.

Le type $\mathsf{State} \triangleq \langle \mathsf{aut}_\circ, n, h, s \rangle$ est le type des états courants d'un automate, où :

- $n \in \mathsf{Name}$ est le nom de l'état courant
- $h \in \mathsf{Name} \rightarrow \mathsf{State}$ est une fonction qui sauvegarde l'état historique associé à chaque état de l'ASTD.
- $s \in \mathsf{State}$ est l'état courant de l'ASTD $\nu(n)$

Les fonctions *init* et *final* sont définies de la manière suivante pour les automates :

$$\begin{aligned} \mathit{init}(\mathsf{aut}, \mathit{name}, \Sigma, N, \nu, \delta, SF, DF, n_0) &\triangleq (\mathsf{aut}_\circ, n_0, h_{\mathit{init}}, \mathit{init}(\nu(n_0))) \\ \text{avec } h_{\mathit{init}} &\triangleq \{n \mapsto \mathit{init}(\nu(n)) | n \in N\} \\ \mathit{final}((\mathsf{aut}_\circ, n, h, s)) &\triangleq (n \in DF \wedge \mathit{final}(s)) \vee (n \in SF) \end{aligned}$$

La sémantique opérationnelle des automates contient six règles, qui sont données en annexe, à la section A.1. La règle aut_1 décrit comment exécuter une transition locale. Les règles aut_2, aut_3 et aut_4 décrivent comment exécuter une transition allant vers un sous-état : aut_2 décrit la sémantique pour exécuter une transition allant vers un sous-état classique, aut_3 décrit le cas où la transition va vers un état historique H et la règle aut_3 décrit le cas où la transition va vers un état historique profond H^*. La règle aut_5 décrit la sémantique pour exécuter une transition partant du sous-état d'un automate. Enfin, la transition aut_6 décrit le cas inductif, c'est-à-dire le cas d'une transition qui peut être exécutée dans le sous-état d'un ASTD.

Implémentation en Coq

L'implémentation du langage ASTD décrit dans la section précédente nécessite de décrire le langage des automates des ASTD, les structures modélisant l'état courant d'un automate, et les règles d'inférence. Le langage ASTD et les états sont décrits en utilisant les types inductifs de Coq, les règles d'inférence sont décrites en utilisant les prédicats inductifs.

Tout d'abord, un type transition permet de décrire l'ensemble des transitions. Il est défini de la manière suivante :

```
Inductive transition :=
|Local : from_state → to_state → transition_label → guard → bool → transition
|From_sub : from_state → to_state → through_state → transition_label → guard →
bool → transition
|To_sub : from_state → to_state → through_state → transition_label → guard →
bool → transition.
```

Cette définition doit être lue de la manière suivante : le type transition peut être construit à partir de trois constructeurs (Local, From_sub et To_sub). Chaque constructeur est une fonction qui permet de construire une transition à partir de paramètres. Le constructeur Local par exemple permet de construire une transition à partir de cinq paramètres : un identifiant correspondant à l'état de départ, un identifiant correspondant à l'état d'arrivée, une étiquette de transition, une garde et une valeur booléenne indiquant si la transition est finale. Dans notre cas, les états sont identifiés par une chaîne de caractère. Ici, from_state et to_state correspondent donc à un renommage du type String de Coq.

De même, le type des automates des ASTD est défini de la manière suivante :

```
Inductive astd :=
|Elem : astd_name → astd
|Automata : astd_name → list astd → list transition → list astd_name → list astd_name
→ astd_name → astd.
```

Le constructeur Elem permet de construire un état élémentaire à partir d'un nom d'ASTD qui correspond au nom donné à cet état. Le constructeur Automata permet de construire un automate à partir d'un nom d'ASTD, d'une liste d'ASTD qui corres-

pondent à l'ensemble des états de l'automate, d'une liste de transitions, de deux listes de noms d'ASTD décrivant les noms des états finaux non profonds et des états finaux profonds et du nom de l'état initial.

L'ensemble des états courants des ASTD est décrit par le type inductif suivant :

```
Inductive state :=
|Elem_state : state
|Automata_state : astd_name → list (astd_name × state) → state → state.
```

Le constructeur Elem_state permet de construire un état élémentaire. Le constructeur Automata_state permet de construire un état d'automate, à partir du nom de l'état dans lequel se trouve l'automate, d'une liste de couples représentant l'état historique et d'un état qui représente le sous-état courant.

Les fonctions *init* et *final* sont encodées grâce à des prédicats inductifs de Coq, qui sont définis grâce à un ensemble de constructeurs. Ces constructeurs permettent de définir un ensemble de règles pour construire des éléments qui satisfont le prédicats (à la manière des règles d'inférence). La fonction *final* par exemple est encodée par un prédicat de type suivant :

```
Inductive isStateFinal : state → astd → Prop
```

Ce prédicat inductif est définie de telle manière que pour tout état s de type state et tout ASTD A de type astd, isStateFinal s A est vrai ssi $final(s)$ est vrai dans l'ASTD A. On remarquera d'ailleurs que la définition en Coq du prédicat isStateFinal rend la définition de la fonction *final* : en effet, dans le rapport technique définissant les ASTD, la fonction n'est pas paramétrée par l'ASTD, tandis que la formalisation en Coq impose ce paramétrage. Ce prédicat est défini à partir de trois règles, qui implémentent les règles d'inférence suivantes :

$$elem\ \frac{}{final_{\langle \mathsf{elem} \rangle}(\mathsf{elem}_\circ)} \qquad sf\ \frac{n_1 \in SF}{final_A(\langle \mathsf{aut}_\circ, n_1, h, s \rangle)} \qquad df\ \frac{n_1 \in DF \qquad final_{\nu(n_1)}(s)}{final_A(\langle \mathsf{aut}_\circ, n_1, h, s \rangle)}$$

avec $A \triangleq \langle \mathsf{aut}, , name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$. Dans ces règles d'inférence, la fonction *final* a été explicitée : l'ASTD en indice est l'ASTD dans lequel l'état est final.

À titre d'exemple, la traduction en Coq du prédicat inductif est donné ci-dessous :

```
Inductive isStateFinal : state → astd → Prop :=
|IsStateFinal_elem :
```

```
  ∀ (s : string),isStateFinal Elem_state (Elem s)
|IsStateFinal_shallowFinal :
  ∀ (stateName astdName initName : string) (transitions : list transition)
        (states : list astd) (subState : state)
        (shallowFinal deepFinal : list string) (history : list astd_name×state),
    In stateName shallowFinal →
    isStateFinal
      (Automata_state stateName history subState)
      (Automata astdName states transitions shallowFinal deepFinal initName)
|IsStateFinal_deepFinal :
  ∀ (stateName astdName initName : string) (subAstd : astd)
        (transitions : list transition) (states : list astd) (subState : state)
        (shallowFinal deepFinal : list string) (history : list astd_name×state),
    In stateName deepFinal →
    isStateFinal subState subAstd →
    isAstdFromList subAstd stateName states →
    isStateFinal
      (Automata_state stateName history subState)
      (Automata astdName states transitions shallowFinal deepFinal initName).
```

Le constructeur IsStateFinal_elem permet de traduire la règle d'inférence *elem*. Le constructeur IsStateFinal_shallowFinal permet de traduire la règle d'inférence *sf*. La fonction In est une fonction de la bibliothèque des listes de Coq et qui permet de construire le prédicat qui vérifie si un élément est dans une liste. Enfin, le constructeur IsStateFinal_deepFinal permet de traduire la règle d'inférence *df*. Le prédicat isAstdFromList est un prédicat qui vérifie que l'ASTD *subAstd* est bien l'ASTD dont le nom est *subState* dans la liste des états *states*. Elle correspond à une implémentation de la fonction α utilisée dans la règle *df*.

De la même manière, la fonction *init* est codée par le prédicat inductif suivant :

```
Inductive isInitialState : state → astd → Prop
```

Enfin, un prédicat inductif astd_step permet d'encoder les règles d'inférence de la sémantique des automates. Ce prédicat est défini avec le type suivant :

`Inductive` astd_step : state → astd → transition_label → state → `Prop`

Soient deux états s et $s' \in$ state, un ASTD $A \in$ astd et une étiquette de transition $l \in$ transition_label, le prédicat inductif (astd_step s A l s') est vrai ssi $s \xrightarrow{l}_A s'$, c'est-à-dire qu'il est possible de passer d'un état s à l'état à l'état s' par la transition l dans l'automate A. Ce prédicat inductif est défini par six constructeur, chaque constructeur correspondant à une des règles d'inférence donnée en annexe A.1.

3.3.2 Sémantique opérationnelle pour un sous-ensemble du langage B

Seul un sous-ensemble du langage B est nécessaire pour traduire les automates des ASTD. Dans ce sous-ensemble, une machine est représentée par un ensemble d'opérations. Ces opérations ont un label, une précondition et une postcondition. Le label appartient à l'ensemble Event_B, l'ensemble des événements possibles dans le langage B. La précondition est un prédicat B et la postcondition est une substitution B.

Dans le sous-ensemble considéré du langage B, les prédicats sont soit des égalités, soit des prédicats d'appartenance à un ensemble, soit la conjonction ou la disjonction de deux prédicats, soit une garde. Puisque le langage ASTD autorise des gardes écrites en langage B, la garde a les mêmes valeurs de vérité en ASTD et en langage B. Il n'est donc pas nécessaire de décrire la manière de les évaluer pour prouver l'équivalence sémantique. Les substitutions du sous-ensemble du langage B nécessaire à la traduction des automates sont soit des affectations, soit une substitution de type **SELECT**, soit une substitution parallèle.

Pour la traduction des ASTD, les variables nécessaires sont des variables simples qui contiennent les états des ASTD. Ces variables sont dans l'ensemble Variable des variables du langage B. Leur valeur est dans l'ensemble Constant des littéraux du langage B. Un état d'une machine B $\mathsf{State}_B \in$ Variable $\rightarrow$ Constant est donc une fonction qui associe à chaque variable une valeur. La sémantique opérationnelle pour ce sous-ensemble du langage B est décrite comme un système de transitions étiquetées qui est un sous-ensemble de $\mathsf{State}_B \times \mathsf{Event}_B \times \mathsf{State}_B$.

Prédicats

L'ensemble $\mathsf{Predicate}_B$ des prédicats du langage B est décrit inductivement par les propositions suivantes :

— $(x =_B c)$ avec $x \in \mathsf{Variable}$ et $c \in \mathsf{Constant}$ est un prédicat du langage B (les règles de traduction n'autorisent que l'affectation d'une constante à une variable)

— $(x \in_B E)$ avec $x \in \mathsf{Variable}$ et $E \subset \mathsf{Constant}$ est un prédicat du langage B

— $p_1 \wedge p_2$ avec $p_1 \in \mathsf{Predicate}_B$ et $p_2 \in \mathsf{Predicate}_B$ est un prédicat du langage B

— $p_1 \vee p_2$ et $p_1 \in \mathsf{Predicate}_B$ et $p_2 \in \mathsf{Predicate}_B$ est un prédicat du langage B

— g avec $g \in \mathsf{Predicate}$ est un prédicat du langage B

La sémantique de ce langage des prédicats est décrite par un système de déduction qui est inclus dans $\mathsf{State}_B \times \mathsf{Predicate}_B$. Soit $v \in \mathsf{State}_B$ et $p \in \mathsf{Predicate}_B$, $v \vdash p$ signifie que l'état v des variables de la machine B permet de satisfaire le prédicat p. Les règles de déduction associées à ce système sont les suivantes :

$$pred_{eq}\ \frac{v(x) = c}{v \vdash (x =_B c)} \qquad pred_{in}\ \frac{v(x) \in E}{v \vdash (x \in_B E)}$$

$$pred_{and}\ \frac{v \vdash p_1 \qquad v \vdash p2}{v \vdash p_1 \wedge p_2} \qquad pred_{g}\ \frac{d(g)}{v \vdash g}$$

$$pred_{or_r}\ \frac{v \vdash p_2}{v \vdash p_1 \vee p_2} \qquad pred_{or_l}\ \frac{v \vdash p_1}{v \vdash p_1 \vee p_2}$$

La règle $pred_g$ est la règle pour savoir si un prédicat de type garde est vrai. Les gardes des ASTD sont écrites dans l'ensemble des prédicats du langage B. Nous supposons donc que nous avons un oracle $d \in \mathsf{Predicate} \rightarrow \mathsf{Boolean}$ qui est capable de décider si une garde est vraie. La garde sera en principe vraie dans les mêmes états pour les ASTD et pour le langage B.

Substitutions

Les substitutions nécessaires pour traduire les automates dans le langage B sont les substitutions définies inductivement par les propositions suivantes

— $x := c$ avec $x \in \mathsf{Variable}$ et $c \in \mathsf{Constant}$ est une substitution. Elle permet d'associer la valeur c à la variable x.

— $s_1||s_2$ avec $s_1 \in \mathsf{Substitution}$ et $s_2 \in \mathsf{Substitution}$ est une substitution. Elle signifie que les substitutions s_1 et s_2 sont exécutées en parallèle.

— $\mathsf{Select}\ l$ avec $l \in \mathit{list}\ (\mathsf{Predicate}_B \times \mathsf{Substitution})$ est une substitution. Cette définition utilise la définition classique des listes des langages fonctionnels. Pour tout ensemble E, $\mathit{list}\ E$ est l'ensemble défini inductivement de la manière suivante :

  — $[]$, la liste vide, est une liste

  — $\forall h \in E$ et $q \in \mathit{list}\ E$, $h :: q$ est une liste dont le premier élément est h et dont le reste est q.

— Skip est la substitution qui ne modifie pas l'état des variables

La sémantique de ces substitutions est décrite comme un système de transitions étiquetées, qui est un sous-ensemble de $\mathsf{State}_B \times \mathsf{Substitution} \times \mathsf{State}_B$. Pour tout état v, tout état v' et pour toute substitution s, on dit $v \xrightarrow{s} v'$ si la substitution s permet de passer de l'état v à l'état v'. Dans le sous-ensemble du langage B décrit ci-dessus, la seule possibilité pour modifier l'état d'une variable est d'affecter une constante à cette variable. De plus, le langage B interdit de modifier l'état d'une même variable dans deux substitutions en parallèle. Sous ces deux conditions ((1)seule l'affectation d'une constante à une variable est permise ET (2) il est impossible de modifier une même variable dans deux branches d'une substitution parallèle), l'exécution de deux substitutions en parallèle est équivalente à l'exécution de ces substitutions dans n'importe quel ordre arbitraire. Pour décrire la sémantique, puisque ces deux conditions sont vérifiées dans le sous-ensemble du langage B nécessaire à la traduction des ASTD, nous avons choisi arbitrairement un ordre pour exécuter des substitutions en parallèle. Le système de transitions est décrit par les règles d'induction suivantes :

$$sub_{aff}\ \frac{}{v \xrightarrow{x:=y} v \lhd \{x \mapsto y\}} \qquad sub_{par}\ \frac{v \xrightarrow{sub_1} v'' \qquad v'' \xrightarrow{sub_2} v'}{v \xrightarrow{sub_1||sub_2} v'}$$

$$sub_{sel_1}\ \frac{v \vdash g \qquad v \xrightarrow{sub} v'}{v \xrightarrow{\mathsf{Select}((g,sub)::l)} v''} \qquad sub_{sel_2}\ \frac{v \xrightarrow{\mathsf{Select}(l)} v''}{v \xrightarrow{\mathsf{Select}((g,sub)::l)} v''}$$

$$skip\ \frac{}{v \xrightarrow{\mathsf{Skip}} v}$$

Nous rappelons que ces règles **ne sont pas valables** pour l'ensemble du langage B, notamment à cause de la règle sub_{par}. Mais elles conviennent pour décrire la séman-

tique du sous-ensemble du langage B nécessaire pour la traduction des automates. Nous supposons également que la machine issue de la traduction est acceptée par la vérification de type de l'atelier B. Dans la règle sub_{aff} complète, il faudrait par exemple vérifier que la variable x existe dans l'état v et que la constante y est bien du type de x, mais ces vérifications sont faites par le vérificateur de type de l'atelier B. Dans ces conditions, l'affectation d'une constante à une variable n'a pas besoin de prémisse.

Machine B

Une machine est un ensemble d'opérations. Ces opérations sont décrites par un label, un prédicat appelé précondition et une substitution appelée postcondition ($M \subset$ Event $\times$ Predicate $\times$ Substitution). L'exécution d'une opération dans une machine B est décrite par la règle suivante :

$$step\ \frac{(\sigma, pre, sub) \in M \quad s \vdash pre \quad s \xrightarrow{sub} s'}{s \xrightarrow{\sigma} s'}$$

Implémentation en Coq

Comme pour les ASTD, le langage B est traduit en utilisant des types inductifs et sa sémantique est traduite en utilisant des prédicats inductifs. Les prédicats du langage B sont traduits par le type inductif suivant :

```
Inductive predicate :=
|Equality : variable → constant → predicate
|And : predicate → predicate → predicate
|Or : predicate → predicate → predicate
|TruePredicate : predicate
|FalsePredicate : predicate
|IsIn : variable → list constant → predicate
|Guard : guard → predicate
```

Les substitutions sont traduites par le type inductif suivant :

```
Inductive substitution :=
|Select : list (predicate × substitution) → substitution
```

|Parallel : substitution → substitution → substitution
|Affectation : variable → constant → substitution
|SkipSubstitution : substitution.

Une opération est décrite par une étiquette, une précondition et une postcondition, grâce au type inductif suivant :

```
Inductive operation :=
|Operation : operation_label → predicate → substitution → operation.
```

Les états de la machine B sont décrits comme des fonctions associant une valeur à chaque variable. En Coq, on pourrait encoder cette fonction par une liste de couples. Mais pour faciliter la mise en relation des états de la machine B et des états des ASTD, une hiérarchie a été introduite dans l'écriture de cette fonction. L'idée de cette hiérarchie est de calquer la structure de l'état d'une machine B sur la structure de l'état des automates (voir section 3.3.4 pour plus de détails). Les états de la machine B sont décrits par le type inductif suivant :

```
Inductive bState :=
|Elem_B_state : bState
|Automata_B_state : variable → constant → list bState → bState.
```

Les règles d'inférence de la sémantique du langage B sont ensuite traduites en utilisant des prédicats inductifs. Un prédicat inductif appelé, predicateIsTrue, permet de traduire la sémantique des prédicats B, un prédicat inductif, appelé substitution_step, permet de décrire la sémantique des substitutions. Enfin, un prédicat inductif, appelé b_step, permet de décrire la sémantique des machines. Ici, une machine est décrite comme une liste d'opérations. Ces prédicats, qui permettent d'implémenter en Coq les règles d'inférences de la sémantique de B décrites plus haut dans cette section, ont les types suivants :

```
Inductive predicateIsTrue : predicate → bState   → Prop
Inductive substitution_step : bState → substitution → bState → Prop
Inductive b_step : bState → machine → operation_label → bState → Prop
```

3.3.3 Règles de traduction

Les règles de traduction des ASTD vers le langage B ont été données dans [Mil11]. Ce sont ces règles qui sont rappelées dans cette section. Le principe de la traduction est le suivant : l'état d'un automate dont le nom est *name* est représenté par une variable B *State_name*. Une opération B est créée pour chaque étiquette de transition. L'ensemble des étiquettes de transition d'un automate est alors l'union de l'ensemble des étiquettes des transitions de l'automate considéré et de l'union des étiquettes de transition de tous les sous-états de l'automate considéré. Dans un automate, plusieurs transitions peuvent avoir la même étiquette. L'opération B correspondant à cette étiquette spécifie alors toutes les transitions possibles pour cette étiquette. La traduction des automates est inductive. Les automates correspondant aux différents états de l'automate considéré sont traduits. Le résultat donne alors les manières d'effectuer une transition dans un sous-automate de l'automate considéré. Ensuite, une précondition et une postcondition sont calculées pour chaque transition de l'automate considéré. Une fois que tous les sous-automates et toutes les transitions sont traduites, les préconditions et postconditions sont rassemblées par étiquette.

Calcul de la précondition et postcondition associée à chaque transition

Soit $t = \langle \eta, \sigma, \phi, \mathit{final}? \rangle \in \delta$ une transition. Pour cette transition une précondition Pre_t et une postcondition $Post_t$ sont calculées. La précondition est un prédicat B qui doit être vrai dans les états B correspondant aux états dans lesquels la transition peut être exécutée. La postcondition est une substitution qui met à jour l'état des variables B de telle manière que l'état après la transition correspond à l'état d'arrivée de la transition ASTD. Le calcul de la précondition et de la postcondition est détaillé dans le tableau 3.1.

Dans ce tableau, la fonction *final* est une fonction qui associe à chaque automate le prédicat B qui est vrai lorsque l'état des variables B correspond à un état final de l'ASTD. La fonction *initAllVar* est une fonction qui associe à chaque automate une substitution B telle que l'état des variables B après l'exécution de cette substitution correspond à l'état initial de l'automate. Certaines notations (ν et N par exemple) font

η	Pre_t		$Post_t$
	P_{t_η}	$final?$	
$\langle \mathsf{loc}, n_1, n_2 \rangle$	p_{all}	$\wedge\ final(\nu(n_1))$	$s_{all} \mid\mid s_{init_{n_2}}$
$\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$	p_{all}	$\wedge\ final(\nu(n_1))$	$s_{all} \mid\mid s_{toSub} \mid\mid s_{init_{n_{2_\flat}}}$
$\langle \mathsf{tsub}, n_1, n_2, \mathsf{H} \rangle$	p_{all}	$\wedge\ final(\nu(n_1))$	$s_{all} \mid\mid s_{init_h}$
$\langle \mathsf{tsub}, n_1, n_2, \mathsf{H}^\star \rangle$	p_{all}	$\wedge\ final(\nu(n_1))$	s_{all}
$\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$	$p_{all} \wedge p_{fromSub}$	$\wedge\ final(\nu_{\nu(n_1)}(n_{1_\flat}))$	$s_{all} \mid\mid s_{init_{n_2}}$

Définition des prédicats

$$p_{all} \triangleq State_name =_B n_1$$
$$p_{fromSub} \triangleq State_n_1 =_B n_{1_\flat}$$

Définition des substitutions :

$$s_{all} \triangleq State_name := n_2$$
$$s_{init_{n_2}} \triangleq initAllVar(\nu(n_2))$$
$$s_{toSub} \triangleq State_n_2 := n_{2_\flat}$$
$$s_{init_{n_{2_\flat}}} \triangleq initAllVar(\nu_{\nu(n_2)}(n_{2_\flat}))$$
$$s_{init_h} \triangleq ||_{n \in N_{\nu(n_2)}} initAllVar(\nu(n))$$

tableau 3.1 – Traduction d'une transition

référence à des éléments de la définition des automates. Lorsque rien n'est précisé, cela signifie que l'attribut est celui de l'automate auquel appartient la transition. Lorsqu'un ASTD est mis en indice, cela signifie qu'on considère l'attribut de l'ASTD passé en indice : $\nu_{\nu(n_1)}$ désigne par exemple la fonction ν (qui associe un ASTD à chaque nom d'état) de l'ASTD $\nu(n_1)$ (qui est l'ASTD nommé n_1 dans l'ASTD courant).

Traduction complète

La précondition et la postcondition calculées pour chaque transition sont ensuite combinées avec la traduction inductive pour construire une unique opération pour chaque label de transition. Soit $A \triangleq \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$ l'ASTD que

$\sigma =$
PRE
typing conditions of the operation $\wedge$
$(\bigvee_{t \in T_\sigma} Pre_t \ \vee \bigvee_{n \in N} \bigvee_{op_\sigma \in M_{n_\sigma}} (State_name = n \wedge preOf(op_\sigma)))$
THEN
SELECT
WHEN Pre_t **THEN** $Post_t$ for each $t \in T_\sigma$
WHEN $State_name = n \wedge preOf(op_\sigma)$
THEN $thenOf(op_\sigma)$ ||
$State_name := n$ for each $op_\sigma \in M_{n_\sigma}$
for each $n \in N$
END
END
For $n \in N$, $M_{n_\sigma} \triangleq \{op \in M_n | labelOf(op) = \sigma\}$

figure 3.1 – Forme générale d'une opération traduisant un automate

nous souhaitons traduire. Chaque ASTD $\nu(n), n \in N$ est traduit en une machine M_n en utilisant des appels inductifs. Dans chacune des traductions, on appelle M_{n_σ} l'ensemble des opérations dont le label est σ dans M_n. Soient enfin les fonctions *labelOf* qui associe son label à chaque opération, *preOf* qui associe à chaque opération sa précondition et *postOf* qui associe chaque opération à sa postcondition. Enfin, T_σ est défini comme l'ensemble des transitions dont le label est σ dans δ. Pour chaque label σ, l'opération qui est associée à ce label est l'opération donnée à la figure 3.1. Il faut noter que dans le cas où les préconditions et postconditions viennent de la traduction d'un sous-automate (deuxième type de **WHEN** sur la figure 3.1), l'affectation $State_name := n$ pourrait être retirée, mais elle est nécessaire pour permettre la traduction de la fermeture de Kleene. Puisque la preuve de traduction présentée ici doit être ensuite complétée par une traduction de l'ensemble du langage des ASTD, cette affectation doit être maintenue.

Implémentation en Coq

Les fonctions de traduction sont implémentées par des fonctions récursives du langage Coq. L'idée de la traduction d'un automate est de traduire l'ensemble des sous-automates de l'ASTD courant puis l'ensemble des transitions et de rassembler ces deux traductions dans une liste d'opérations qui ne contient qu'une seule opération par étiquette d'opération. La fonction de traduction principale est une fonction mutuellement récursive dont l'expression (simplifiée pour des raisons de compréhension) est la suivante :

```
Fixpoint translateAstd a :=
  match a with
    |Elem _ ⇒ []
    |(Automata name states trans sFinal dFinal init) ⇒
     let subMachine := translateStateList states in
     let transitionMachine := translateAllTransitions trans in
     mergeOpList subMachine transitionMachine
  end
with translateStateList states :=
  match states with
    |[] ⇒ []
    |head :: tail ⇒
     let headMachine := modifiyPrePost (translateAstd head) in
     let tailMachine := translateStateList tail in
     mergeOpList heMachine taMachine
  end.
```

Pour faciliter la lecture, l'expression de la fonction de traduction donnée ci-dessus est une simplification de la véritable fonction de traduction. En effet, en Coq, toutes les fonctions doivent être totales. Pour simuler le mécanisme des exceptions, on utilise par exemple un type "option", qui est un type paramétré qui permet de gérer les exceptions. Un objet de type "option" est soit un objet du type paramétré (Some x), soit aucun objet (None). Ici, le type "option" a volontairement été omis afin de faciliter la lecture. De même, dans l'appel des fonctions, certains arguments ont été

retirés. Seuls les arguments importants pour la compréhension de la fonction ont été conservés.

La fonction translateAstd permet de traduire un ASTD. Si l'ASTD est élémentaire, aucune opération ne peut être exécutée dans l'ASTD. Si c'est un automate, la variable *subMachine* contient la liste des opérations qui peuvent être exécutées dans un des sous-automates de l'automate courant et la variable *transitionMachine* contient la liste des opérations correspondant à la traduction des transitions de l'automate courant. Ces deux listes d'opérations sont ensuite rassemblées dans une liste unique qui contient une unique opération par étiquette, grâce à la fonction mergeOpList. La variable *subMachine* est calculée à partir de la fonction mutuellement inductive translateStateList. Cette fonction traduit le premier automate de la liste d'états et modifie ses préconditions et postconditions, puis traduit le reste de la liste. Les deux listes d'opérations obtenues sont fusionnées en utilisant la fonction mergeOpList. La modification des préconditions et postconditions consiste à ajouter l'égalité $State_name = n$ et l'affectation $State_name := n$ de la figure 3.1.

La fonction mergeOpList prend en argument deux listes d'opérations et renvoie une liste d'opérations telle que chaque étiquette de transition est unique. Elle ajoute récursivement chaque opération de la première liste dans la deuxième. La fonction d'ajout d'une opération ajoute l'opération en fin de liste si aucune opération ne porte la même étiquette. Sinon, elle fusionne les deux opérations en une opération unique. La précondition de l'opération est la disjonction des préconditions des deux opérations, la postcondition est une substitution de type **SELECT** qui permet de choisir la bonne postcondition en fonction de la précondition.

La fonction translateAllTransitions traduit récursivement l'ensemble des transitions. Pour chaque transition, une précondition et une postcondition sont calculées par une fonction makePrePost. Cette fonction implémente directement le tableau 3.1. Enfin, pour calculer toute les préconditions et postconditions, une fonction makeFinalPredicate permet de calculer le prédicat qui correspond au fait qu'un automate soit dans un état final et une fonction initializeAllVariable permet de créer une substitution qui met toute les variables correspondant à l'état d'un ASTD à leur valeur initiale.

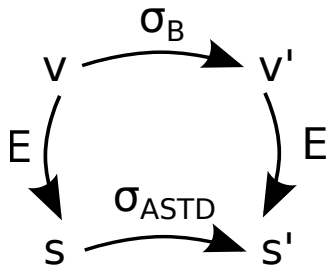


figure 3.2 – Simulation

3.3.4 Théorème à prouver

Théorème

On cherche à montrer que si la traduction d'un ASTD produit une liste d'opérations B, alors l'ASTD traduit simule la machine obtenue par la traduction. Une spécification ASTD simule une spécification B (voir figure 3.2) si il existe une relation E entre les états de l'ASTD et les états de la machine B telle que s'il est possible d'exécuter une opération B à partir d'un état v des variables, et qu'à partir de v, cette opération aboutit à un état v', alors à partir de tout état s de l'ASTD en relation avec l'état v selon la relation E il est possible d'exécuter une transition de même étiquette, et cette transition aboutit à un état s' en relation avec v' selon la relation E. Plus formellement, le théorème à montrer est donc le théorème suivant :

Théorème 6. ***Correction de la traduction***

Pour tout ASTD A

Si il existe une machine B M_A *telle que* M_A *est la traduction de A, alors,*

Pour toute étiquette σ *:*

$$
\begin{array}{l}
\forall (v, v' \in \mathit{State}_B) \\
\quad (s \in \mathit{State}) \cdot \\
\qquad E(s, v) \wedge v \xrightarrow{\sigma}_{M_A} v' \Rightarrow \\
\qquad \exists (s' \in \mathit{State}) \cdot \\
\qquad\qquad s \xrightarrow{\sigma}_{A} s' \wedge E(s', v')
\end{array}
$$

Pour prouver ce théorème, il est nécessaire de définir la relation entre les états de la machine B et les états des automates. L'idée de cette relation est de dire qu'un état B est équivalent à un état d'ASTD si la valeur associée à la variable représentant l'état d'un ASTD est le nom de l'état courant et si le reste de la fonction qui représente l'état est équivalent à l'état historique, sauf pour les variables de l'ASTD correspondant à l'état courant. Plus formellement soit un ASTD $A \triangleq \langle \mathsf{aut}, name, \Sigma, \nu, \delta, SF, DF, init \rangle$ dans un état $s = \langle \mathsf{aut}_\circ, n_1, h, s_1 \rangle$. Un état v est dit équivalent à l'état s selon la relation E si

— il existe un état v' équivalent à la fonction h associant à chaque ASTD son état historique,

— un état v_1 équivalent à l'état s_1,

— $v = (v' \lhd\!\!\!- v_1) \cup \{\mathit{State_name} \mapsto n_1\}$

Un état v' est dit équivalent à l'état h s'il est possible de faire une partition (au sens ensembliste) de v' telle que chaque partie de v' est équivalente à un des états historiques de h.

Illustrons cette relation sur l'automate de la figure 3.3. Pour coder l'état de cet ASTD, la machine B contiendra 4 variables. Une variable *State_auto* contient la valeur associée au plus haut niveau hiérarchique. Ses valeurs possibles sont les valeurs de l'ensemble $\{S1, S2, S3, S4\}$. Une variable *State_S2* contient la valeur associée à l'état de l'automate $S2$. Ses valeurs possibles sont les valeurs de l'ensemble $\{S2_1, S2_2, S2_3, S2_4\}$. De même, une variable *State_S3* encode l'état associé à l'automate $S3$ et une variable *State_S4* encode l'état associé à l'automate $S4$.

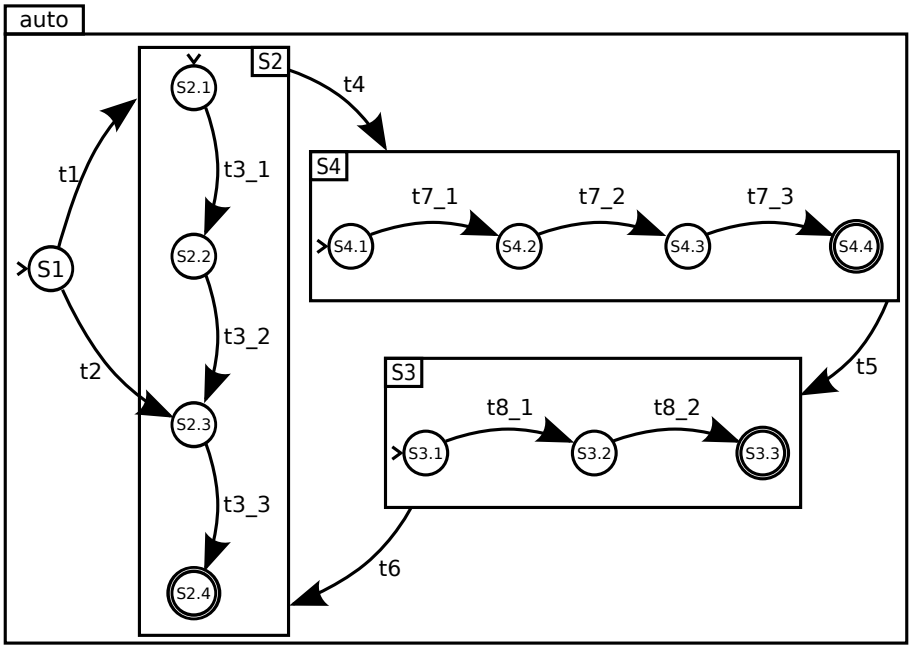


figure 3.3 – Un exemple d'automate

Nous cherchons à illustrer l'état courant de l'ASTD après avoir exécuté la séquence d'événements suivant :

$$t_1; t_4; t_5; t_{8_1}; t_6; t_{3_1}; t_4; t_5$$

L'ASTD est alors dans l'état $S3_1$ de l'état $S3$. Suite à un passage dans l'état $S2$, l'état local courant de l'état $S2$ est l'état $S2_2$. De même, l'état local courant de l'état $S3$ est l'état $S3_2$ et l'état local courant de l'état $S4$ est l'état $S4_1$. L'état s décrit ci-dessus est formalisé de la manière suivante :

$$\begin{aligned} s &= \langle \mathsf{aut}_\circ, S3, h, \langle \mathsf{aut}_\circ, S3_1, [], \mathsf{elem}_\circ \rangle \rangle \\ &\text{avec} \\ h &= \{ S2 \mapsto \langle \mathsf{aut}_\circ, S2_2, [], \mathsf{elem}_\circ \rangle, \\ &\quad\ S3 \mapsto \langle S3_2, [], \mathsf{elem}_\circ \rangle, \\ &\quad\ S4 \mapsto \langle \mathsf{aut}_\circ, S4_1, [], \mathsf{elem}_\circ \rangle \} \end{aligned}$$

Un état v équivalent à l'état s peut être trouvé suivant la méthode suivante :

— L'état v' équivalent à h est l'état

$$v' = \{State_S2 \mapsto S2_2, State_S3 \mapsto S3_2, State_S4 \mapsto S4_1\}$$

— L'état v_1 équivalent à $\langle \mathsf{aut}_\circ, S3_1, [], \mathsf{elem}_\circ \rangle$ est l'état

$$\{State_S3 \mapsto S3_1\}$$

— L'état v est donc l'état

$$\{State_S2 \mapsto S2_2, State_S3 \mapsto S3_1, State_S4 \mapsto S4_1, State_auto \mapsto S3\}$$

Implémentation en Coq

Le langage Coq permet d'écrire directement des théorèmes. Le théorème 6 est donc directement traduit dans le langage. Il faut ensuite effectuer la preuve. Une esquisse de cette preuve est expliquée dans la section 3.4.

La relation entre les états des automates et les états de la machine B est modélisée en utilisant un prédicat inductif de type suivant :

```
Inductive stateAreEquivalent : bState → state → astd_name → Prop
```

3.4 Preuve

L'objectif de cette section est de présenter le schéma de la preuve du théorème de correction de la traduction. Cette preuve a été réalisée dans le langage Coq. Cependant, lire une preuve Coq n'est pas facile et peu intéressant. Dans cette section, nous présentons donc le schéma de l'induction mutuelle utilisé pour la preuve, quelques lemmes principaux nécessaires à la preuve et enfin le schéma de la preuve principale du théorème. Pour faciliter la lecture, les théorèmes ne sont pas écrits en Coq, mais dans un langage plus naturel, qui utilise cependant les mots clés de Coq.

3.4.1 Induction mutuelle

La structure des automates est définie en Coq par le type inductif suivant :

```
Inductive astd :=
|Elem : astd_name → astd
|Automata : astd_name → list astd → list transition → list astd_name → list astd_name
→ astd_name → astd.
```

Pour construire un automate, il faut donc une liste d'automates. Le type list de la bibliothèque standard de Coq est définie de la manière suivante :

```
Inductive list (A : Type) : Type :=
|nil : list A
|cons : A → list A → list A.
```

Il y a donc une induction mutuelle entre les automates et les listes d'automates. Pour prouver une proposition sur les automates, le schéma d'induction simple ne suffit pas : il est nécessaire d'utiliser un schéma d'induction mutuelle. On cherche à prouver une proposition P : astd → Prop. Il faut pour cela exhiber une proposition $P0$: list astd → Prop sur les listes d'automates et montrer les proposition suivantes :

- P(Elem *name*) : la proposition P est vraie pour les ASTD élémentaires.
- $P0$(*states*) ⇒ P(Automata *name states transitions DF SF init*) : si la propriété $P0$ sur les listes d'automates est vraie pour une liste d'automates *states*, alors la proposition est vraie pour un automate construit à partir de cette liste d'automates.
- $P0$(nil) : la proposition $P0$ est vraie pour la liste d'automates vide.
- $P(A) \wedge P0$(*states*) ⇒ $P0$(cons A *states*) : si la propriété P est vraie pour un automate A et que la propriété $P0$ est vraie pour une liste d'automates *states*, alors la propriété $P0$ est vraie pour la liste d'automates construite à partir de A et *states*.

Si ces quatre propositions sont vraies, alors la propriété P est vraie pour tout automate A.

3.4.2 Lemmes

Plusieurs lemmes sont nécessaires pour montrer le théorème de correction de la traduction. On montre d'abord que les fonctions makeFinalPredicate et initAllVariable sont correctes. Pour montrer que la fonction makeFinalPredicate est correcte, il faut montrer que pour tout état v de la machine B et pour tout état s de l'automate A équivalent à l'état v, si le prédicat obtenu par la fonction makeFinalPredicate est vrai dans l'état v, alors l'état s est final pour l'automate A. Ce lemme se traduit plus formellement de la manière suivante :

Lemme 6. ***Correction de makeFinalPredicate***

Pour tout automate A,
Soit $pred$ = makeFinalPredicate A,
Pour tout état v d'une machine B et pour tout état s de A,

$$v \vdash pred \wedge E(v, s) \Rightarrow final(s)$$

De même, il est nécessaire de montrer que la fonction initializeAllVariable est correcte. Pour montrer la correction de cette fonction, il faut montrer que si la substitution générée par la fonction initializeAllVariable à partir d'un ASTD A permet d'atteindre un état v' à partir d'un état v, alors il existe un état s' équivalent à l'état v' et que cet état s' est l'état initial de A soit le lemme suivant :

Lemme 7. ***Correction de initializeAllVariable***

Pour tout automate A,
Soit sub = initializeAllVariable A,
Pour tout état v, v' d'une machine B,

$$v \xrightarrow{sub} v' \Rightarrow \exists(s' \in \mathsf{State}) \cdot E(v', s') \wedge s' = init(A)$$

Les lemmes de correction des fonctions makeFinalPredicate et initializeAllVariable se montrent par induction sur la forme de l'ASTD A. Ces fonctions sont définies grâce à une induction mutuelle sur la forme de l'ASTD. Les lemmes sont donc prouvés en utilisant l'induction mutuelle définie dans la section 3.4.1.

A partir de ces deux lemmes, on peut montrer que la construction des préconditions et des postconditions est correcte, puis par induction que la fonction qui traduit une liste de transitions (translateAllTransitions) est correcte. La fonction translateAllTransitions construit une liste d'opérations B à partir d'une liste de transitions. Pour prouver que cette fonction est correcte, ils s'agit de montrer que s'il est possible d'exécuter une opération dans cette liste d'opérations, alors il est possible d'exécuter une transition portant le même label dans un automate construit à partir de cette liste de transitions soit le lemme suivant :

Lemme 8. ***Correction de translateAllTransitions***

Pour tout automate $A =$ *(Automata name states transitions SF DF init)*

Soit $ opes =$ *translateAllTransitions transitions*

Pour tout état v *et* v' *d'une machine B, pour tout état* s *de A,*

$$v \xrightarrow{\sigma}_{opes} v' \wedge E(v, s) \Rightarrow \exists(s' \in \mathsf{State}) \cdot E(v', s') \wedge s \xrightarrow{\sigma}_{A} s'$$

Enfin, on cherche à montrer que la fonction mergeOpList est correct. Pour montrer que cette opération est correcte, on montre que s'il est possible d'exécuter une opération dans la liste des opérations obtenues à partir de deux listes d'opérations, c'est qu'il était possible d'exécuter cette opération soit dans la première liste, soit dans la deuxième, soit le lemme suivant :

Lemme 9. ***Correction de mergeOpList***

Pour tout liste d'opérations l_1 *et* l_2*,*

Soit $l =$ *mergeOpList* l_1 l_2

Pour tout état v *et* v' *d'une machine B,*

$$v \xrightarrow{\sigma}_{l} v' \Rightarrow v \xrightarrow{\sigma}_{l_1} v' \vee v \xrightarrow{\sigma}_{l_2} v'$$

3.4.3 Preuve du théorème

À partir des lemmes définis dans la partie précédente, on peut donc montrer le théorème de correction de la traduction. Ce théorème est défini comme une proposition sur les automates et doit être montré en utilisant un schéma d'induction mutuelle

(comme expliqué dans la section 3.4.1). Il est donc nécessaire d'exhiber une proposition sur les listes d'automates qui pourra servir d'hypothèse d'induction. L'idée de cette proposition est la suivante : s'il est possible d'exécuter une opération dans la liste d'opérations créée grâce à la fonction translateStateList, alors il existe un sous-automate dans la liste des automates dans lequel il est possible d'exécuter une transition portant la même étiquette. Plus formellement cette proposition s'écrit de la manière suivante :

Hypothèse d'induction. *Soit un automate A*

Soit opes le résultat de la traduction de la liste des sous-automates de A

Pour tout état v et v' de la machine B, pour tout état $(\mathsf{aut}_\circ, n, h, s)$ de l'automate A, On a

$$v \xrightarrow{\sigma}_{opes} v' \wedge E(v, (\mathsf{aut}_\circ, n, h, s)) \Rightarrow \\ \exists(s' \in \mathsf{State}) \cdot (E(v', (\mathsf{aut}_\circ, n, h, s')) \wedge s \xrightarrow{\sigma}_{\nu(n)} s')$$

Il faut ensuite démontrer les quatre cas de l'induction mutuelle.

Traduction d'un état élémentaire

Si l'état traduit est un état élémentaire, il est traduit par une liste vide d'opérations. Or il est impossible d'exécuter une opération dans une liste vide d'opérations. La preuve est donc triviale dans ce cas.

Traduction d'un automate

Il s'agit de prouver que s'il est possible de traduire un automate en une liste d'opérations et qu'il est possible d'exécuter une opération dans cette liste d'opérations, il est possible d'exécuter une transition portant la même étiquette dans l'automate de départ.

Soit l la liste d'opérations résultant de la traduction de l'automate A. On sait que l est obtenue en fusionnant la liste d'opérations l_1 obtenue en traduisant l'ensemble des sous-automates de A avec la liste l_2 obtenue en traduisant l'ensemble des

transitions de A. En utilisant le **Lemme 9** (à propos de la fonction mergeOpList), on sait que s'il est possible d'exécuter une opération dans l, c'est qu'il est possible d'exécuter cette opération soit dans l_1, soit dans l_2. Si l'opération est exécutée dans l_1, en utilisant l'**Hypothèse d'induction** et la sixième règle d'inférence (aut_6), on montre qu'il est possible d'exécuter une transition portant la même étiquette dans l'automate A. Si l'opération est exécutée dans l_2, en utilisant le **Lemme 8** (à propos de la traduction d'un liste de transitions), on montre qu'il est possible d'exécuter une transition portant la même étiquette dans A.

Traduction d'une liste vide d'automates

Une liste vide d'automates est traduite par une liste vide d'opérations. Or il est impossible d'exécuter une opération dans une liste vide d'opérations. La preuve est donc triviale dans ce cas.

Traduction d'une liste non vide d'automates

Il s'agit de montrer l'hypothèse d'induction, c'est-à-dire que s'il est possible de traduire une liste d'automates par une liste d'opérations, alors il est possible d'exécuter une transition dans un des sous-automates de cette liste. Pour prouver cela, nous disposons de deux hypothèses d'induction : la première est que le théorème de correction de la traduction est vrai sur le premier automate de la liste, la deuxième est que l'**Hypothèse d'induction** est vraie sur le reste de la liste d'automates.

Soit l la liste d'opérations résultant de la traduction d'une liste d'automates. Cette liste est obtenue en fusionnant une liste d'opérations l_1 obtenue en traduisant le premier automate de la liste avec la liste d'opérations l_2 obtenue en traduisant le reste de la liste. En utilisant le **Lemme 9** (à propos de la fonction mergeOpList), on sait que s'il est possible d'exécuter une opération dans l, c'est qu'il est possible d'exécuter cette opération soit dans l_1 soit dans l_2. Si l'opération est exécutée dans l_1, en utilisant l'hypothèse d'induction sur le premier automate de la liste, on montre qu'il est possible d'exécuter une transition dans le premier automate de la liste. Si l'opération est exécutée dans l_2, en utilisant l'hypothèse d'induction sur le reste de

la liste, on montre qu'il est possible d'exécuter une transition dans un sous-automate du reste de la liste.

3.5 Conclusion et perspectives

Dans les sections précédentes, nous avons résumé succinctement les principaux lemmes et le schéma général de la preuve réalisée. Cette preuve a été réalisée en utilisant l'assistant de preuve Coq, et nécessite en réalité environ 200 lemmes et théorèmes. La traduction en Coq de la sémantique et la preuve globale de tous ces théorèmes et lemmes représentent plus de 8000 lignes de Coq (Environ 1500 lignes pour la traduction des langages et de leur sémantique, 400 lignes pour les fonctions de traduction et le reste pour la définition des théorèmes et leurs preuves - 4350 lignes de preuves). Cette preuve permet l'extraction d'un code OCaml qui garantit que la traduction réalisée est correcte. L'ensemble de la preuve est disponible en ligne [1].

Par manque de temps cependant, quelques lemmes ont été admis. En effet, il y a trois types de transitions (*local*, *from_sub* et *to_sub*) et chacune de ces transitions peut être soit une transition de type final, soit une transition de type non final. Il y a donc six lemmes qui permettent de montrer que les préconditions et postconditions générées pour modéliser une transition sont corrects. Parmi ces six lemmes, un seul a été prouvé (celui pour les transitions de type *local* et finales). Les preuves pour les autres transitions suivent un raisonnement similaire : leurs preuves sont techniques et fastidieuses mais les lemmes associés devraient être corrects.

De plus, ce chapitre ne présente qu'un preuve partielle de l'outil de traduction. En effet, nous ne présentons que la preuve de correction de la traduction des automates. Pour valider complètement l'outil de traduction pour les automates, il faudrait également montrer la complétude de la traduction, c'est-à-dire que la machine B qui résulte de la traduction d'un ASTD simule l'ASTD. De plus, l'ajout de cette preuve ne permettrait de valider que la traduction des automates. Il faudrait ensuite étendre la traduction et sa preuve à tous les autres opérateurs des ASTD. Enfin, la preuve de la correction comporte une condition forte : le théorème prouvé vérifie que si l'outil de traduction arrive à traduire un ASTD, alors cette traduction est correcte vis-à-vis de

1. www.lacl.fr/~tfayolle/

l'ASTD. La manière la plus simple d'avoir un outil vérifiant ce théorème est donc de créer un outil qui ne renvoie jamais de traduction. Il faudrait donc définir ce qu'est un ASTD valide et prouver que l'outil de traduction produit une machine B pour tout ASTD valide.

Chapitre 4

Règles de simplification des ASTD

4.1 Introduction

Lors de la modélisation d'un système en utilisant la méthode présentée dans le chapitre 2, il est nécessaire de prouver la cohérence de la modélisation. Pour rappel, le système est spécifié en deux parties. D'une part, une spécification ASTD modélise l'agencement des actions et d'autre part, une spécification Event-B spécifie les modification qu'apportent ces actions au modèle de données. Les événements du modèle de données peuvent être gardés. Prouver la cohérence de la spécification consiste à montrer que lorsqu'une transition doit être exécutée (selon la spécification ASTD), les variables d'état caractérisant les données du système sont dans un état permettant l'exécution de l'événement Event-B correspondant (les variables d'état satisfont la garde). Pour cela, les ASTD sont traduits en langage B et l'événement modélisant les modifications apportées au modèle de données est appelé dans l'opération qui est la traduction de chaque étiquette de transition. Dans la traduction des ASTD en langage B, l'état des ASTD est codé par un ensemble de variables d'état et le nombre de variables d'état nécessaires croît en même temps que le nombre de niveaux hiérarchiques de l'ASTD. Plus l'ASTD aura de niveaux hiérarchiques et plus la relation entre les variables d'état associées à l'ASTD et les variables d'état associées à la modélisation de données sera complexe. L'objectif de ce chapitre est de présenter quelques règles de simplification des ASTD pour simplifier les preuves en B associées à la modélisation.

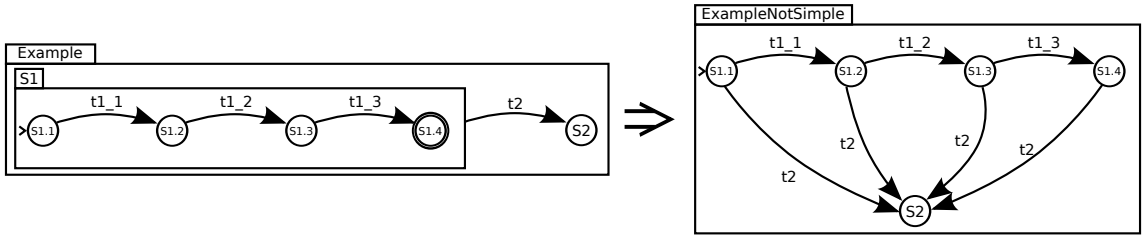


figure 4.1 – Un exemple de “simplification” non souhaitée

L'objectif de ce chapitre n'est cependant pas de proposer des règles systématiques permettant de réduire le nombre de niveaux hiérarchiques de tous les astd, notamment pour les deux raisons suivantes :

— Tous les types d'astd ne sont pas simplifiables. Il est très compliqué de traduire une synchronisation quantifiée en automate par exemple : l'automate contiendrait toute les combinaisons d'entrelacement possible pour toute les variables de l'ensemble de quantification.

— Certaines “simplifications” diminuant le nombre de niveaux hiérarchiques augmentent en pratique la complexité de la modélisation. Dans le cas de la modélisation présentée à la figure 4.1 par exemple, le seul moyen de diminuer le nombre de niveaux hiérarchiques est de faire partir la transition t_2 de tous les sous états de S_1. Dans la spécification de départ, il faut vérifier que l'état de l'astd est S_1. Dans la spécification d'arrivée, la condition pour exécuter t_2 est une disjonction de quatre conditions.

Dans ce chapitre nous proposons donc un catalogue non exhaustif de règles que nous avons jugées utile d'appliquer dans nos cas de modélisation. Ces règles sont au nombre de trois. La première concerne les fermetures de Kleene (section 4.3), la deuxième l'opérateur de choix (section 4.4) et la troisième la simplification des automates (section 4.5). Les règles permettant de réécrire l'opérateur de choix et la fermeture de Kleene s'appliquent à des automates dont aucune transition n'arrive à l'état initial. Nous définissons donc également une règle qui permet de transformer un astd ayant une transition arrivant à l'état initial en un astd sans transition arrivant à l'état initial (section 4.2). Ce catalogue pourrait évidemment être complété suivant

les besoins de futurs cas. Ces règles de réécriture ont été écrites avec Étienne Corbillé, dans le cadre d'un stage de fin d'étude au sein de l'entreprise Ikos Consulting.

Enfin, nous rappelons que l'objectif de ces règles est de réécrire les spécifications ASTD afin de faciliter les preuves après la traduction en langage B. Les spécifications obtenues après réécriture ne sont donc généralement pas plus lisibles. Cependant, ces règles ont été traduites en OCamL et ont été intégrées à l'outil de traduction des ASTD vers le langage B. Les réécritures sont donc transparentes pour la personne qui modélise le système.

4.1.1 Généralités

Les règles données dans les sections suivantes permettent de simplifier un certain nombre d'opérateurs d'ASTD. Cependant, ces règles ne s'appliquent pas à tous les ASTD du type concerné. La section 4.3 présente par exemple une règle de réécriture de la fermeture de Kleene, mais il n'est possible de réécrire que quelques formes de fermeture de Kleene. Chaque section présente donc brièvement la règle et les conditions que doit respecter l'ASTD pour que la règle puisse être appliquée. Ces conditions d'application ne sont pas nécessairement les conditions les plus faibles (il est parfois possible de définir une règle de réécriture fonctionnant en dehors de ces conditions). Le choix de ces critères d'application est le fruit d'un arbitrage entre deux critères parfois opposés : la règle doit pouvoir être décrite suffisamment simplement pour être comprise, mais elle doit aussi pouvoir s'appliquer sur un nombre maximum d'ASTD. De plus, le langage ASTD est un langage très expressif dans lequel il est possible de combiner énormément d'opérateurs. Cependant, beaucoup de ces combinaison sont peu utilisées dans la pratique, et des restrictions même fortes permettent de couvrir les cas les plus courants (ou au moins la majorité des cas les plus courants).

Puisque les ASTD sont un langage formel et que ces règles permettent de transformer des spécifications écrites dans ce langage, il est nécessaire de s'assurer que la spécification obtenue après la réécriture a le même sens que la spécification de départ. Pour s'assurer que le sens de la spécification ne change pas, il faut vérifier que tout ASTD sur lequel une règle de réécriture a été appliquée est équivalent à l'ASTD réécrit au sens de la bisimulation. Soient deux ASTD A_1 et A_2, on dit que A_1 et A_2 sont

équivalents au sens de la bisimulation si A_1 simule A_2 et si A_2 simule A_1. De plus, on dit que A_2 simule A_1 s'il existe une relation entre les états de A_2 et les états de A_1 telle que s'il est possible d'exécuter une transition dans un état de A_2, alors il est possible d'exécuter cette transition dans un état équivalent de A_1 et la transition arrive dans un état équivalent de A_1.

Les règles de réécriture peuvent s'appliquer sur un sous astd d'un astd donné. Montrer qu'un astd sur lequel s'applique une règle est équivalent à l'astd résultant de la réécriture n'est donc pas suffisant : il faut montrer la bisimulation pour tout astd contenant un astd pouvant être réécrit. Dans la pratique, le reste de l'astd n'est pas modifié. Pour montrer la bisimulation, il suffit donc de montrer la bisimulation de l'astd sur lequel s'applique la règle et de montrer que les états initiaux et finaux sont équivalents. La définition formelle de la simulation est la définition suivante :

Définition 4. ***Simulation***

Soient A_1 et A_2 deux astd*,*

On dit que A_2 simule A_1 ssi il est possible de trouver une relation R entre les états de A_1 et les états de A_2 telle que pour toute transition t, on a

$$\forall(s_1, s_1', s_2) \cdot (s_1 \xrightarrow{t}_{A_1} s_1' \wedge R(s_1, s_2) \Rightarrow \exists s_2' \cdot (s_2 \xrightarrow{t}_{A_2} s_2' \wedge R(s_2, s_2')))$$

Et

$$\forall s_1 \cdot (s_1 = \mathit{init}(A_1) \Rightarrow (\exists s_2 \cdot (s_2 = \mathit{init}(A_2) \wedge R(s_1, s_2))))$$
$$\forall s_1 \cdot (\mathit{final}(s_1) = \mathit{true} \Rightarrow \exists s_2 \cdot (\mathit{final}(s_2) = \mathit{true} \wedge R(s_1, s_2)))$$

Les preuves de bisimulation sont très longues et techniques. Dans ce chapitre, nous ne présentons donc que les esquisses de ces preuves, en donnant les arguments clé pour lesquels nous sommes persuadés que ces règles sont bonnes. Pour une utilisation de ces règles en toute confiance, ces preuves devraient être effectuées plus rigoureusement (avec un prouver interactif par exemple).

4.1.2 Notations

Les règles de réécriture s'appliquent dans le cadre de structures contenant des automates et l'idée générale de ces règles est d'ajouter des transitions et de fusionner des états pour permettre de garder le comportement de la spécification en modifiant sa structure. Pour simplifier l'écriture de ces règles, nous définissons un certains nombre de notations. Avant de définir ces notations, nous rappelons quelques notions sur le type automate des ASTD.

Rappels

Soit un ASTD $A \triangleq \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$ de type automate. L'ensemble Σ est l'alphabet de l'automate (l'ensemble des noms de transitions possibles). L'ensemble N est l'ensemble des noms des états de l'ASTD A et ν est une fonction qui associe à chaque nom d'état le sous ASTD qui lui correspond. L'ensemble δ est l'ensemble des transitions de l'ASTD. Les ensembles SF et DF contiennent les noms des états finaux (SF pour *Shallow Final* et DF pour *Deep Final*) n_0 est l'état initial.

L'ensemble des transitions δ est sous ensemble de l'ensemble $\langle \eta, \sigma, \phi, final? \rangle$. $\sigma \in \Sigma$ est l'étiquette de la transition, ϕ est la garde et *final*? est une valeur booléenne qui est à vraie si la transition est une transition dite d'état final. Une transition d'état final ne peut être exécutée que si l'ASTD d'où part la transition est dans un état final. Graphiquement, si *final*? est à l'état *true*, la flèche représentant la transition à un point à son origine. Enfin, η représente la flèche. Il y a trois types de flèche : les flèches locales ($\langle \mathsf{loc}, n_1, n_2 \rangle$), qui relient localement deux états de l'ASTD courant, les flèches dites *FromSub* ($\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2 \rangle$) qui relient le sous état d'un état de l'ASTD courant à un état de l'ASTD courant et les flèches dites *ToSub* ($\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat} \rangle$) qui relient un état de l'ASTD courant au sous états d'un état de l'ASTD courant. Dans le cas de ces deux dernières flèches, l'état par lequel on passe doit être un automate.

Notations

Le principe des règles de réécriture consiste à ajouter des transitions bien choisies pour simuler le fonctionnement des opérateurs réécrits. Nous avons donc besoin d'une notation qui exprime la sélection de transitions. Pour cela, nous utilisons la notation

suivante : soit un ensemble de noms d'état inclus dans l'ensemble des états d'un astd $E \subset N$ et soit δ un ensemble de transitions, on note ${}_{E\triangleleft}\delta$ l'ensemble des transitions de δ dont l'état de départ est dans E. Cet ensemble est défini formellement de la manière suivante :

$$\begin{aligned} {}_{E\triangleleft}\delta \triangleq & \{\langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_1 \in E\} \cup \\ & \{\langle\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_1 \in E\} \end{aligned}$$

L'ensemble ${}_{E\triangleleft}\delta$ ne contient pas de transitions dites *FromSub*. En effet, nous ne souhaitons sélectionner que les transitions qui partent effectivement de l'état tandis que les transitions dites *FromSub* partent d'un sous états. De la même manière, nous définissons l'ensemble $\delta_{\triangleright E}$ qui est l'ensemble des transitions arrivant à un état dont le nom est dans $E \subset N$. Cet ensemble ne contient pas de transitions dites *ToSub* pour les même raisons que précédemment. Il est défini de la manière suivante :

$$\begin{aligned} \delta_{\triangleright E} \triangleq & \{\langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_2 \in E\} \cup \\ & \{\langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_2 \in E\} \end{aligned}$$

De plus, pour réécrire les astd, des transitions sont ajoutées aux automates. Ces transitions utilisent les transitions existantes et modifient leur état de départ ou leur état d'arrivée. Une notation permet de définir un ensemble de transitions dont les états de départ ont été modifiés. Soit $n \in N$ le nom d'un état d'un automate et soit δ un ensemble de transitions. On note ${}_{(\lessdot n)}\delta$ l'ensemble des transitions issues de δ dont le départ a été remplacé par n. Cet ensemble est défini formellement de la manière suivante :

$$\begin{aligned} {}_{(\lessdot n)}\delta \triangleq & \{\langle\langle \mathsf{loc}, n, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \mid \exists n_1 \cdot \langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta\} \cup \\ & \{\langle\langle \mathsf{tsub}, n, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \mid \exists n_1 \cdot \langle\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta\} \end{aligned}$$

Pour les même raisons que celle invoquées ci-dessus pour la sélection des transitions à partir de leur état de départ, l'ensemble ${}_{(\not\leftarrow n)}\delta$ ne contient pas de transitions dites *FromSub*.

On définit symétriquement $\delta_{(\not\leftarrow n)}$ qui est l'ensemble des transitions issues de δ dont l'arrivée a été remplacé par n. Cet ensemble ne contient pas de transitions dite *ToSub*. Il est défini formellement de la manière suivante :

$$\begin{aligned}\delta_{(\not\leftarrow n)} \triangleq &\{\langle\langle \mathsf{loc}, n_1, n\rangle, \sigma, \phi, \mathit{final}?\rangle \mid \exists n_2 \cdot \langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final}?\rangle \in \delta\} \cup \\ &\{\langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n\rangle, \sigma, \phi, \mathit{final}?\rangle \mid \exists n_2 \cdot \langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final}?\rangle \in \delta\}\end{aligned}$$

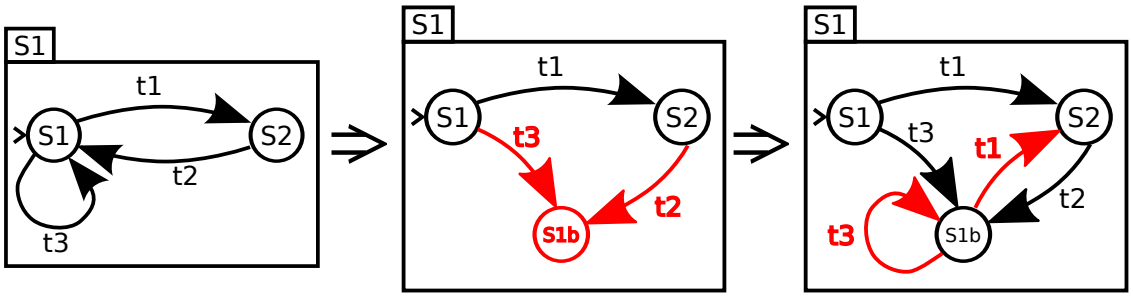
Enfin, on souhaite parfois forcer le caractère final ou non d'une transition. Soit δ un ensemble de transitions. L'ensemble $\delta[\mathit{final}? := \mathit{true}]$ est l'ensemble δ dans lequel on a forcé toutes les transitions à être des transitions d'état final et l'ensemble $\delta[\mathit{final}? := \mathit{false}]$ est l'ensemble δ dans lequel on a forcé toutes les transitions à ne pas être des transitions d'état final. Ces deux ensembles sont définis formellement de la manière suivante :

$$\begin{aligned}\delta[\mathit{final}? := \mathit{true}] &\triangleq \{\langle \eta, \sigma, \phi, \mathit{true}\rangle \mid \exists \mathit{final}? \cdot \langle \eta, \sigma, \phi, \mathit{final}?\rangle \in \delta\} \\ \delta[\mathit{final}? := \mathit{false}] &\triangleq \{\langle \eta, \sigma, \phi, \mathit{false}\rangle \mid \exists \mathit{final}? \cdot \langle \eta, \sigma, \phi, \mathit{final}?\rangle \in \delta\}\end{aligned}$$

4.2 Suppression des transitions arrivant à un état initial

4.2.1 Présentation de la règle

Les règles de transformation de la fermeture de Kleene et du choix ne sont applicables que si aucune transition n'arrive à l'état initial. Il est cependant possible dans certains cas de transformer un automate dont au moins une transition arrive à l'état initial. Pour cela, il faut créer un nouvel état qui sera la copie de l'état initial précédent. Il faut ensuite faire arriver à cet état toutes les transitions qui arrivaient à



(a) ASTD de départ

(b) L'état $S_{1_\flat}$ est ajouté. Les transitions qui arrivaient à S_1 arrivent à $S_{1_\flat}$ (les anciennes transitions sont supprimées).

(c) Les transitions qui partaient de l'état initial sont ajoutées au départ du nouvel état créé. (Elle conservent leur état d'arrivée)

figure 4.2 – Illustration du mécanisme permettant de supprimer les transitions arrivant à l'état initial

l'état initial puis ajouter toutes les transitions qui partaient de l'état initial en remplaçant leur état de départ par le nouvel état créé. La transformation est montrée à la figure 4.2. Cette règle n'est pas réellement une règle de simplification mais elle est utile pour pouvoir appliquer les règles présentées ci-dessous.

4.2.2 Formalisation de la règle

Soit un automate $A = \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle$. La règle ne s'applique que si l'état de départ est un ASTD élémentaire, ce qui se traduit par la condition formelle suivante : $\nu(n_0) \in \mathsf{ASTDElem}$. Un nouvel état doit être ajouté à l'ensemble des états de l'automate. Cet état ne peut pas porter le même nom qu'un état existant. Soit $n_{0_\flat}$ le nouvel état tel que $N \cap \{n_{0_\flat}\} = \varnothing$. L'ensemble de transitions est modifié en deux étapes : (4.1) toutes les transitions qui arrivent à l'état initial sont supprimées et remplacées par des transitions qui partent des même états mais qui arrivent à l'état $n_{0_\flat}$ et (4.2) l'ensemble des transitions partant de l'état initial dans lequel on a remplacé l'état de départ par l'état $n_{0_\flat}$ est ajouté à l'ensemble de transitions précédemment obtenu.

$$\delta' = (\delta \backslash (\delta_{\triangleright\{n_0\}})) \cup (\delta_{\triangleright\{n_0\}})_{(\triangleleft n_{0_\flat})} \tag{4.1}$$

$$\delta'' = \delta' \cup ((\triangleleft n_{0_\flat})(\{n_0\} \triangleleft \delta')) \tag{4.2}$$

Enfin, l'état $n_{0_\flat}$ est ajouté à l'ensemble N des noms d'état et à la fonction ν qui associe un ASTD à chaque nom d'état. Il est également ajouté à l'ensemble des états finaux si n_0 était un état final. Les ensembles SF' et DF' sont donc calculés de la manière suivante :

$$SF' = \begin{cases} SF \cup \{n_{0_\flat}\} & \textbf{(Si } n_0 \in SF\textbf{)} \\ SF & \textbf{(Sinon)} \end{cases}$$

$$DF' = \begin{cases} DF \cup \{n_{0_\flat}\} & \textbf{(Si } n_0 \in DF\textbf{)} \\ DF & \textbf{(Sinon)} \end{cases}$$

L'ASTD résultant est l'ASTD suivant :

$$\langle \mathsf{aut}, name, \Sigma, N \cup \{n_{0_\flat}\}, \nu \cup \{n_{0_\flat} \mapsto \mathsf{elem}\}, \delta'', SF', DF', n_0 \rangle$$

4.2.3 Esquisse de la preuve

Soit un automate A_1 vérifiant les conditions permettant d'appliquer la règle de réécriture. Soit A_2 l'automate résultant. On cherche à montrer que A_1 et A_2 sont équivalents au sens de la bisimulation. Il faut donc montrer que A_1 simule A_2 et A_2 simule A_1. Pour cela, il faut exhiber une relation entres les états possibles des deux ASTD et vérifier la définition 4. Ici la relation est la même pour les deux sens de la preuve. Dans l'ASTD A_2, on ajoute un état possible et aucun état n'est retiré. Le nouvel état ajouté est mis en relation avec l'état initial. La relation R est donc la relation identité dans laquelle on ajoute le couple $\langle \mathsf{aut}_0, n_{0_\flat}, h, \mathsf{elem} \rangle \mapsto \langle \mathsf{aut}_0, n_0, h, \mathsf{elem} \rangle$ pour tout état historique h.

A_1 **simule** A_2

L'idée de la preuve de simulation pour A_1 et A_2 est de parcourir toutes les possibilités d'exécuter une transition dans A_2 et de vérifier que cette transition peut être exécutée dans A_1 entre des états vérifiant la relation R. Il y a six règles d'inférence (voir Annexe A.1) pour les transitions dans les automates (donc six possibilités d'exécuter une transition).

— **Transition de type *local*** : On distingue les transitions qui partent ou arrivent à l'état $n_{0_\flat}$ des autres transitions. Toutes les transitions qui ne partent pas et qui n'arrivent pas à $n_{0_\flat}$ sont recopiées à l'identique dans δ''. La simulation est donc immédiate. Toutes les transitions qui arrivent ou partent de $n_{0_\flat}$ étaient des transitions qui partaient de n_0 ou arrivaient à n_0 et ces états sont en relation de simulation.

— **Transition de type *toSub* (3 règles)** : Puisque l'état initial est un état de type élémentaire, aucune transition de ce type n'arrive à l'état $n_{0_\flat}$. Il faut cependant distinguer celle qui partent de $n_{0_\flat}$ des autres. De même que précédemment, les transitions qui ne partent pas de $n_{0_\flat}$ sont recopiées à l'identique et la simulation est immédiate, et les transitions qui partent de $n_{0_\flat}$ étaient des transitions qui partaient de n_0, donc les états sont en relation.

— **Transition de type *fromSub*** : Par le même raisonnement que pour les transitions de type *toSub*, aucune transition de ce type ne part de l'état $n_{0_\flat}$. On distingue maintenant les transitions qui arrivent à $n_{0_\flat}$ de celles qui arrivent à un autre état. Les transitions qui arrivent à un autre état sont à l'identique dans δ''. Les transitions qui arrivent à $n_{0_\flat}$ sont des transitions qui arrivaient à n_0, qui est en relation de simulation avec $n_{0_\flat}$.

— **Transition dans un sous-**astd : Puisque l'état $n_{0_\flat}$ est un état élémentaire, aucune transition ne peut être exécutée dedans. Les autres états sont recopiés tels quels, la simulation est donc immédiate.

Il faut ensuite vérifier la relation pour les états initiaux et finaux. On a $init(A_1) = init(A_2)$ donc on a bien $R(init(A_1), init(A_2))$. De plus, si $\langle \mathsf{aut}_0, n_{0_\flat}, h, \mathsf{elem} \rangle$ est un état final, cela signifie que $n_0 \in DF \cup SF$ et donc que $\langle \mathsf{aut}_0, n_0, h, \mathsf{elem} \rangle$ est final. Le reste des états finaux est identique.

A_2 simule A_1

On parcourt de même toutes les possibilité d'exécuter une transition dans A_1 et on vérifie que cette transition peut être exécutée dans A_2. Les six possibilités d'exécuter une transition dans A_1 sont les mêmes que dans A_2 :

- **Transition de type *local*** : Si la transition n'arrive pas à n_0, elle n'est pas retirée de l'ensemble des transitions (donc elle est dans δ''). La simulation est donc immédiate. Si la transition arrive à n_0, elle est ajoutée à l'ensemble des transitions dans l'étape 4.2 et arrive à l'état $n_{0_\flat}$. Les états d'arrivée de la transition sont alors en relation.
- **Transition de type *toSub* (3 règles)** : Puisque l'état n_0 est élémentaire, aucune transition de ce type n'arrive à n_0. Or, seules les transitions arrivant à n_0 sont retirées. Toutes les transitions de type *fromSub* sont donc conservées dans A_2.
- **Transition de type *fromSub*** : Le raisonnement est le même que pour les transitions de type *local*. Si la transition n'arrive pas à n_0, elle n'est pas retirée de l'ensemble des transitions (elle est dans δ''). La simulation est donc immédiate. Si la transition arrive à n_0, elle est ajoutée à l'ensemble des transitions dans l'étape 4.2 en remplaçant l'état d'arrivée par $n_{0_\flat}$. Les deux états d'arrivée sont donc en relation.
- **Transition dans un sous-**ASTD : Puisque l'état n_0 est un état élémentaire, aucune transition ne peut être exécutée dedans. Les autres états sont recopiés tels quels, la simulation est donc immédiate.

Comme pour la simulation dans l'autre sens, la preuve pour les états initiaux est triviale. Pour les états finaux, puisque $SF \subseteq SF'$ et $DF \subseteq DF'$, la preuve est triviale.

4.3 Fermeture de Kleene

4.3.1 Présentation de la règle

La fermeture de Kleene est un opérateur qui permet de modéliser l'exécution d'un ASTD un nombre arbitraire de fois (ce nombre pouvant être zéro). Lorsque l'ASTD

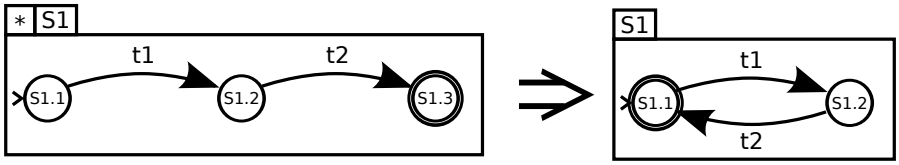


figure 4.3 – Une première intuition de la réécriture d'une fermeture de Kleene

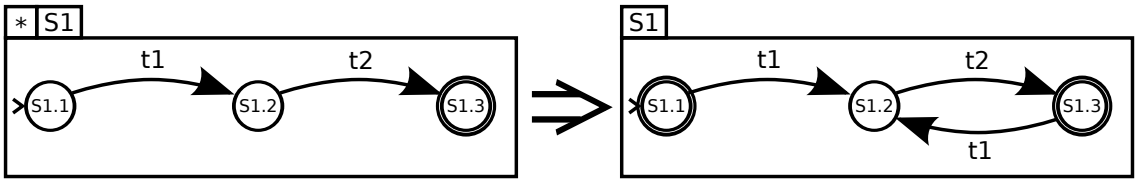


figure 4.4 – Deuxième solution proposée pour la réécriture de la fermeture de Kleene

arrive dans état final, il peut être exécuté une nouvelle fois en partant de son état initial. Nous ne traitons ici que les cas de fermeture de Kleene s'appliquant à un automate. Une première règle de réécriture est proposée : les états finaux sont fusionnés avec l'état initial. Un exemple d'application de cette règle est présenté à la figure 4.3. Ici, la règle ne fonctionne que parce les états initiaux et finaux sont tous des états élémentaires.

Pour permettre de transformer une fermeture de Kleene dont les états initiaux et finaux ne sont pas tous élémentaires, la règle est modifiée : les états de l'automate ne changent pas, mais des transitions sont ajoutées pour simuler le fonctionnement de la fermeture de Kleene. On ajoute au départ de tous les états finaux toutes les transitions qui partent de l'état initial en gardant leur état de destination. Pour modéliser le fait que l'ASTD peut ne jamais être exécuté, l'état initial devient final. L'application de cette règle sur le même ASTD que celui de la figure 4.3 est illustrée à la figure 4.4.

Cependant, puisque l'état initial devient final, cette règle ne fonctionne que si dans l'automate passé en paramètre de la fermeture de Kleene, aucune transition n'arrive à l'état initial. La figure 4.5 illustre le problème. Dans l'ASTD de départ, il est possible de sortir soit avant d'avoir exécuté la moindre transition, soit après être arrivé à l'état $S_{1.3}$. Dans l'automate résultant de la réécriture, il est possible de sortir après avoir exécuté la transition t_3 puis la transition t_4. Les deux spécifications ne sont donc pas équivalentes. Cependant, la règle décrite à la section 4.2 permet

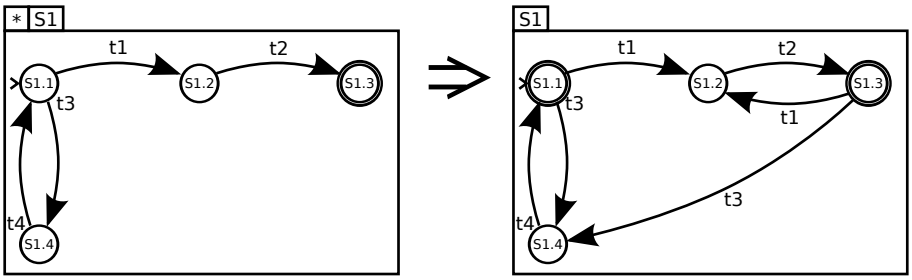


figure 4.5 – Réécriture d'un ASTD dont au moins une transition arrive à l'état initial

de transformer un automate dont au moins un transition arrive à l'état initial en un automate dans lequel aucune transition n'arrive à l'état initial. En effectuant d'abord cette transformation puis la transformation décrite ci-dessus, on obtient la transformation de la figure 4.6. Cette réécriture illustre d'ailleurs parfaitement la remarque introductive : l'ASTD obtenu n'est pas plus simple à lire, cependant sa traduction en B permettra d'écrire des invariants plus simples à prouver.

4.3.2 Formalisation de la règle

Soit un ASTD $A = \langle \star, nameKl, \langle \mathsf{aut}, name, \Sigma, N, \nu, \delta, SF, DF, n_0 \rangle \rangle$. La règle ne peut s'appliquer que si l'état initial de l'automate fermé par la fermeture de Kleene est un état élémentaire. On ne peut donc transformer l'automate A que si $\nu(n_0) \in \mathsf{ASTDElem}$. Dans la section précédente, nous avons vu qu'il n'était possible de réécrire que les automates dont aucune transition n'arrive à l'état initial. Nous distinguons donc les deux cas pour la définition de la règle.

Aucune transition n'arrive à l'état initial ($\delta_{\triangleright\{n_0\}} = \varnothing$)

Les transitions partant de l'état initial sont ajoutées à l'ensemble des transitions, en les faisant partir de tous les état finaux. Pour chaque état final, si l'état final est un état final profond, la transition doit être de type final. Si l'état final est un état final de type non profond (*Shallow Final*), la transition est doit être de type non final. Enfin, l'état initial est ajouté à l'ensemble des états finaux non profonds (puisque

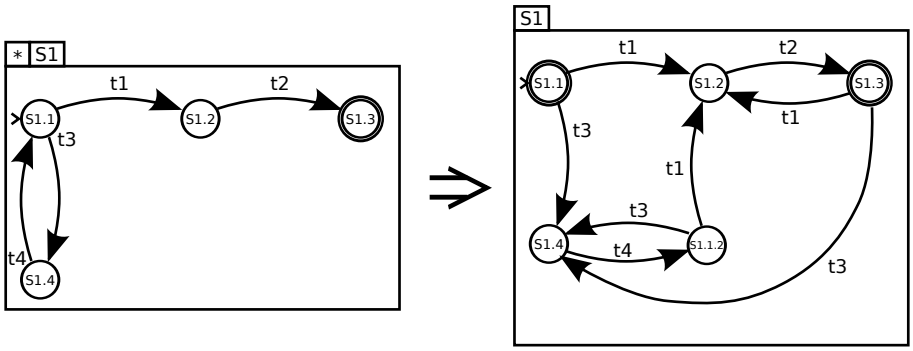


figure 4.6 – La bonne réécriture de l'automate de la figure 4.5

l'état initial est élémentaire, il peut être indifféremment dans les états profonds ou non profonds).

Formellement, le nouvel ensemble de transitions δ' est donné par :

$$\delta' = \delta \bigcup_{n_i \in DF} \{((\lessdot n_i)(n_0 \lhd \delta))[\mathit{final}? := \mathit{true}]\} \bigcup_{n_j \in SF} \{((\lessdot n_j)(n_0 \lhd \delta))[\mathit{final}? := \mathit{false}]\}$$

Et le nouvel astd remplaçant la fermeture de Kleene est l'astd :

$$\langle \mathsf{aut}, \mathit{name}, \Sigma, N, \nu, \delta', SF \cup n_0, DF, n_0 \rangle$$

Au moins une transition arrive à l'état initial ($\delta_{\rhd\{n_0\}} \neq \varnothing$)

Dans le cas où au moins une transition arrive à l'état initial, on applique la transformation de la section 4.2 puis on se ramène au cas décrit ci-dessus.

4.3.3 Esquisse de la preuve

Soit un astd $A_1 = \langle \star, \mathit{name}, A_1' \rangle$ vérifiant les conditions pour appliquer la transformation présentée ci-dessus. Soit l'astd A_2 résultat de la transformation. Il faut

montrer que A_1 simule A_2 et que A_2 simule A_1. Pour cela, il est faut exhiber une relation entre les états admis par A_1 et les états admis par A_2.

L'état d'une fermeture de Kleene est décrit sous la forme suivante : $\langle \star_0, \mathit{started}?, s \rangle$. L'état s est l'état du sous-ASTD passé en paramètre de la fermeture de Kleene et la mention *started*? est une valeur booléenne qui est vraie si la fermeture de Kleene est commencée. Dans ces conditions, la relation R est définie de la manière suivante : Pour tout s, $R(\langle \star_0, \mathit{true}, s \rangle, s)$ et $R(\langle \star_0, \mathit{false}, \mathit{init}(A_1)' \rangle, \mathit{init}(A_2))$. Sur le reste des états, la relation R est la relation identité.

A_1 simule A_2

Il y a six règles d'inférence pour les transitions dans les automates, donc six possibilités d'exécuter une transition (voir Annexe A.1). Ici nous les résumons en deux catégories : (1) la transition est une transition de l'automate A_2 et (2) la transition est une transition d'un sous ASTD de A_2. De plus, il y a deux règles d'inférence pour la fermeture de Kleene (voir Annexe A.4) : (1) la transition part de l'état initial, et elle peut être exécutée si l'ASTD A_1' est dans un état final (ou si la fermeture de Kleene n'a jamais commencé) et (2) la transition part d'un autre état et elle peut être exécutée si elle peut être effectuée dans A_1'

- **La transition est une transition de** A_2 : Les transitions de A_1' sont incluses dans les transitions de A_2. On distingue donc les deux cas : (1) la transition est dans δ (l'ensemble des transitions de A_1') et (2) la transition n'est pas dans δ
  - **Si la transition est dans** δ, elle peut être exécutée dans A_1'. Donc elle peut être exécutée dans A_1.
  - **Si la transition n'est pas dans** δ, elle fait partie des transitions ajoutées et part d'un état qui était un état final de A_1'. D'après la première règle d'inférence de la fermeture de Kleene, une transition peut être exécutée depuis un état final si elle part d'un état initial. Or les transitions ajoutées à l'ensemble des transitions sont celles partant de l'état initial.
- **La transition est une transition d'un sous-ASTD de** A_2 : A_2 contient exactement les mêmes états que A_1'. Il est donc possible de faire dans un sous-

astd de A'_1 toutes les transitions qu'il est possible de faire dans un sous-astd de A_2.

On vérifie également que les conditions pour l'état initial et les états finaux sont vérifiées. L'état $\langle \star_0, \mathit{false}, \mathit{init}(A'_1) \rangle$ est initial pour A_1 et $R(\langle \star_0, \mathit{false}, \mathit{init}(A'_1) \rangle, \mathit{init}(A_2))$. La fermeture de Kleene a deux types d'états finaux : (1) la fermeture de Kleene n'a jamais commencé et (2) le sous-astd est dans un état final. Les états finaux de A_2 sont les états finaux de A'_1 auxquels on a ajouté l'état initial de A_2. Si l'état final de A_2 est un des états de A'_1 on est dans le cas (2), si on est dans l'état initial de A_2, l'état $\langle \star_0, \mathit{false}, \mathit{init}(A'_1) \rangle$ est final pour la fermeture de Kleene et vérifie la relation.

A_2 simule A_1

Il y a deux règles d'inférence pour décrire l'exécution d'un transition dans une fermeture de Kleene : (1) la transition est une transition qui peut être exécutée dans le sous-astd ou (2) la transition peut être exécutée depuis l'état initial et le sous-astd est dans un état final ou la fermeture de Kleene n'a jamais commencé ($\mathit{started}? = \mathit{false}$).

- **La transition peut être exécutée dans** A'_1 : Soit c'est une transition de A'_1 et puisque l'ensemble des transitions de A'_1 est un sous ensemble des transitions de A_2, elle peut être exécutée dans A_2. Soit c'est une transition d'un sous-astd de A'_1 et puisque les états de A_2 sont identiques aux états de A'_1, elle peut être exécutée dans A_2.
- **La transition peut être exécutée si l'état est final ou que la fermeture de Kleene n'a jamais commencé** : Si la fermeture de Kleene n'a jamais commencé, on peut exécuter une transition qui part de l'état initial de A'_1. Les transitions de A'_1 sont dans A_2, donc A_2 peut exécuter la transition. Si l'état est final, on peut exécuter toutes les transitions qui partent de l'état initial. Or ce sont ces transitions qui ont été ajoutées dans l'ensemble des transitions de A_2.

L'état initial de A_1 est l'état $\langle \star_0, \mathit{false}, \mathit{init}(A'_1) \rangle$ qui est en relation avec $\mathit{init}(A_2)$. Il y a deux états finaux à une fermeture de Kleene : l'état où la fermeture n'a jamais commencé ($\mathit{started}? = \mathit{false}$) et l'état où le sous-astd est dans un état final. Si elle n'a jamais commencé elle correspond à l'état initial de A_2, et l'état initial de A_2 est

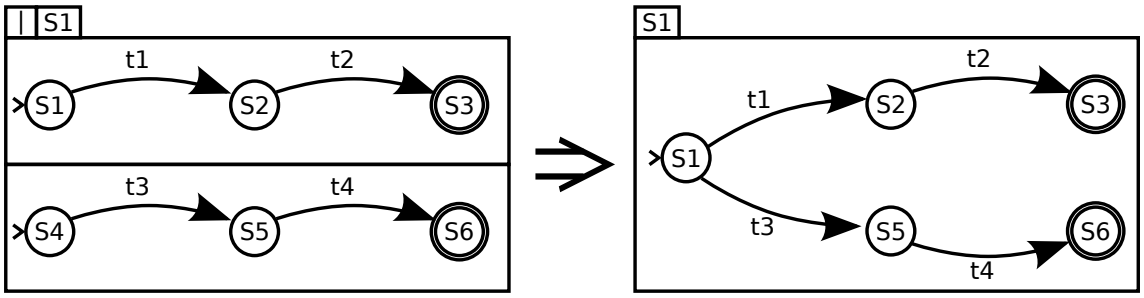


figure 4.7 – Intuition pour la réécriture du choix

final. Les états finaux de A_1' sont inclus dans les états finaux de A_2 donc si A_1' est final, A_2 l'est aussi.

4.4 Choix

4.4.1 Présentation de la règle

L'opérateur de choix des ASTD prend en argument deux ASTD et en exécute un au choix. Le choix entre les deux ASTD est donc effectué lorsque la première transition d'un des deux automates est exécutée. La règle que nous définissons ne s'applique que si les deux ASTD passés en paramètre sont des automates. Dans ce cas, on peut intuitivement rassembler leurs états initiaux en un seul état et faire partir du nouvel état toutes les transitions qui partaient des deux états initiaux, tel qu'illustré à la figure 4.7.

Cependant, la transformation présentée à la figure 4.7 ne fonctionne que parce qu'une fois que la première transition est effectuée, il n'est plus possible de repasser par l'état initial. En effet, si des transitions arrivent à l'état initial dans un des deux ASTD, il serait possible de commencer le parcours dans cet ASTD et de le finir dans l'autre. L'exemple de la figure 4.8 illustre ce cas de figure. Dans l'ASTD résultant de la réécriture, il est possible d'exécuter la trace $t_3; t_5; t_1; t_2$ alors que cette trace n'est pas admissible dans l'ASTD de départ.

Cette règle ne peut donc s'appliquer si dans les deux automates, aucune transition n'arrive à l'état initial. Si au moins une transition arrive à l'état initial, nous appli-

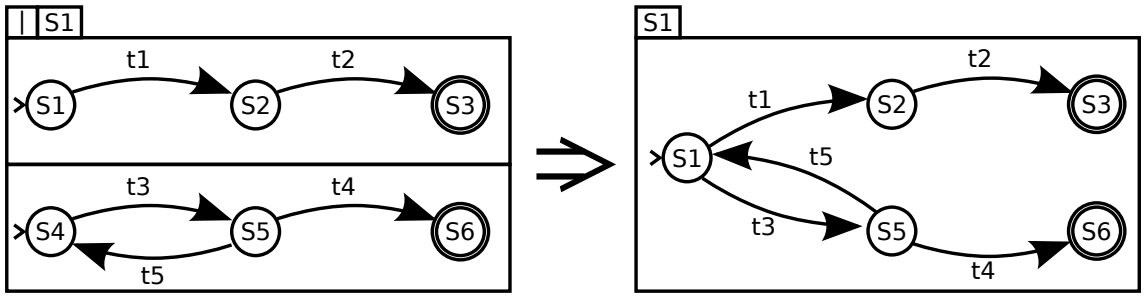


figure 4.8 – Une erreur dans la réécriture du choix

quons la transformation décrite dans la section 4.2 avant de lui appliquer la règle de transformation du choix. De plus, puisqu'il faut fusionner les états initiaux des deux automates, nous limitons l'application de cette règle aux astd de type choix dont les états initiaux des deux sous automates sont des états élémentaires.

4.4.2 Formalisation

Soit un astd de type choix $A = \langle |, name, A_1, A_2 \rangle$ dont les deux sous astd $A_1 = \langle \mathsf{aut}, name_1, \Sigma_1, N_1, \nu_1, \delta_1, SF_1, DF_1, n_{0_1} \rangle$ et $A_2 = \langle \mathsf{aut}, name_2, \Sigma_2, N_2, \nu_2, \delta_2, SF_2, DF_2, n_{0_2} \rangle$ sont deux automates. Cet astd ne peut être réécrit que si les états initiaux des deux automates sont des états élémentaires ($\nu_1(n_{0_1}) \in \mathsf{ASTDElem} \wedge \nu_2(n_{0_2}) \in \mathsf{ASTDElem}$). Pour la réécriture de ces automates, on distingue deux cas : soit au moins une transition arrive à un des états initiaux, soit aucune transition n'arrive à un état initial.

Aucune transition n'arrive à un état initial ($\delta_{1 \triangleright \{n_{0_1}\}} \cup \delta_{2 \triangleright \{n_{0_2}\}} = \varnothing$)

Si aucune transition n'arrive à un état initial, on peut appliquer la transformation illustrée à la figure 4.7. L'état initial d'un des deux automates est choisi pour être l'état initial de l'astd résultat (par exemple n_{0_1}). Le nouvel ensemble des noms d'états est l'union des ensembles de noms d'état des deux automates de départ auxquels l'état initial n_{0_2} est retiré. De même la fonction associant aux noms d'état l'astd correspondant est l'union des fonctions des deux astd à laquelle l'état n_{0_2} est également retiré.

Puisqu'on a choisi l'état n_{0_1} comme état initial, toutes les transitions de δ_1 sont conservées. On ajoute à cet ensemble toutes les transitions qui ne partent pas de n_{0_2} dans δ_2. Pour les transitions qui partent de n_{0_2}, on remplace l'état de départ de la transition par n_{0_1}. Le nouvel ensemble de transitions δ' est calculé de la manière suivante :

$$\delta' = \delta_1 \cup (\delta_2 \backslash_{\{n_{0_2}\}\triangleleft}\delta_2) \cup_{(\triangleleft n_{0_1})} (_{\{n_{0_2}\}\triangleleft}\delta_2)$$

Les états finaux de l'automate résultant sont l'union des états finaux des deux sous automates de A, sauf si l'état initial de A_2 est final. Dans ce cas on ajoute n_{0_1} à l'ensemble des états finaux de A_2. Les ensembles d'états finaux sont donc calculés de la manière suivante :

$$SF' = \begin{cases} SF_1 \cup SF_2 & \textbf{(Si } n_{0_2} \notin SF_2\textbf{)} \\ SF_1 \cup (SF_2 \backslash \{n_{0_2}\}) \cup \{n_{0_1}\} & \textbf{(Sinon)} \end{cases}$$

$$DF' = \begin{cases} DF_1 \cup DF_2 & \textbf{(Si } n_{0_2} \notin DF_2\textbf{)} \\ DF_1 \cup (DF_2 \backslash \{n_{0_2}\}) \cup \{n_{0_1}\} & \textbf{(Sinon)} \end{cases}$$

Et l'automate résultant est l'automate suivant :

$$\langle \mathsf{aut}, name, \Sigma_1 \cup \Sigma_2, N_1 \cup (N_2 \backslash \{n_{0_2}\}), \nu_1 \cup (\{n_{0_2}\} \triangleleft \nu_2), \delta', SF', DF', n_{0_1} \rangle$$

Au moins une transition arrive à un des états initiaux ($\delta_{1\triangleright\{n_{0_1}\}} \cup \delta_{2\triangleright\{n_{0_2}\}} \neq \varnothing$)

On applique la réécriture décrit à la section 4.2 dans le ou les ASTD dans lesquels au moins un transitions arrive à l'état initial, puis on se ramène au cas précédent.

4.4.3 Esquisse de la preuve

Soit l'ASTD $A_1 = \langle |, A_{1_1}, A_{1_2} \rangle$ vérifiant les conditions permettant de réécrire l'ASTD (c'est-à-dire que les état initiaux sont des états élémentaires et aucune transition n'arrive à un état initial). Soit A_2 l'ASTD résultant de cette réécriture. L'état d'un

astd de type choix s'écrit sous la forme $\langle|, side, s\rangle$ où $side$ est soit $\perp$ si aucun côté n'a été choisi, soit right si c'est l'astd de droite qui a été choisi, soit left si c'est l'astd de gauche qui a été choisi et s est soit l'état courant de l'astd lorsqu'un côté a été choisi, soit $\perp$ si aucun côté n'a été choisi.

Pour montrer la bisimulation il faut exhiber une relation entre les états de A_1 et les états de l'astd A_2. Si un côté a été choisi, l'état du choix est en relation avec l'état correspondant de l'automate A_2. On a donc $R(\langle|_0, \mathsf{left}, s\rangle, s)$ si s est un état de A_{1_1} et $R(\langle|_0, \mathsf{right}, s\rangle, s)$ si s est un état de A_{1_2} (sauf pour l'état initial). L'état initial de A_1 est en relation avec l'état initial de A_2 ($R(\langle|_0, \perp, \perp\rangle, init(A_2))$). L'état initial de A_2 est également en relation avec l'état initial de l'astd A_{1_2} qui a été retiré ($R(init(A_{1_2}), init(A_2))$).

Enfin, la sémantique des astd de type choix est définie par quatre règles d'inférence (voir Annexe A.3). Deux règles traitent le cas où on démarre l'un des deux astd (aucun choix n'a été fait et l'astd de gauche est choisi et aucun choix n'a été fait et l'astd de droite est choisi) et deux règles traitent le cas où on effectue une transition dans l'astd choisi (on est dans l'astd de gauche et une transition est exécutée dans l'astd de gauche et on est dans l'astd de droite et on exécute une transition dans l'astd de droite).

A_1 simule A_2

Puisque A_2 est un automate, il y a six règles d'inférence pour décrire l'exécution d'une transition. Ces règles sont regroupées en deux catégories : soit la transition est dans l'ensemble de transitions de A_2, soit la transition peut être exécutée dans un sous astd de A_2.

— **La transition est une transition de** A_2 : Il y a alors trois possibilités d'après la manière dont a été construit l'ensemble des transitions de A_2 : soit la transition est une transition de l'ensemble des transitions de A_{1_1}, soit c'est une transition de l'ensemble des transitions de A_{1_2} qui ne part pas de l'état initial, soit c'est un transition de l'ensemble des transitions de A_{1_2} qui part de l'état initial.

- **La transition est une transition de A_{1_1}** : Si la transition part de l'état initial, elle correspond à une transition au cours de laquelle le choix est fait. La transition part de l'état initial de A_2 qui correspond à l'état où aucun choix n'a été fait. Puisqu'elle arrive dans un état de A_{1_1} et ne peut pas arriver à l'état initial (d'après les conditions d'application de la règle), elle arrive bien dans un état qui correspond à un état dans lequel le choix de l'ASTD de gauche a été fait. De même, si elle part d'un autre état que l'état initial, l'état correspond à un état où le choix de l'ASTD de gauche a été fait et puisque la transition arrive dans un état de A_{1_1} et ne peut arriver à l'état initial, elle arrive dans un état qui correspond à un état dans lequel le choix de l'ASTD de gauche a été fait.
- **La transition est une transition de A_{1_2} qui ne part pas de l'état initial** : La transition part d'un état qui correspond à un état dans lequel le choix de l'ASTD de droite a été fait. Puisque la transition est une transition de l'ASTD A_{1_2} et qu'elle ne peut pas arriver à l'état initial, elle arrive dans un état qui correspond à un état dans lequel le choix de l'ASTD de droite a été fait.
- **La transition est une transition de A_{1_2} qui part de l'état initial** : La transition part de l'état qui correspond à l'état où aucun choix n'a été fait. Puisque c'est une transition de A_{2_2} et qu'elle ne peut arriver à l'état initial, elle arrive dans un état qui correspond à un état dans lequel le choix de l'ASTD de droite a été fait

— **La transition est une transition d'un sous-ASTD de A_2** : Tous les sous états de A_{1_1} et de A_{1_2} sont conservés sauf l'état initial de A_{1_2} qui est élémentaire (qui ne peut donc pas exécuter de transition). Les transitions permises dans A_2 sont donc permises dans le sous état correspondant de A_1.

L'état initial de A_2 est en relation avec l'état initial de A_1 (par définition de la relation R). Les états finaux de A_2 sont l'union des états finaux de A_{1_1} et A_{1_2} (en remplaçant éventuellement l'état initial de A_{1_2} par l'état initial de A_{1_1} si l'état initial de A_{1_2} est final). Pour les états finaux qui ne sont pas initiaux, la preuve est immédiate. Si l'état initial de A_2 est final, soit l'état initial de A_{1_1} soit l'état initial

de A_{1_2} est final. Cet état correspond à $\langle |_0, \perp, \perp \rangle$ qui, par définition de l'état final de l'astd choix, est final si l'un des deux états initiaux des sous-astd est final.

A_2 **simule** A_1

Les quatre possibilités d'exécuter une transition dans A_1 sont parcourues :

- **Aucun choix n'a été fait et on choisi l'astd de gauche** : La transition est une transition qui part de l'état initial de A_{1_1} et qui peut être exécutée dans A_{1_1}. Or l'ensemble des transitions de A_{1_1} est inclus dans les transitions de A_2. La transition peut donc être exécutée dans A_2.
- **Aucun choix n'a été fait et on choisi l'astd de droite** : La transition est une transition qui part de l'état initial de A_{1_2} et qui peut être exécutée dans A_{1_2}. Les transitions qui partent de l'état initial de A_{1_2} sont ajoutées à l'ensemble des transitions de A_2 en modifiant l'état de départ par l'état initial de A_{1_1}. Elles peuvent donc être exécutées dans A_2 en arrivant dans un état correspondant.
- **On exécute une transition dans l'astd de gauche (déjà choisi)** : La transition est dans l'ensemble de transitions de A_{1_1} qui est inclus dans l'ensemble des transitions de A_2. Elle peut donc être exécutée.
- **On exécute une transition dans l'astd de droite (déjà choisi)** : La transition est dans l'ensemble de transitions de A_{1_2} et ne part pas de l'état initial (puisqu'il n'est pas possible de revenir à l'état initial). Elle est donc dans l'ensemble de transitions qui est inclus à l'identique dans A_2.

L'état initial de A_1 est en relation avec l'état initial de A_2 (par définition de la relation R). Si A_1 est final, cela signifie que soit A_{1_1} est final, soit A_{1_2} est final. Or les états finaux de A_2 sont l'union des états finaux de A_{1_1} et des états finaux de A_{1_2} privé de n_{0_2} si il est final. Si c'est n_{0_2} qui est final, l'état initial de A_{1_1} a été rajouté à l'ensemble des états finaux. Donc l'état correspondant de A_2 est final.

4.5 Automates

4.5.1 Présentation de la règle

Dans certaines conditions, il est possible de simplifier un automate dont l'un des états est un automate. Le sous état est alors retiré et tous ses états sont ajoutés à l'automate supérieur, les transitions qui arrivent au sous-état sont transformées pour arriver à l'état initial de l'automate retiré. Les transitions de type *toSub* qui arrivent à un sous état de l'automate retiré sont transformées en transition de type *local* et gardent leur état d'arrivée. Cette transformation ne peut donc pas s'appliquer si une transition arrive à un état historique. Cette transformation ne s'applique que pour les états dont toutes les transitions partant de l'état supprimé sont des transitions de type final (c'est-à-dire que la transition ne peut être effectuée que si l'ASTD de départ est dans un état final). En effet, si une transition sortante n'est pas de type final, l'ASTD est réécrit suivant la règle illustrée à la figure 4.1. Dans le nouvel automate, ces transitions partent de tous les états finaux de l'automate supprimé. Un traitement symétrique à celui appliqué aux transitions de type *toSub* est appliqué aux transitions de type *fromSub*.

Un exemple de cette transformation est illustré à la figure 4.9. La seule transition sortante est la transition t_4, et elle est de type final (représenté par le point noir à son origine). La règle peut donc être appliquée. Dans l'ASTD résultant de la réécriture, la transition t_4, qui partait de l'état S_2, part de l'état final de S_2. De la même manière la transition t_1 qui arrivait à S_2 arrive à l'état initial de S_2. La transitions t_2 qui était de type *toSub* est transformée en transition locale et arrive à l'état $S_{2.3}$ où elle arrivait dans l'ASTD de départ. Symétriquement la transition t_5 est transformée en transition de type *local* et part de l'état $S_{2.3}$ dans les deux ASTD.

4.5.2 Formalisation

Notations

Avant de définir cette règle, il est nécessaire de définir de nouvelles notations. En effet, il faut pouvoir transformer en transition locale toutes les transitions de type *fromSub* et *toSub* partant ou arrivant à un état. L'ensemble ${}_{n_1}\delta$ est donc l'ensemble des

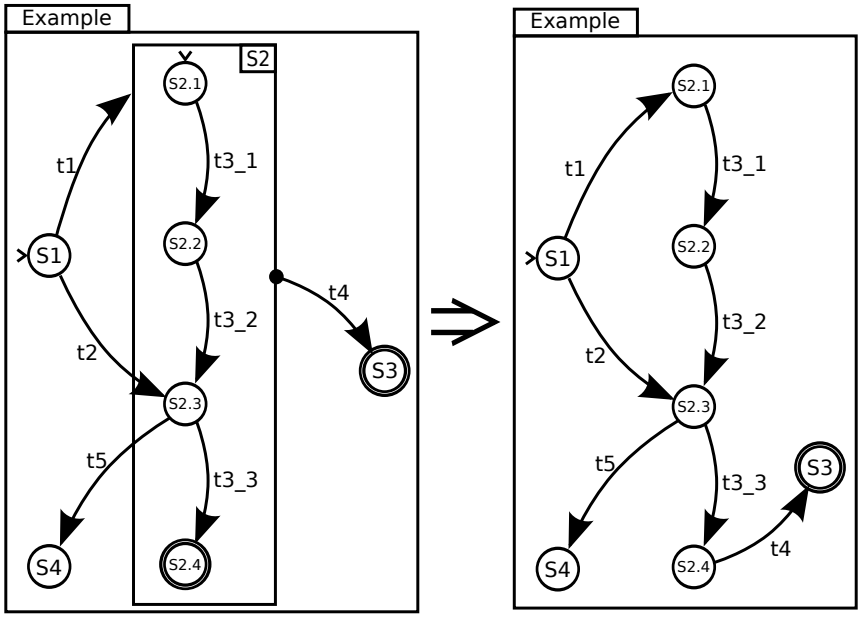


figure 4.9 – Un exemple de réécriture d'un automate

transitions issues de δ dans lequel toutes les transitions de type *toSub* qui arrivaient d'un sous état de n_1 sont transformées en transitions de type *local* en gardant leur état d'arrivée. Cet ensemble est défini de la manière suivante :

$$\not{n_1}\delta \triangleq \{\langle\langle \mathsf{loc}, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle, \mid \langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta\}$$

Symétriquement, l'ensemble $\delta_{\not{n_2}}$ est l'ensemble des transitions qui arrivaient à un sous état de n_2 et qui sont transformées en transitions locales. Cet ensemble est défini de la manière suivante :

$$\delta_{\not{n_2}} \triangleq \{\langle\langle \mathsf{loc}, n_1, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle, \mid \langle\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta\}$$

Nous souhaitons également retirer de l'ensemble des transitions toutes les transitions qui arrivent à un ensemble d'états donné ou au sous-état d'un ensemble d'états

donnés. Soit E un ensemble d'états et δ un ensemble de transitions, on note ${}_{E\lhd}\delta$ l'ensemble défini de la manière suivante :

$$\begin{aligned}{}_{E\lhd}\delta \triangleq &\{\langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_1 \notin E\} \cup \\ &\{\langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_1 \notin E\} \cup \\ &\{\langle\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_1 \notin E\}\end{aligned}$$

Symétriquement, l'ensemble $\delta_{\rhd E}$ est l'ensemble δ duquel on a retiré toutes les transitions qui arrivent à un état de E ou un sous état d'un état de E. Il est défini de la manière suivante :

$$\begin{aligned}\delta_{\rhd E} \triangleq &\{\langle\langle \mathsf{loc}, n_1, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_2 \notin E\} \cup \\ &\{\langle\langle \mathsf{fsub}, n_1, n_{1_\flat}, n_2\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_2 \notin E\} \cup \\ &\{\langle\langle \mathsf{tsub}, n_1, n_2, n_{2_\flat}\rangle, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid n_2 \notin E\}\end{aligned}$$

Enfin, afin de vérifier si toutes les transitions partant d'un état sont finales, il faut pouvoir sélectionner les transitions finales. L'ensemble $\delta_{|(\mathit{final?}=\mathit{true})}$ est donc défini de la manière suivante :

$$\delta_{|(\mathit{final?}=\mathit{true})} = \{\langle \eta, \sigma, \phi, \mathit{final?}\rangle \in \delta \mid \mathit{final?} = \mathit{true}\}$$

Formalisation de la règle

Soit un automate $A_1 = \langle \mathsf{aut}, name_1, \Sigma_1, N_1, \nu_1, \delta_1, SF_1, DF_1, n_{0_1}\rangle$ dont l'un des sous-ASTD $n_1 \in N_1$ est un automate ($\nu(n_1) = \langle \mathsf{aut}, n_1, \Sigma_2, N_2, \nu_2, \delta_2, SF_2, DF_2, n_{0_2}\rangle$). Cette ASTD ne peut être supprimé que si toutes les transitions qui en partent sont des transitions finales, ce qui se traduit par la condition suivante :

$${}_{\{n_1\}\lhd}\delta = ({}_{\{n_1\}\lhd}\delta)_{|(\mathit{final?}=\mathit{true})}$$

Si cette condition est vérifiée, on commence par retirer toutes les transitions qui arrivent et partent de l'état à supprimer (étape 4.3 ci-dessous). Toutes les transitions de type *fromSub* qui partent d'un sous état de l'état n_1 sont transformées en transition locale, de même toutes les transitions de type *toSub* qui arrivent un sous état de l'état n_1 sont transformées en transition locale (étape 4.4). Les transitions qui arrivaient à l'état n_1 sont remplacées par des transitions qui arrivent à l'état initial de n_1 (étape 4.5). De même, les transitions qui partaient de l'état n_1 sont remplacées par des transitions qui partent des états finaux de n_1. Si l'état final est un état final non profond (*Shallow Final*), la transition est une transition non finale (étape 4.6), si l'état final est un état final profond (*Deep Final*), la transition est une transition finale (étape 4.7). Enfin, toutes les transitions de n_1 sont ajoutées à l'ensemble des transitions (étape 4.8).

$$\delta' = {}_{\{n_1\}\lhd}\delta_{1\rhd\{n_1\}} \tag{4.3}$$

$$\bigcup {}_{\not{n_1}}\delta_1 \bigcup \delta_{1\not{n_1}} \tag{4.4}$$

$$\bigcup (\delta_{1\rhd\{n_1\}})_{(\lhd n_{0_2})} \tag{4.5}$$

$$\bigcup_{n_i \in SF} ((\lhd n_i)({}_{\{n_1\}\lhd}\delta_1))[\mathit{final}? := \mathit{false}] \tag{4.6}$$

$$\bigcup_{n_j \in DF} ((\lhd n_j)({}_{\{n_1\}\lhd}\delta_1))[\mathit{final}? := \mathit{true}] \tag{4.7}$$

$$\bigcup \delta_2 \tag{4.8}$$

Si l'état n_1 est l'état initial de A_1, le nouvel état initial est l'état initial de n_1, sinon il reste inchangé. De même, si n_1 est un état final de type *deep final*, les états de type *deep final* de $\nu(n_1)$ sont ajoutées aux états de type *deep final* de A_1 et les états de type *shallow final* de $\nu_1(n_1)$ sont ajoutés aux états de type *shallow final* de A_1. Si n_1 est un état final de type *shallow final*, tous les états de $\nu_1(n_1)$ sont ajoutées aux états de type *shallow final* de N_1. Ces règles permettent de calculer n'_{0_1} le nouvel état initial de A_1 et SF' et DF' les nouveaux ensembles d'état finaux. Ces calculs sont résumés dans les tableaux 4.1.

Le nouvel automate obtenu est alors l'automate suivant :

Initial	$n_1 = n_{0_1}$	$n_1 \neq n_{0_1}$
n'_{0_1}	n_{0_2}	n_{0_1}

Final	$n_1 \in SF_1$	$n_1 \in DF_1$	$n_1 \notin SF_1 \cup DF_1$
SF'	$(SF_1 \backslash \{n_1\}) \cup N_2$	$SF_1 \cup SF_2$	SF_1
DF'	DF_1	$(DF_1 \backslash \{n_1\}) \cup DF_2$	DF_1

tableau 4.1 – Définition des états initiaux et finaux

$$\langle \mathsf{aut}, name_1, \Sigma_1 \cup \Sigma_2, (N_1 \backslash \{n_1\}) \cup N_2, (\{n_1\} \lhd \nu_1) \cup \nu_2, \delta', SF', DF', n'_{0_1} \rangle$$

4.5.3 Esquisse de la preuve

Soit A_1 un automate et A_{1_1} un automate dont le nom est n_1 qui est un des états de A_1 vérifiant les conditions permettant d'appliquer la règle. Soit A_2 l'automate résultant de l'application de la règle. On cherche à prouver la bisimulation de A_1 par A_2. Pour cela il faut exhiber une relation R entre les états de A_1 et les états de A_2. Pour tout état s de A_{1_1} et tout état historique h, on a $R(\langle n_1, h, s \rangle, s)$. Pour les autres états, la relation R est la relation identité.

A_1 **simule** A_2

Il faut parcourir l'ensemble des possibilités d'exécuter une transition dans A_2 et vérifier que cette transition peut être exécutées dans A_1. Il y a six règles d'inférences pour exécuter une transition dans un automate :

— **Transition de type *local*** : Les transitions de A_2 qui n'arrivent pas et ne partent pas de A_{1_1} sont les mêmes que les transitions de A_1. Elles peuvent donc être exécutées dans A_1 dans les mêmes états. On ajoute ensuite quatre types de transitions locales :

— **Les transitions issues de la transformation d'une transition de type *fromSub* ou *toSub*** : Dans le cas des transitions de type *fromSub*, la transition exécutée dans A_2 part de l'état correspondant au sous état de

la transition *fromSub*. Le raisonnement est symétrique pour les transitions de type *toSub*.

- **Les transitions issues de la transformation d'une transition arrivant à** A_{1_1} : Les transitions qui arrivaient à A_{1_1} arrivent dans A_2 à l'état qui était initial dans A_{1_1}. Or, dans la sémantique des astd, lorsqu'une transition arrive à un état hiérarchique, elle arrive à l'état initial.
- **Les transitions issues de la transformation d'une transition partant de** A_{1_1} : Les transitions qui partaient de A_{1_1} étaient des transitions de type final (d'après la condition d'application de la règle). Dans A_2 ces transitions partent des états finaux, et peuvent donc bien être exécutées dans des états équivalents à ceux de A_1.
- **Les transitions de l'automate** A_{1_1} : Les transitions locales de l'automate A_{1_1} peuvent être exécutées dans A_1

— **Transition de type *toSub* (trois règles)** : Les seules transitions de type *toSub* qui ont été ajoutées dans les transitions de A_2 sont les transitions de type *toSub* de l'automate A_{1_1}. Ces transitions peuvent donc être exécutées dans A_1.

— **Transition de type *fromSub*** : De même que pour les transitions de type *toSub*, les seules transitions de type *fromSub* ajoutées sont les transitions de l'automate A_{1_1}. Ces transitions peuvent donc être exécutées dans A_1.

— **Transition dans un sous-**astd : Les sous-astd de A_2 sont soit des sous-astd de A_1 soit des sous-astd de A_{1_1}. Toutes les transitions que peut exécuter un sous-astd de A_2 peuvent donc être exécutées par un sous-astd de A_1.

Si l'état initial de A_2 est l'état initial de A_{1_1} cela signifie que n_1 était l'état initial de A_1. Donc l'état correspondant est initial dans A_1. Sinon, l'état initial est inchangé. Si un état est final dans A_2, il peut venir des états de A_{1_1} ou d'un autre état de A_1. Si c'est un état de A_1, c'est qu'il état final dans A_1. Si c'est un état de A_{1_1} cela signifie que l'état n_1 était final. Si l'état est dans les états de type final profond, cela signifie que l'état n_1 était de type final profond, donc l'état correspondant dans A_1 est final. Si l'état est dans les états de type final non profond, et qu'il était dans l'ensemble des états finaux non profond de A_{1_1}, l'état A_1 est dans un état final. Si c'est un état

quelconque de A_{1_1}, cela signifie que n_1 était un état final non profond de A_1 et donc l'état correspondant est final dans A_1.

A_2 simule A_1

On parcourt de même l'ensemble des possibilités d'exécuter une transition dans A_1. Il y en a également six :

- **Transition de type *local*** : Les seules transitions de A_1 qui sont modifiées sont les transitions qui partent ou arrivent à l'état n_1. Les autres transitions sont conservées à l'identique dans A_2. Elles peuvent être donc être exécutées dans les mêmes états. Deux types de transitions locales sont modifiées :
  - **Les transitions arrivant à** A_{1_1} : Dans la sémantique des ASTD, après une telle transition, l'ASTD est dans l'état initial de A_{1_1}. Dans A_2, ces transitions ont été transformées pour arriver à l'état correspondant à l'état initial de A_{1_1}.
  - **Les transitions partant de** A_{1_1} : D'après les conditions d'application de la règles, ces transitions sont des transitions d'état final. Elles ne peuvent être exécutées que si A_{1_1} est dans un état final. Or, dans A_2, ces transitions ont été transformées pour partir des états finaux de A_{1_1}.
- **Transition de type *toSub* (trois règles)** : Si ces transitions n'arrivent pas à un sous état de A_{1_1}, elles sont conservées dans A_2. D'après les conditions d'applications de la règle, si elles arrivent à un sous-état de A_{1_1}, ce sous état n'est pas un état historique. De plus la transition a été modifiée pour arriver à l'état de A_2 correspondant à l'état de A_{1_1}.
- **Transition de type *fromSub*** : Si les transitions ne partent pas d'un état de A_{1_1}, elles sont conservées dans A_2. Si elles partent d'un sous état de A_2, elles sont transformées en transition de type *local* et partent de l'état de A_2 correspondant à l'état de A_{1_1} dont elle partaient dans A_1.
- **Transition dans un sous-**ASTD : Les transitions qui pouvaient être exécutées dans un sous-ASTD autre que A_{1_1} ne sont pas modifiées dans A_2. Les transitions qui pouvaient être exécutées dans A_{1_1} sont soit des transitions de A_{1_1} et elles sont ajoutées aux transition de A_2, soit ce sont des transitions qui peuvent être

exécutées dans un sous-astd de A_{1_1} et les sous-astd de A_{1_1} sont ajoutés dans A_2.

Si l'état initial de A_1 n'est pas n_1, la preuve pour l'état initial est triviale. Si l'état initial est n_1, l'état initial de A_2 est l'état initial de A_{1_1}, et la relation est préservée. Pour les états finaux ne A_1 autres que n_1, la preuve est également triviale. Si n_1 est un état final non profond, tous les états de A_{1_1} sont ajoutées au états finaux de A_2. Si n_1 est un état final profond, seuls les états finaux sont ajoutés aux états finaux de A_2, en respectant le fait que ces états soient finaux profonds et finaux non profonds.

4.6 Conclusion

Dans ce chapitre, nous avons défini un ensemble de règles de réécriture pour les astd. Une esquisse de preuve nous a permis de nous convaincre que l'application de ces règles de réécriture à une spécification ne changeait pas le sens de cette spécification. Cependant, pour une assurance plus complète, une preuve plus rigoureuse serait nécessaire.

De plus, puisque nous définissons un système de réécriture, il est nécessaire de s'assurer de sa terminaison. Ici, la preuve de la terminaison du système de règles est assez triviale. En effet, toutes les règles produisent des automates. Après l'application d'une règle, les règles des sections 4.3 et 4.4 ne peuvent donc plus s'appliquer. De plus, puisque les règles suppriment chacune le niveau hiérarchique sur lequel elles s'appliquent, il n'est donc pas possible d'appliquer infiniment une même règle sur un astd donné. En combinant ces deux arguments, nous pouvons nous convaincre que l'application de système de réécriture sur n'importe quelle spécification termine.

Enfin, ces règles de réécriture nous amènent à réfléchir à la définition d'une éventuelle forme normale pour l'écriture des astd. Cependant, cette éventuelle recherche de forme normale dépasse le cadre de ce chapitre, dont l'objectif était uniquement de simplifier certaines spécification astd. De plus, d'une part nous ne sommes pas convaincu de l'existence d'une forme normale et d'autre part, si cette forme normale existe, la recherche de la forme normale et les preuves d'existence et d'unicité de cette forme normale seraient sans doute extrêmement complexes.

Chapitre 5

Cas d'étude

La méthode définie dans le chapitre 2 doit permettre de spécifier des systèmes industriels, notamment ferroviaires. Cette méthode a été mise en place en modélisant un système de contrôle de trains basé sur le principe du *CBTC*. La spécification de ce cas d'étude est présentée dans la section 5.1. Ce cas d'étude est également utilisé pour mesurer l'impact des améliorations proposées dans cette thèse. Il a donc été modélisé en utilisant le raffinement combiné présenté à la section 2.3.1 et en utilisant les règles de simplification définies dans le chapitre 4. Les résultats obtenus lors de ces différentes modélisations sont détaillés dans la section 5.1.4.

De plus, afin de mesurer l'intérêt de cette méthode sur d'autres types de systèmes, elle a été utilisée pour modéliser une machine à hémodialyse. Cette modélisation est une réponse au cas d'étude proposé pour la conférence ABZ2016 [Mas15]. Ce cas d'étude est présenté dans la section 5.2.

La modélisation de ces cas d'étude a parfois nécessité des notions qui ne sont pas définies dans la méthode ou des opérateurs qui n'étaient pas définis à l'origine dans le langage des ASTD. Nous avons accepté dans ces cas d'étendre certaines notions de la méthode et du langage ASTD. Ces extensions peuvent être vues comme des extensions qui n'existent pour le moment que dans les cas d'études présentés, mais qui pourraient être intégrées au langage si des modélisations futures montrent leur pertinence. Les extensions nécessaires à chaque cas sont introduites avant la présentation des modélisations.

5.1 Modélisation d'un système de contrôle automatique des trains

Le cas d'étude présenté dans cette section est le cas d'étude qui a permis de mettre en place la méthode présentée au chapitre 2. Dans ce cas d'étude, la synchronisation quantifiée conditionnelle est introduite. De même, lors de la modélisation du quatrième niveau de spécification, la définition du raffinement a été relâchée. Ces extensions sont présentées dans la section 5.1.1. Dans la section 5.1.2, nous présentons le cas d'étude et les hypothèses de travail. Les différents niveaux de la spécification sont présentés dans la section 5.1.3. Enfin, ce cas d'étude a été utilisé pour évaluer les améliorations apportées à la méthode. Les résultats de ces améliorations sont présentées à la section 5.1.4.

5.1.1 Notions spécifiques au cas d'étude

Synchronisation quantifiée conditionnelle

L'opérateur de synchronisation quantifiée conditionnelle (symbole $\pitchfork$) est une extension de l'opérateur de synchronisation quantifiée. Soit $\langle \pitchfork, n, x, T, \Delta, p, b \rangle$ l'ensemble des ASTD de type synchronisation quantifiée conditionnelle, où n est le nom de l'ASTD, x une variable choisie dans l'ensemble T, Δ est l'ensemble des étiquettes de transition qui sont synchronisées, p est un prédicat et b est un ASTD. Cet opérateur signifie que l'ASTD b est exécuté en autant d'instances x qu'il y a de valeurs dans l'ensemble T. Ces instances sont exécutées en entrelacement, mais doivent se synchroniser pour toutes les transitions dont l'étiquette se trouve dans l'ensemble Δ, sauf pour les instances pour lesquelles le prédicat p n'est pas vrai. Plus formellement, les règles d'inférences pour exécuter une transition dans un ASTD en synchronisation quantifiée conditionnelle sont les suivantes :

$$\pitchfork_1 \frac{\alpha(\sigma) \notin \Delta \qquad f(v) \xrightarrow{\sigma, (\![x := v]\!) \lhd \Gamma} s'}{(\pitchfork_\circ, f) \xrightarrow{\sigma,\Gamma} (\pitchfork_\circ, f \lhd\!\!+ \{x \mapsto s'\})}$$

$$\pitchfork_2 \frac{\alpha(\sigma) \in \Delta \qquad \forall v \in T.((\neg([x := v]p) \wedge f(v) = f'(v)) \vee (f(v) \xrightarrow{\sigma,(\![x:=v]\!)\lhd\Gamma} f'(v)))}{(\pitchfork_\circ, f) \xrightarrow{\sigma,\Gamma} (\pitchfork_\circ, f')}$$

La règle $\pitchfork_1$ décrit la possibilité d'exécuter une transition n'appartenant pas à l'ensemble de synchronisation Δ. La règle $\pitchfork_2$ décrit la possibilité d'exécuter une transition appartenant à l'ensemble de synchronisation Δ. Elle signifie qu'il est possible d'exécuter une telle transition si pour toutes les instances de l'ensemble de quantification T, soit le prédicat p est faux pour cette instance et l'état associé à l'instance ne change pas ($[x := v]p \wedge f(v) = f'(v)$), soit l'instance est dans un état permettant la transition, et la transition est alors exécutée ($f(v) \xrightarrow{\sigma,([x:=v])\triangleleft\Gamma} f'(v)$).

Cet opérateur permet de décrire le comportement de plusieurs instances fonctionnant en parallèle tout en évitant de créer des situations dans lesquelles des instances sont bloquées par d'autres pour exécuter une transition synchronisée. Dans la spécification de ce cas d'étude par exemple, nous modélisons un ensemble de trains qui peuvent être démarrés et arrêtés. Nous souhaitons alors que les trains arrêtés ne puissent pas bloquer les autres trains qui doivent exécuter une transition synchronisée. Pour un exemple plus précis, voir le troisième raffinement dans la section 5.1.3.

On remarquera que la synchronisation quantifiée des ASTD (dont les règles d'inférence sont données en Annexe A.7) est le cas particulier de la synchronisation quantifiée conditionnelle où le prédicat p de condition de synchronisation est le prédicat *true*, qui est toujours vrai.

Transformation d'un entrelacement en synchronisation

Dans le système que nous souhaitons modéliser, nous voulons transformer un système dans lequel plusieurs instances s'exécutent en parallèle en un système dans lequel les différentes instances sont synchronisées sur une transition. Soit un ASTD abstrait $A = \langle \pitchfork, n, x, T, \Delta, p, b \rangle$ de type synchronisation quantifiée conditionnelle et soit une étiquette de transition non synchronisée $\sigma(x)$ ($\sigma(x) \notin \Delta$) qui prend x en paramètre. Nous souhaitons transformer l'ASTD A en l'ASTD $C = \langle \pitchfork, n, T, \Delta \cup \sigma, p, b' \rangle$ où σ remplace toutes les transitions $\sigma(x)$ qui doivent être exécutées en même temps en se synchronisant. L'ASTD b' est donc l'ASTD b dans lequel toutes les transitions ayant pour étiquette $\sigma(x)$ ont été remplacées par une transition étiquetée σ.

Cette transformation ne vérifie pas la définition du raffinement des ASTD donnée à la section 2.3. En effet, dans la spécification abstraite, pour une instance v, il est possible d'exécuter la transition σ si l'instance v est dans un état qui permet d'exé-

cuter σ. Dans la spécification concrète, il faut attendre pour exécuter σ que toutes les instances qui doivent se synchroniser soient dans un état qui leur permet d'exécuter σ. Un certain nombre de traces ne sont donc pas préservées dans la spécification concrète.

Cependant, dans le cas où cette transformation est utilisée dans le cas d'étude présenté ci-après, pour toute instance devant se synchroniser (toute instance vérifiant le prédicat p), la spécification permet d'exécuter un cycle de transitions (une séquence de transition qui revient au même état) au cours duquel les variables d'état ne sont pas modifiées, et ce cycle contient la transition σ synchronisée. Dans ce cas, il est possible d'ajouter à toute trace un tel cycle de transitions sans véritablement changer le comportement observé. Si cette condition est vraie pour un système, il est possible de trouver une trace équivalente à toutes les traces abstraites dans les traces concrètes. On dit ici que deux traces sont équivalentes si quelque soit l'état des données avant l'exécution de chaque trace, les données sont dans le même état après l'exécution des deux traces.

5.1.2 Présentation

La spécification présentée dans cette section est une spécification d'un système de contrôle automatique des trains, basé sur le principe du *CBTC* (*Communication Based Train Control System*). Ce système est un système standard qui permet le contrôle de ligne de métro automatique. Le principe général de ce système est basé sur la communication entre deux sous-systèmes. Un système à bord est embarqué sur chaque train et permet d'en contrôler sa marche. Pour permettre ce contrôle en toute sécurité, le système à bord communique sa position à un système au sol qui, en fonction des positions connues des trains, calcule une limite que chaque train ne doit pas dépasser. Dans ce système, nous souhaitons vérifier que deux trains n'entrent jamais en collision.

Cette modélisation a été utilisée pour définir la méthode de spécification décrite au chapitre 2, c'est-à-dire définir les interactions entre la spécification de l'ordonnancement des actions et la spécification des changements sur les données du système. Pour cela, la partie données du système décrit a été abstraite. Nous considérons un

système composé d'une voie unique, sur laquelle le sens de mouvement des trains est unique. Cette approximation est une approximation forte d'un système ferroviaire, mais nous supposons qu'il est possible à chaque instant d'isoler une partie de voie sur laquelle cette hypothèse est vraie. Les évolutions de cette partie de voie pourraient alors être prises en compte lors de raffinement futur.

La voie est modélisée dans le contexte Event-B de la figure 5.1. Elle est représentée par un ensemble appelé *TRACK*. Cet ensemble est ordonné par une relation d'ordre stricte (c'est-à-dire irreflexive - `axm2` - transitive - `axm3` - et antisymétrique - `axm4`) totale (`axm5`) appelée *is_behind*. Pour deux éléments x_1 et x_2 de l'ensemble *TRACK*, $x_1 \mapsto x_2 \in is_behind$ signifie que l'élément x_1 est situé derrière l'élément x_2 sur la voie. Nous supposons également qu'il existe un élément $position_0$ situé au début de la voie (c'est-à-dire derrière tous les éléments de la voie). Dans ce contexte, nous souhaitons vérifier la propriété de non collision, c'est-à-dire que deux trains n'occupent jamais la même position sur la voie.

Le système est modélisé en plusieurs étapes de raffinement. Dans les deux premières spécifications, les trains bougent de manière autonome. La troisième spécification introduit une opération de contrôle qui calcule une limite de mouvement pour chaque train. Dans la quatrième spécification, les opérations qui calculent une limite à chaque train sont regroupées en une opération de contrôle unique qui calcule une limite pour tous les trains. La cinquième spécification introduit des opérations de communication qui permettent dans la sixième spécification de décomposer le système en deux sous-systèmes fonctionnant en parallèle.

5.1.3 Spécifications

Spécification abstraite

La spécification abstraite de l'ordonnancement des actions du système est donnée à la figure 5.2. À partir de son état initial (état S_1), un train peut démarrer. Il peut ensuite effectuer zéro, une ou plusieurs opérations de mouvement puis s'arrêter. Lorsqu'un train est arrêté (état S_3), il peut à nouveau démarrer grâce à l'opérateur de fermeture de Kleene. Enfin, le système contient plusieurs trains qui bougent en parallèle, ce qui est décrit par l'opérateur d'entrelacement.

CONTEXT TRACK_DEF

SETS

$TRACK$

$TRAIN$

CONSTANTS

is_behind

$position0$

AXIOMS

axm1 : $is_behind \in (TRACK \leftrightarrow TRACK)$

axm2 : $(\forall p1 \cdot (p1 \in TRACK \Rightarrow \neg(p1 \mapsto p1 \in is_behind)))$

axm3 : $\forall p1, p2, p3 \cdot (p1 \in TRACK \wedge p2 \in TRACK \wedge p3 \in TRACK \wedge p1 \mapsto p2 \in is_behind \wedge p2 \mapsto p3 \in is_behind \Rightarrow p1 \mapsto p3 \in is_behind)$

axm4 : $\forall p1, p2 \cdot (p1 \in TRACK \wedge p2 \in TRACK \wedge p1 \mapsto p2 \in is_behind \Rightarrow \neg(p2 \mapsto p1 \in is_behind))$

axm5 : $\forall p1, p2 \cdot (p1 \in TRACK \wedge p2 \in TRACK \wedge p1 \neq p2 \Rightarrow p1 \mapsto p2 \in is_behind \vee p2 \mapsto p1 \in is_behind)$

axm6 : $position0 \in TRACK$

axm7 : $\forall pp \cdot (pp \in TRACK \wedge pp \neq position0 \Rightarrow position0 \mapsto pp \in is_behind)$

END

figure 5.1 – Contexte Event-B définissant la voie

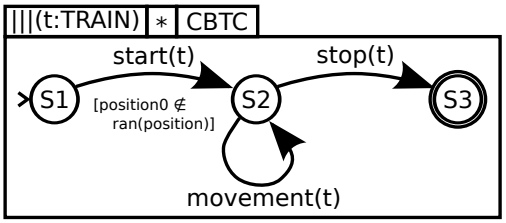


figure 5.2 – Spécification d'un système ferroviaire en utilisant les ASTD : Spécification abstraite

Afin de modéliser la propriété de non-collision, la partie données introduit une variable *position* qui est une fonction partielle qui associe un élément de la voie aux trains. Cette variable est modifiée par les événements Event-B associés à chaque étiquette de transition de la spécification ASTD. Dans la suite de cette modélisation l'événement Event-B associé à une étiquette de transition ASTD portera le nom de cette étiquette suivi du suffixe *_act*. Dans l'événement *start_act*, la position $position_0$ est associée au train démarré. C'est la raison pour laquelle la transition *start* comporte une garde qui vérifie que cette position n'est associée à aucun train. Dans l'événement *movement_act* une nouvelle position est associée au train qui bouge. L'événement *stop_act* retire le train de la liste des trains ayant une position.

La propriété de non collision peut alors être traduite par deux propriétés. La première vérifie que les positions de deux trains différents sont différentes. Cependant, cette propriété ne suffit pas si la position d'un train peut être modifiée d'une position à une position non adjacente. En effet, un train pourrait passer d'une position derrière un train à une position devant (ce qui signifierait alors que le train de derrière est passé à travers le train de devant, et qu'il y a eu collision). Il faut donc s'assurer également que l'ordre des trains ne change pas. La première propriété est exprimée par la proposition 5.1. Pour être exprimée, la deuxième propriété nécessite l'opérateur *next* (X) de la logique temporelle [Pnu81]. Cette condition est traduite par la proposition 5.2.

$$\begin{aligned}
\forall(t_1, t_2) \cdot (t_1 \in TRAIN \wedge \\
t_2 \in TRAIN \wedge \\
t_1 \neq t_2 \Rightarrow \\
position(t_1) \neq position(t_2)) \qquad (5.1)
\end{aligned}$$

$$\begin{aligned}
\forall(t_1, t_2).(t_1 \in TRAIN \wedge \\
t_2 \in TRAIN \wedge \\
position(t_1) \mapsto position(t_2) \in is_behind \Rightarrow \\
\mathsf{X}(position(t_1) \mapsto position(t_2) \in is_behind)) \qquad (5.2)
\end{aligned}$$

La proposition 5.1 est traduite directement sous la forme d'un invariant. La deuxième proposition ne peut être exprimée directement en Event-B. Cependant, il est possible de l'encoder sous la forme d'un théorème Event-B. L'idée de cet encodage est de transformer les événements sous la forme de prédicats *Before/After* et de réécrire la proposition 5.2 en remplaçant les variables de la proposition en paramètre de l'opérateur *next* par l'état des variables après et la variable du reste de la proposition par l'état avant.

Au cours du mouvement, la position du train est modifiée de manière à vérifier ces conditions. La nouvelle position du train est donc choisie telle que :

— la nouvelle position est différente des positions des autres trains ;

— la nouvelle position est derrière la position de tous les trains dont la position est devant la position actuelle du train.

De plus, puisque les trains se déplacent tous dans le même sens :

— la nouvelle position est devant la position actuelle du train ou ne change pas.

La spécification de l'événement *movement_act* est donnée à la figure 5.3. Pour mettre à jour la position du train selon ces conditions, le train doit avoir une position, ce qui se traduit par la garde gu2. Il faudra ensuite vérifier qu'à chaque fois qu'une transition *movement* est exécutée, le train pour lequel cette transition est exécutée a une position. Cette vérification se fait grâce à la vérification de la cohérence horizontale (voir la présentation de la méthode en section 2.1).

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

any

tt

where

gu1 : $tt \in TRAIN$

gu2 : $tt \in \mathrm{dom}(position)$

then

act1 : $position :\mid$

$(\exists pp \cdot (pp \in TRACK \land$

$position' = position \Leftarrow \{tt \mapsto pp\} \land$

$(\forall t2 \cdot (t2 \in dom(position') \land$

$t2 \neq tt \Rightarrow pp \neq position'(t2))) \land$

$(\forall t2 \cdot (t2 \in dom(position) \land$

$position(tt) \mapsto position(t2) \in behind \Rightarrow$

$pp \mapsto position(t2) \in behind)) \land$

$(pp = position(tt) \lor$

$position(tt) \mapsto pp \in behind)))$

end

figure 5.3 – Spécification Event-B abstraite de l'événement associé à la transition *movement*

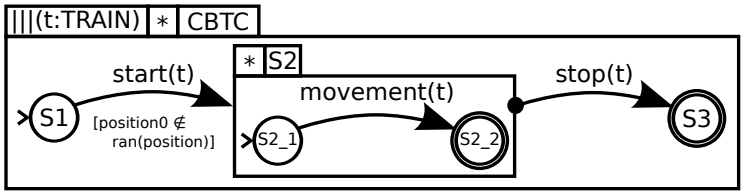


figure 5.4 – Spécification d'un système ferroviaire en utilisant les ASTD : Premier Raffinement

Premier raffinement

L'objectif de ce raffinement est de réécrire la spécification abstraite afin de faciliter les raffinements suivants. Dans les raffinements suivants, nous souhaitons développer la modélisation du mouvement en spécifiant comment la position peut être modifiée en vérifiant les propriétés de non-collision. La transition *movement* est donc transformée en fermeture de Kleene. La spécification ASTD de ce niveau de modélisation est donnée à la figure 5.4. Cette modélisation contient exactement les mêmes étiquettes de transition que la spécification abstraite. La spécification Event-B reste donc inchangée.

Deuxième raffinement

Le *CBTC* est un système de contrôle des trains basé sur la communication entre deux sous-systèmes. Le sous-système à bord conduit le train en respectant des consignes données par le sous-système au sol. Dans ce raffinement, nous commençons à introduire le sous-système au sol. À ce niveau, il est représenté par une opération de contrôle. Cette opération, spécifique à chaque train, calcule une limite que le train ne doit pas dépasser lorsque sa position change.

Cette opération de contrôle doit calculer une nouvelle limite à chaque fois que le train change de position. Dans la spécification ASTD, une transition *compute_l* est ajoutée après la transition *start* et après l'opération *movement*. La spécification ASTD est donnée à la figure 5.5.

Une variable appelée *mal* est ajoutée dans la modélisation de la partie données. Cette variable est une fonction partielle qui associe aux trains une position sur la

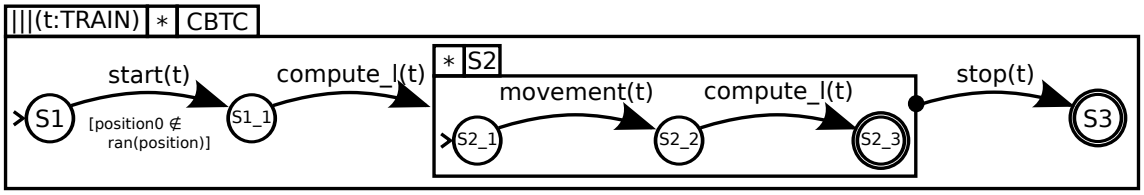


figure 5.5 – Spécification d'un système ferroviaire en utilisant les ASTD : Deuxième Raffinement

voie. Cette position correspond à la limite que le train ne doit pas dépasser au cours de son mouvement. La limite est calculée en respectant les conditions suivantes :

— la limite est située devant la position du train ou au niveau de la position du train ;

— la limite est située derrière la position de tous les trains dont la position est devant la position du train pour lequel la limite est calculée.

Au cours du mouvement, la position est maintenant modifiée en prenant en compte cette limite, de la manière suivante :

— la nouvelle position est située devant l'ancienne position ou au même niveau que l'ancienne position ;

— la nouvelle position est située derrière la limite du train.

Les spécifications des événements associés aux transitions *compute_l* et *movement* sont données à la figure 5.6.

Les propriétés 5.1 et 5.2 de non-collision peuvent être exprimées par les deux propriétés suivantes :

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

any

tt

where

gu1 : $tt \in TRAIN$

gu2 : $tt \in dom(position)$

then

act1 : $mal :\mid$

$(\exists mm \cdot (mm \in TRACK \land$

$\forall t2 \cdot (t2 \in TRAIN \land$

$t2 \in dom(position) \land$

$position(tt) \mapsto position(t2) \in behind \Rightarrow$

$mm \mapsto position(t2) \in behind) \land$

$(position(tt) \mapsto mm \in behind \lor$

$position(tt) = mm) \land$

$mal' = mal \lhd\!\!\!\!- \{tt \mapsto mm\}))$

end

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

refines $Movement_act$

any

tt

where

gu1 : $tt \in TRAIN$

gu2 : $tt \in dom(position)$

gu3 : $tt \in dom(mal)$

then

act1 : $position :\mid$

$(\exists pp \cdot (pp \in TRACK \land$

$position' = position \lhd\!\!\!\!- \{tt \mapsto pp\} \land$

$(position(tt) = mal(tt) \Rightarrow position(tt) = pp) \land$

$(\neg(position(tt) = mal(tt)) \Rightarrow (position(tt) \mapsto pp) \in behind) \land$

$(pp = mal(tt) \lor (pp \mapsto mal(tt)) \in behind)))$

end

figure 5.6 – Deuxième raffinement : Spécification Event-B des événements associés aux transitions *movement* et *compute_l*

$$\begin{aligned}
&\forall(t_1, t_2) \cdot (t_1 \in TRAIN \wedge \\
&\qquad t_2 \in TRAIN \wedge \\
&\qquad position(t_1) \mapsto position(t_2) \in is_behind \Rightarrow \\
&\qquad\qquad mal(t_1) \mapsto position(t_2) \in is_behind) \qquad (5.3) \\
&\forall t_1 \cdot (t_1 \in TRAIN \wedge \\
&\qquad t_1 \in \mathrm{dom}(mal) \wedge \\
&\qquad t_1 \in \mathrm{dom}(position) \Rightarrow \\
&\qquad\qquad (position(t_1) \mapsto mal(t_1) \in is_behind \vee \\
&\qquad\qquad\ position(t_1) = mal(t_1))) \qquad (5.4)
\end{aligned}$$

La propriété 5.3 signifie que la limite d'un train est toujours située derrière la position de tous les trains qui sont situés devant lui. La propriété 5.4 signifie que la position d'un train doit toujours être derrière sa limite ou au niveau de sa limite. Ces propriétés sont traduites sous la forme d'invariants.

Troisième raffinement

Dans la spécification du niveau précédent, la transition *compute_l* calcule indépendamment une limite pour chaque train. Or, dans le système *CBTC*, la limite est calculée par un sous-système qui calcule au même moment les limites autorisées pour tous les trains. Dans ce raffinement, nous regroupons les transitions *compute_l* en une transition *compute*. Cette transition calcule la limite de mouvement de tous les trains démarrés.

Pour calculer en même temps la limite pour tous les trains démarrés, cette transition doit être synchronisée. Ce raffinement n'est pas un raffinement classique mais applique la notion de raffinement définie au début de ce cas d'étude (voir section 5.1.1). L'idée de ce raffinement est de transformer une transition non synchronisée en transition synchronisée. Ici, la transition *compute_l* est transformée en transition *compute* et ajoutée à l'ensemble de synchronisation. L'entrelacement quantifié est transformé

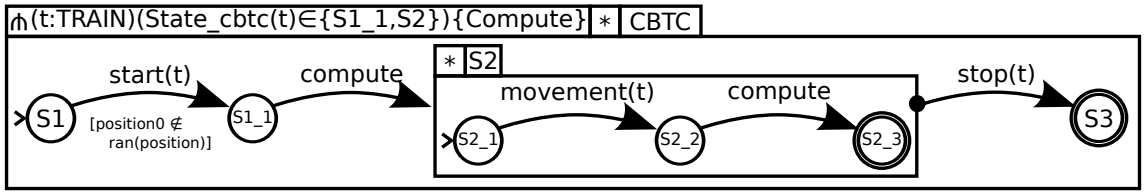


figure 5.7 – Spécification d'un système ferroviaire en utilisant les ASTD : Troisième Raffinement

en synchronisation quantifiée conditionnelle (voir section 5.1.1). La spécification ASTD est donnée à la figure 5.7.

Au cours de cette transformation, des traces admises par la spécification abstraite (la spécification de la figure 5.5) sont perdues. En effet, si l'ensemble *TRAIN* contient au moins deux trains t_1 et t_2, il est possible d'exécuter la trace suivante :

$$start(t_1); compute_l(t_1); movement(t_1); compute_l(t_1); start(t_2); \\ compute_l(t_2); movement(t_1)$$

alors que cette trace n'est pas acceptable dans la spécification de la figure 5.7. Cependant, puisqu'il est possible d'exécuter la transition *movement* en ne changeant pas la variable *position* et la transition *compute_l* en ne changeant pas la variable *mal*, il est possible d'exécuter les transitions *movement* et *compute_l* pour le train t_1 de telle manière que les variables d'état du système ne sont pas modifiées. La trace

$$start(t_1); compute_l(t_1); movement(t_1); compute_l(t_1); start(t_2); \\ movement(t_1); compute_l(t_1); compute_l(t_2); movement(t_1)$$

par exemple est une trace qui est préservée dans l'ASTD de la figure 5.7 (en remplaçant les étiquettes *compute_l* pour chaque train démarré par une étiquette *compute*) et qui est équivalente à la trace précédente (c'est-à-dire que l'état des données du système est le même après l'exécution de cette trace).

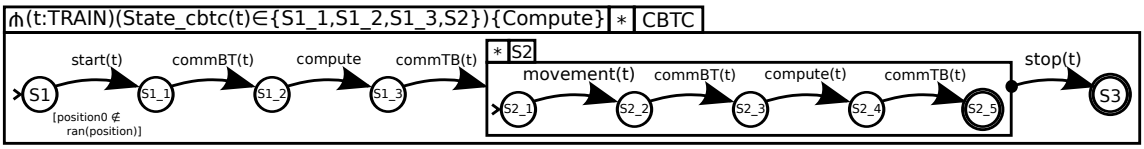


figure 5.8 – Spécification d'un système ferroviaire en utilisant les ASTD : Quatrième Raffinement

Quatrième raffinement

Nous souhaitons modéliser le système de contrôle des trains sous la forme de deux sous-systèmes fonctionnant en parallèle. Ces deux sous-systèmes ont chacun des transitions spécifiques et des variables spécifiques, qu'ils échangent grâce à des transitions de communication. Les transitions *movement* et *start* sont des transitions spécifiques au sous-système à bord du train. La transition *compute* est une transition spécifique au sous-système au sol. Après chaque transition spécifique au sous-système à bord, une transition de communication du bord vers le sol (appelée *commBT* pour *Communication Board Track*) transmet les nouvelles valeurs des variables modifiées par cette transition. De même, après chaque transition spécifique au sous-système au sol, une transition de communication du sol vers le bord (*commTB*) transmet les nouvelles valeurs des variables au sous-système à bord. La transition *stop* est une transition commune aux deux sous-systèmes (elle est donc considérée de la même manière que les transitions de communication). La spécification de ce niveau de modélisation est décrite à la figure 5.8.

Dans la modélisation des données, la variable *mal* et la variable *position* sont dupliquées. La variable *on_board_position* est la position du train vue par le sous-système à bord du train et la variable *track_position* est la position du train vue par le sous-système au sol. De même, la variable *track_mal* est la limite de mouvement vue par le sous-système au sol et la variable *on_board_mal* est la limite de mouvement vue par le sous-système à bord du train. Les événements associées aux transitions spécifiques au sous-système à bord du train (les événements *start_act* et *movement_act*) ne peuvent lire et modifier que les variables qu'elles peuvent voir (les variables *on_board_mal* et *on_board_position*). De même, l'événement *compute_act* ne peut lire et modifier que les variable *track_mal* et *track_position*. Les événements associés aux transitions de

Event $CommBT_act \widehat{=}$
any
tt
where
gu1 : $tt \in TRAIN$
gu2 : $tt \in dom(on_board_position)$
then
act1 : $track_position(tt) := on_board_position(tt)$
end

figure 5.9 – Spécification Event-B de l'événement associée à la transition *commBT*

communication permettent d'échanger les variables entre les deux sous-systèmes. À titre d'exemple, l'événement associé à l'étiquette de transition *commBT* est donné à la figure 5.9.

Enfin, pour prouver le raffinement Event-B, qui est nécessaire d'après la définition du raffinement (**Définition 3**), un invariant de collage doit relier les variables de la modélisation concrète aux variables de la modélisation abstraite. Dans le niveau de modélisation précédente, lorsque la variable *position* est utilisée, c'est en fait la position à bord du train (*on_board_position*). De même lorsque la limite de mouvement est utilisée, c'est en fait la limite au sol. Ceci est traduit par l'invariant de collage suivant :

$$position = on_board_position \wedge$$
$$mal = track_mal$$

Cinquième raffinement

Dans ce niveau de modélisation, le système modélisé à la figure 5.8 est décomposé en deux sous-systèmes fonctionnant en parallèle. Les transitions spécifiques au sous-système à bord (*movement* et *start*) sont dans un ASTD modélisant le comportement du sous-système à bord et la transition *compute* (spécifique au sous-système au sol)

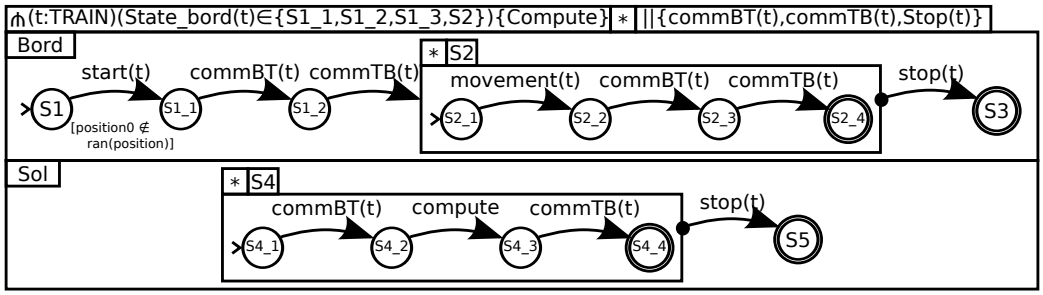


figure 5.10 – Spécification d'un système ferroviaire en utilisant les ASTD : Quatrième Raffinement

est dans un ASTD modélisant le comportement du sous-système au sol. Les transitions communes aux deux sous-systèmes (*commBT*, *commTB* et *stop*) sont synchronisées entre les deux ASTD. La spécification du système est décrite à la figure 5.10.

Cette spécification contient exactement les mêmes étiquettes de transition que la spécification précédente. La modélisation des données du système contient donc exactement les mêmes événements qui modifient les données de la même manière. La spécification Event-B associée est donc identique.

5.1.4 Comparaison des modélisations

La méthode décrite dans le chapitre 2 et utilisée pour modéliser ce cas d'étude nécessite de montrer la cohérence de chaque niveau de spécification. La preuve de cette cohérence génère un grand nombre d'obligations de preuve. Afin de mesurer la difficulté de ces preuves et les apports des améliorations apportées par les règles de réécriture, nous avons modélisé plusieurs fois ce cas d'étude. Dans un premier cas, nous avons spécifié naïvement ce cas d'étude en traduisant les ASTD selon les règles de traduction décrites dans la thèse de Milhau [Mil11]. Dans une deuxième modélisation, nous avons utilisé les règles de réécriture des ASTD décrites dans le chapitre 4 avant de les traduire. Dans un troisième cas, nous avons modifié les spécifications de donnée en utilisant la définition du raffinement de la section 2.3.2. Enfin, afin de situer ces modélisations, nous avons modélisé ce cas d'étude sans les règles de traduction. Ces différentes méthodes de modélisation sont décrites dans les sections ci-dessous.

Spécification classique

Dans cette spécification, nous avons utilisé naïvement les règles de traduction données dans la thèse de Milhau. L'idée de cette traduction est de modéliser les états des ASTD par des variables d'état en B. Une variable d'état est associée à chaque ASTD hiérarchique. Un outil de traduction automatique a été développé pour la traduction des ASTD en langage B. L'avantage de cette traduction est que la spécification en B colle parfaitement à la spécification graphique. Il est donc facile de se représenter l'état B dans lequel est la machine en fonction de l'état de l'ASTD. L'inconvénient principal de cette modélisation est qu'elle multiplie les variables d'état. Or, pour la preuve de la cohérence, il faut relier les variables d'état représentant l'état de l'ASTD aux variables d'état de la spécification de donnée. Plus il y a des variables d'état, plus il faut ajouter d'invariant pour faire la preuve complète de la machine.

Prenons par exemple la spécification abstraite du système ferroviaire. La spécification ASTD est visible à la figure 5.2. L'état de cet ASTD est donné par deux variables : *State_cbtc* et *State_kleene*. Puisque l'ASTD est un entrelacement quantifié, ces deux variables sont des fonctions totales qui associent un état à l'ensemble des variables de l'ensemble de quantification :

$$State_cbtc \in TRAIN \rightarrow \{S_1, S_2, S_3\}$$
$$State_kleene \in TRAIN \rightarrow \{started, notStarted\}$$

D'après les règles d'inférences de la fermeture de Kleene (voir annexe A.4), la transition *Start* peut être exécutée dans trois cas. Si l'ASTD est dans l'état S_1, si l'ASTD est dans l'état S_3 et quelque soit l'état de l'ASTD si l'état de la fermeture est *notStarted*. Dans la spécification de données, l'événement associé à la transition *Start* est gardé par le fait que le train pour lequel la transition est exécutée n'a pas de position ($tt \notin \text{dom}(position)$). Dans la spécification B qui permet de vérifier la cohérence de la spécification, il faut donc relier la variable *position* aux variables qui encodent l'état de l'ASTD. Il faut donc ajouter deux invariants : un premier invariant qui spécifie que les trains qui ont une position sont ceux qui sont dans l'état S_2 ($\text{dom}(position) = \text{dom}(State_cbtc \rhd \{S_2\})$) et un deuxième qui spécifie que si la fermeture de Kleene est

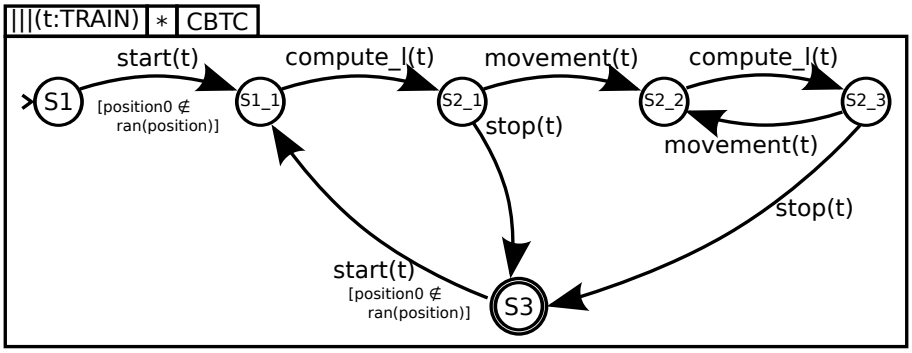


figure 5.11 – ASTD obtenu par la réécriture de l'ASTD 5.5

dans l'état *notStarted* alors il est dans l'état S_1 ($\mathrm{dom}(State_kleene \rhd \{notStarted\}) = \mathrm{dom}(State_cbtc \rhd \{S_1\})$).

Spécification obtenue avec les règles de simplification (Chapitre 4)

L'objectif des règles de simplification décrites dans le chapitre 4 est de réduire le nombre d'ASTD hiérarchiques, pour réduire le nombre de variables nécessaires pour décrire en B l'état d'un ASTD. Dans cette modélisation, avant de traduire en B les ASTD de la spécification, nous avons appliqué les règles de simplification sur les ASTD. La spécification ASTD du deuxième raffinement par exemple (voir figure 5.5) est réécrite par l'ASTD de la figure 5.11.

La spécification obtenue n'est pas vraiment plus lisible. La lisibilité d'une spécification est une qualité difficilement mesurable mais on peut admettre ici qu'il est plus compliqué de voir les cycles de transitions qu'il est possible d'exécuter sur la spécification de la figure 5.11 que sur la figure 5.5. Cependant, l'état d'un ASTD de la spécification de la figure 5.11 n'est caractérisé que par une seule variable d'état en B tandis que l'état d'un ASTD de la figure 5.5 nécessite quatre variables d'état : les états possibles de l'automate de S_2, les états de la fermeture de Kleene de cet automate, les états de l'automate global et les états de la fermeture de Kleene de l'automate global. Les chiffres illustrant la simplification sont donnés ci-dessous (section 5.1.4).

Spécification obtenue avec la définition du raffinement étendu pour simplifier la preuve (voir section 2.3.2)

Dans la méthode, nous avons défini une méthode de raffinement qui permet de simplifier les preuves de la cohérence. La preuve de la cohérence d'un niveau de raffinement consiste à montrer que lorsqu'une transition doit être exécutée, alors la garde de l'événement associé est vraie. L'idée de la méthode de raffinement est la suivante : puisque le comportement de l'ASTD concret est un comportement qui était admis par l'ASTD abstrait, et puisqu'on a prouvé la cohérence pour l'ASTD abstrait, il n'est plus nécessaire de montrer la cohérence pour l'ASTD concret dans le cas où les gardes ne changent pas. Dans cette section, nous expliquons comment appliquer ce raffinement en nous intéressant à la relation de raffinement entre le premier et le deuxième raffinement. À ce niveau de raffinement, une transition *compute_l(t)* permet de calculer une limite de mouvement pour chaque train. Cette limite de mouvement est stockée dans une variable appelée *mal*. Dans l'événement associé à la transition *movement(t)*, cette limite est utilisée pour déterminer la nouvelle position du train. Les deux spécifications de l'événement associé à cette transition sont données à la figure 5.12.

Dans l'événement associé à la transition *Movement* du troisième raffinement, la garde concernant la variable *position* ne change pas. La garde $tt \in \mathrm{dom}(position)$ de l'événement concret peut donc être déduite de l'événement abstrait. Pour montrer la cohérence du niveau concret, on peut donc retirer cette garde de l'événement concret et prouver la cohérence avec comme garde $tt \in \mathrm{dom}(mal)$.

Puisque la relation de raffinement entre le troisième raffinement et le quatrième raffinement ne suit pas la définition exacte du raffinement défini dans le chapitre 2, la méthode du raffinement étendu n'a pas été employée entre ces deux niveaux.

Spécification sans les règles de traduction

Afin d'avoir une idée du coût de l'utilisation des règles de traduction systématiques, nous avons choisi de spécifier également ce cas d'étude sans utiliser ces règles. Dans cette modélisation, les ASTD sont utilisés comme une spécification formelle de niveau supérieur. On suppose qu'ils correspondent à la spécification du comportement

Event *Movement_act* $\widehat{=}$
  any
    tt
  where
    gu1 : $tt \in TRAIN$
    gu2 : $tt \in dom(position)$
  then
    act1 : $position : |(\ldots)$
  end

(a) *Movement_act* : premier raffinement

Event *Movement_act* $\widehat{=}$
refines *Movement_act*
  any
    tt
  where
    gu1 : $tt \in TRAIN$
    gu2 : $tt \in dom(position)$
    gu3 : $tt \in dom(mal)$
  then
    act1 : $position : |(\ldots)$
  end

(b) *Movement_act* : deuxième raffinement

figure 5.12 – Événement associé à la transition *Movement* dans les deuxième et troisième raffinements

attendu d'un système que nous devons modéliser en B. Comme pour la spécification classique, une opération est créée pour chaque étiquette de transition et l'événement correspondant est appelé dans le corps de la transition.

Dans cette méthode de spécification, la spécification du comportement du système a été faite à la main. Aucun outil de traduction des ASTD a été utilisé, ce qui a permis de simplifier la modélisation du comportement beaucoup plus que si la traduction était automatique. Cependant, puisque cette spécification a été faite à la main, il n'est pas possible de prouver que la spécification B est équivalente à la spécification ASTD de départ, alors que lorsque la spécification est traduite automatiquement, si l'outil de traduction a été validé, la confiance dans la traduction obtenue est beaucoup plus grande.

Comparaison des spécifications

La tableau 5.1 résume le nombre d'obligations nécessaires pour la preuve de chaque niveau de spécification dans les 4 méthodes de spécification décrites ci-dessus. Pour chaque niveau, trois chiffres sont donnés. La colonne *tot* contient le nombre total

	Classique			Réécriture			Raffinement étendu			Spécification à la main		
	tot	man	usr	tot	man	usr	tot	man	usr	tot	man	usr
N 1	37	5	3	21	1	1	21	1	1	13	2	2
N 2	87	5	3	30	3	2	18	0	0	13	1	1
N 3	121	26	11	53	13	6	51	12	6	41	7	5
N 4	97	31	14	40	16	9	-	-	-	28	12	10
N 5	237	110	42	119	66	38	115	65	38	86	43	34
N 6	396	190	64	202	69	40	54	0	0	100	43	35
TOT	**975**	**367**	**137**	**465**	**168**	**96**	**299**	**94**	**54**	**281**	**108**	**87**

tableau 5.1 – Comparaison des obligations de preuves générées pour les différentes modélisations

d'obligations de preuve générées pour chaque niveau. Certaines de ces obligations de preuves sont prouvées automatiquement par les prouveurs de l'atelier B. La colonne *man* contient le nombre d'obligations de preuves qu'il est nécessaire de prouver manuellement à l'aide du prouveur interactif de l'atelier B. Enfin, lorsqu'une obligation de preuve est prouvée à l'aide du prouveur interactif, il est possible de sauvegarder la tactique de preuve et de l'utiliser pour essayer de prouver d'autres obligations. La colonne *usr* contient le nombre de tactiques différentes qu'il a fallu utiliser pour prouver la machine. Par exemple, pour prouver le niveau 6 de la spécification classique, 396 obligations de preuve ont été générées, 190 ont été prouvées manuellement à l'aide du prouveur interactif. Cependant, pour prouver ces 190 obligations, seules 64 tactiques ont été nécessaires.

On remarque d'abord que le nombre d'obligations de preuve générées pour prouver la cohérence de la spécification dans le cas de la spécification naïve est beaucoup plus important que dans le cas de la spécification traduite manuellement. En effet, la traduction automatique impose de générer du code qui pourrait être simplifié en traduisant intelligemment. Cela se retrouve par le fait que le nombre de tactique nécessaires à la preuve ne croît pas autant que le nombre total d'obligations de preuve. La seconde remarque est que, dans notre cas d'étude, les règles de réécriture permettent de diminuer significativement le coût du travail de preuve.

Enfin, on remarque que l'apport du raffinement étendu est important sur les niveaux 2 et 6. Les spécifications des niveaux 2 et 6 sont les spécifications dans lesquelles seul le comportement du système est raffiné (la spécification des données du système ne change pas). Dans ce cas, il est évidemment possible de retirer toutes les gardes, ce qui explique que la machine est entièrement prouvée automatiquement (les quelques obligations qui restent à prouver sont les obligations associées à la vérification des types).

Cependant, l'apport de cette relation de raffinement étendu est assez faible sur les autres niveaux. Pour prouver la cohérence du système, il est nécessaire d'ajouter des invariants qui lient les variables d'état du système à certaines variables d'état qui représentent les états de l'ASTD. Cet invariant doit alors être montré pour chaque opération modifiant la variable d'état du système ou modifiant la variable d'état représentant les états de l'ASTD. C'est la preuve de l'invariant qui génère autant d'obligations de preuve.

Dans le cas de la spécification de niveau 3, par exemple, on ajoute à la spécification de données la variable *mal* et dans la spécification du comportement, on ajoute une opération *compute_l*. Les événements associés aux transitions *Start* et *Movement* ont une garde qui dépend de la variable *position* qui est identique à la garde abstraite. Cette garde peut donc être retirée dans la preuve de la cohérence. Cependant, dans la garde de l'événement associé à la transition *compute_l*, il faut vérifier que le train pour lequel on calcule une limite a une position ($t \in \mathrm{dom}(position)$). Il faut donc quand même relier la variable *position* aux variables d'état modélisant l'état de l'ASTD. Le cas où le raffinement étendu est le plus efficace est le cas où les événements associés aux nouvelles transitions ajoutées au cours du raffinement ne modifient et ne lisent que des nouvelles variables ajoutées au cours du raffinement.

5.2 Modélisation d'une machine à hémodialyse

Le cas d'étude est une réponse au cas d'étude proposé dans le cadre de la conférence ABZ'16 [Mas15] dont l'objectif est de modéliser une machine à hémodialyse. La machine a été modélisée en utilisant la méthode décrite dans le chapitre 2 [FFGL16]. Dans le document décrivant le cas d'étude, le fonctionnement de la machine est décrit

en deux parties principales. La première partie décrit le comportement général de la machine et la seconde partie présente les exigences de sécurité, sans lien évident entre les deux. Notre spécification se concentre donc sur la modélisation du comportement de la machine.

5.2.1 Notions spécifiques

Pour simplifier l'écriture des spécifications dans ce cas d'étude, un opérateur de séquence quantifiée est ajouté. L'idée de cette opérateur est de permettre d'exécuter une séquence pour plusieurs valeurs dans un ensemble de quantification. Plus formellement, soit $\langle \Rightarrow_s, name, x, E, a(x) \rangle$ un ASTD de type séquence quantifiée, où $name$ est un nom d'ASTD, x une variable et E une liste de valeurs (la liste de valeurs est décrite ici comme une liste des langages fonctionnels) et $a(x)$ un ASTD. La sémantique de cet opérateur n'est pas donnée par une règle d'inférence, mais par une fonction de réécriture qui permet de la transformer en une séquence classique. Cette fonction de réécriture r est décrite de la manière suivante :

$$r(\langle \Rightarrow_s, name, x, [v], a(x) \rangle) \triangleq a(v) \tag{5.5}$$

$$r(\langle \Rightarrow_s, name, x, h :: t, a(x) \rangle) \triangleq \langle \Rightarrow, name_h, a(h), r(\Rightarrow_s, name, x, t, a(x)) \rangle \tag{5.6}$$

La règle 5.5 signifie qu'exécuter une séquence quantifiée dont l'ensemble de quantification ne contient qu'une instance consiste à exécuter l'ASTD a pour cette instance. La règle 5.6 signifie que la réécriture d'une séquence quantifiée dont la liste de quantification est la liste $h :: t$ (la liste $h :: t$ est une liste constituée de l'instance h en tête et de la liste t en queue) est une séquence dont le premier ASTD est l'ASTD a pour l'instance h et dont le deuxième ASTD est la réécriture de la séquence quantifiée avec comme ensemble de quantification la liste t.

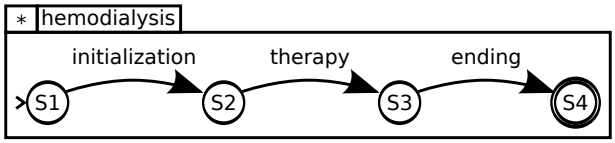


figure 5.13 – Spécification abstraite

5.2.2 Spécifications

Spécification abstraite

Dans la description la plus générale du système, le fonctionnement de la machine à hémodialyse est décrit comme la succession de trois phases principales. Lors de la première phase, la machine est initialisée. Après cette initialisation, le sang peut être filtré lors de la phase de thérapie, Enfin, une phase permet de réinjecter le sang filtré dans le corps du patient. Ce fonctionnement est représenté par un ASTD à quatre états et trois transitions. Après la phase de réinfusion du sang, la machine peut être réutilisée pour une nouvelle hémodialyse, ce qui est modélisé par une fermeture de Kleene. La spécification abstraite complète est à la figure 5.13.

Dans la partie données de la modélisation, une variable d'état appelée *state* représente l'état de la machine. Cette variable est mise à jour lors de chaque transition. Avant l'initialisation de la machine, la variable est à *OFF*. Après l'initialisation, sa valeur devient *INITIALISED*, puis *HD_COMPLETE* après la dialyse. Cette mise à jour est modélisée dans les événements associés à chaque transition. L'événement d'initialisation de la machine par exemple est gardé par le fait que l'état de la machine est à *OFF* puis change l'état de la machine pour *INITIALISED*. L'événement complet est à la figure 5.14

Pour prouver la cohérence de la spécification, il est nécessaire de lier les variables d'état modélisant l'ASTD aux variables d'état décrivant les données du système. Ici, il faut lier la variable *state* à la variable *State_hemodialysis* qui décrit l'état de l'ASTD. Par exemple, pour prouver que la garde de l'événement associé à la transition *initialisation* est vraie lorsque la transition *initialisation* est exécutée, il faut ajouter l'invariant suivant :

Event $initialisation_act \mathrel{\widehat{=}}$
  when
    gu1 : $state = OFF$
  then
    act1 : $state := INITIALISED$
  end

figure 5.14 – Spécification Event-B de l'événement associé à la transition *initialisation*

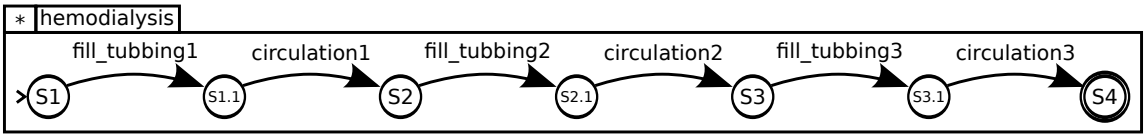


figure 5.15 – Modélisation de la machine à hémodialyse

$$State_hemodialysis \in \{S_1, S_2\} \Rightarrow state = OFF$$

Premier raffinement

Chacune des trois phases de l'hémodialyse est divisée en deux étapes. Lors de l'initialisation, les tubes de la machine sont remplis d'eau, puis l'eau circule dans le système pour rincer la machine. Durant la phase de thérapie, le système est rempli de sang puis le sang circule pour être filtré. Enfin, durant la phase de fin, le système est rempli d'eau puis l'eau circule dans le système afin de réinjecter l'ensemble du sang. La spécification ASTD de ce niveau de modélisation est à la figure 5.15.

Un capteur permet de déclencher les phases de circulation lorsque du liquide est détecté. Une variable *detected_liquid* modélise ce capteur. Le liquide détecté est soit de l'eau (*WATER*), soit du sang (*BLOOD*), soit rien (*NOTHING*). Les événements associés aux transitions $fill_tubbing_i$ changent les valeurs de la variable *detected_liquid*. Les événement associés aux transitions $circulation_i$ raffinement les événements associés aux transition de la spécification abstraite (on montre alors que ce raffinement suit la définition du raffinement défini dans la section 2.3.1).

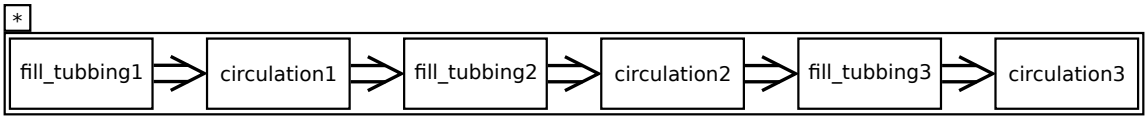


figure 5.16 – Modélisation de la machine à hémodialyse - Deuxième raffinement - Spécification principale

Dans l'état initial, la machine est à l'état *OFF* et le système est vide ($detected_liquid = NOTHING$). Lors de la transition $fill_tubbing_1$, le système est rempli d'eau. Dans l'événement associé à la transition $circulation_1$, l'état de la machine est mis à la valeur *INITIALIZED* (ce qui ne modifie pas la spécification de l'événement abstrait qu'il raffine). Lors de l'événement associé à la transition $fill_tubbing_2$, le liquide détecté devient du sang, puis dans l'événement associé à la transition $circulation_2$, l'état de la machine devient *HD_COMPLETE*. Enfin, dans l'événement associé à la transition $fill_tubbing_3$, le liquide détecté devient de l'eau et lors de l'événement associé à la transition $circulation_3$, l'état de la machine passe à *OFF* et la machine est vidée ($detected_liquid := NOTHING$).

Deuxième raffinement

Dans ce niveau de spécification, l'objectif et de décrire plus précisément ce qui se passe lors de chaque phase correspondant aux six transitions de la spécification précédente. Cependant, d'après le document de spécification, chacune de ces phases est une séquence de plusieurs actions. Le développement de chacune de ces actions donnerait un ASTD plat de plusieurs dizaines de transitions en séquence, ce qui serait illisible. Pour cela, chacune des transitions est remplacée par une séquence d'ASTD, qui sont définis ailleurs et qui sont appelés grâce à l'opérateur d'appel (*call*) du langage ASTD. La spécification ASTD est à la figure 5.16.

Dans la suite, nous détaillons les spécifications des deux premiers ASTD. L'ASTD $fill_tubbing_1$ correspond à la séquence d'événement durant l'initialisation pendant laquelle le système est rempli d'eau. D'abord, une solution d'eau salée est connectée à la machine. Ensuite, l'opérateur appuie sur le bouton de démarrage de la machine, ce qui permet de démarrer la pompe. Pendant que la pompe fonctionne, la machine lit le capteur *VRD* (*Venous Red Detector*). Lorsque de l'eau est détectée par ce capteur,

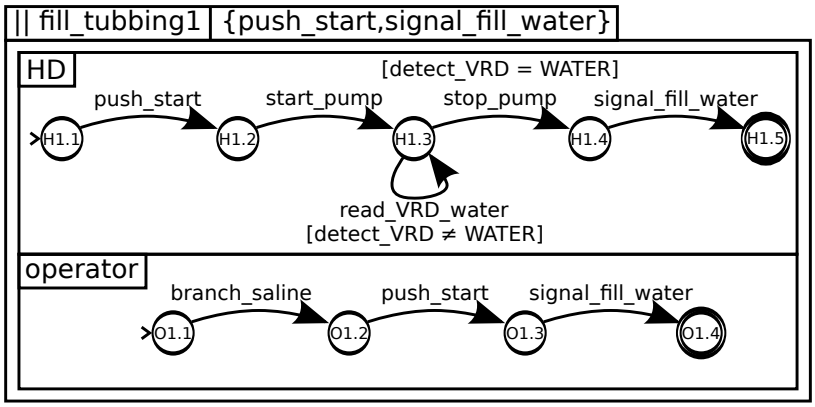


figure 5.17 – Modélisation de la machine à hémodialyse - Deuxième raffinement - Spécification de l'ASTD $fill_tubbing_1$

la pompe s'arrête. Il est alors inscrit que dans ce cas, la solution saline est retirée et que la sortie veineuse et la sortie artérielle du système sont connectées ensemble. Nous supposons donc que l'opérateur a été averti que la pompe s'est arrêtée et que le système est rempli d'eau.

Le comportement décrit ci-dessus est effectuée par deux entités, un opérateur humain et la machine à hémodialyse. Ces deux entités sont représentées par deux ASTD synchronisés. L'opérateur connecte la solution d'eau salée à la machine. La machine à hémodialyse démarre la pompe, lit le capteur et stoppe la pompe une fois que de l'eau est détectée. Les autres transitions (appuyer sur le bouton de démarrage et signaler que la pompe s'est arrêtée) sont des transitions synchronisées qui permettent aux deux entités d'échanger des informations. La spécification de l'ASTD $fill_tubbing_1$ est donnée à la figure 5.17.

L'ASTD $circulation_1$ correspond à l'étape pendant laquelle la machine à hémodialyse est rincée. Lorsque de l'eau est détectée par le capteur *VRD*, les extrémités artérielle et veineuse sont branchées l'une avec l'autre par l'opérateur. L'opérateur appuie sur le bouton de démarrage puis la machine démarre la pompe. Le document de spécification ne précise pas à quel moment la pompe s'arrête. Nous supposerons ici que la machine est rincée pendant un certain temps. Après ce temps de rinçage, la pompe s'arrête et une lampe jaune s'allume pour signaler que la machine est rincée. Ce comportement nécessite à nouveau la collaboration de deux entités, l'opérateur et

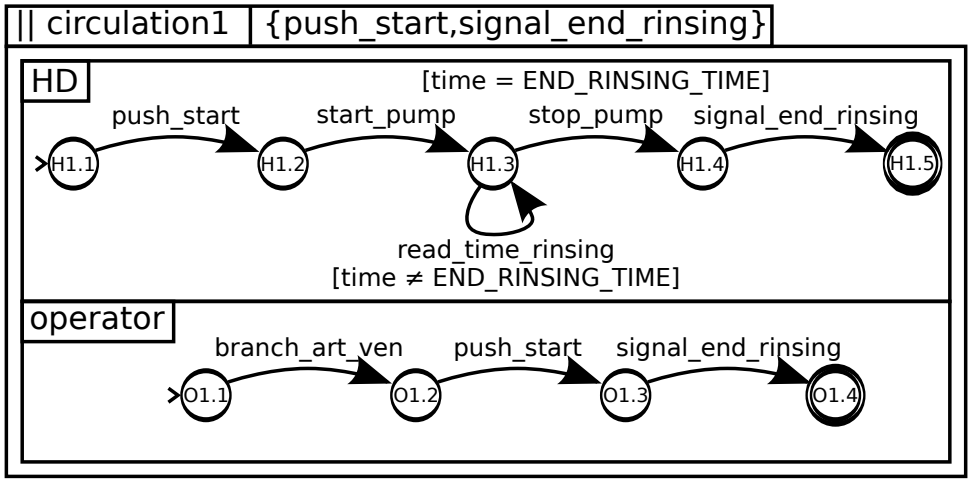


figure 5.18 – Modélisation d'un machine à hémodialyse - Deuxième raffinement - Spécification de l'ASTD $circulation_1$

la machine à hémodialyse. L'opérateur branche les extrémités, la machine démarre la pompe, lit le temps et arrête la pompe lorsque le temps est écoulé. La transition pendant laquelle l'opérateur appuie sur le bouton et celle pendant laquelle la lumière s'allume sont des opérations synchronisées. Elles permettent aux deux entités de communiquer. La spécification de l'ASTD est visible à la figure 5.18.

Les ASTD $fill_tubbing_2$ et $fill_tubbing_3$ suivent des spécifications proches de celle de l'ASTD $fill_tubbing_1$. Cependant, dans l'ASTD $fill_tubbing_2$, l'opérateur branche l'extrémité artérielle au patient, la machine lit le capteur *VRD* en attendant de trouver du sang (et plus de l'eau), et la machine signale que le système est rempli du sang du patient. Dans l'ASTD $fill_tubbing_3$, l'opérateur branche une solution saline à l'extrémité artérielle, la machine lit le capteur *VRD* en attendant de détecter de l'eau, et la machine signale alors que le sang a été réinjecté dans le patient. Les autres transitions restent inchangées.

De même, les ASTD $circulation_2$ et $circulation_3$ suivent des spécifications proches de celle de l'ASTD $circulation_1$. Dans l'ASTD $circulation_2$, l'opérateur branche l'extrémité veineuse au patient. De même, aucune condition d'arrêt de la dialyse n'est précisée dans le document, nous supposons donc que la dialyse est arrêtée après un temps donné. La machine lit le temps et arrête la pompe lorsque ce temps est atteint.

La machine signale alors à l'opérateur la fin de la dialyse. Dans l'ASTD $circulation_3$, l'extrémité veineuse est retirée du patient. La machine lit alors le capteur *VRD* jusqu'à ce qu'il ne détecte plus rien (la machine est alors vide). Un signal permet à l'opérateur de savoir que la machine est vide.

Plusieurs variables sont ajoutées pour modéliser les comportements décrits ci-dessus. Une variable permet de modéliser ce que détecte le capteur *VRD*, une variable permet de détecter quand cela est nécessaire, une variable modélise le fait que la pompe est démarrée ou non, etc... Le raffinement Event-B, les événements associés aux dernières transitions de chaque ASTD (les transitions *signal_...*) raffinent les événements associés aux transitions de l'ASTD de la spécification précédente (l'ASTD de la figure 5.15). L'événement associé à la transition *signal_fill_water* de l'ASTD de la figure 5.17 raffine l'événement associé à la transition $circulation_1$ de l'ASTD de la figure 5.15 par exemple.

Factorisation de la spécification ASTD

Dans la spécification précédente, on remarque que les comportements décrits par les ASTD des figures 5.17 et 5.18 sont similaires. L'ordre des transitions ne change pas et la spécification des événements associés à chaque transition ne change que pour trois transitions :

— Dans la transition pendant laquelle l'opérateur branche les extrémités, ce à quoi l'opérateur branche les extrémités change selon l'étape (solution saline, patient, etc...

— Dans la transition de lecture de la condition d'arrêt, ce qui est lu dépend de l'étape (capteur *VRD*, temps, etc...)

— Dans la transition qui signale la fin de la séquence, le signal dépend de l'étape (machine rincée, hémodialyse terminée, etc...)

Dans la section précédente, seuls deux ASTD étaient présentés alors que la spécification ASTD principale (la spécification de la figure 5.16) en contient six. La spécification globale est donc beaucoup plus grande que celle présentée. Afin de réduire cette spécification, nous souhaitons utiliser le fait que les spécifications se ressemblent et factoriser ces spécifications. La spécification de la figure 5.13 décrit trois phases

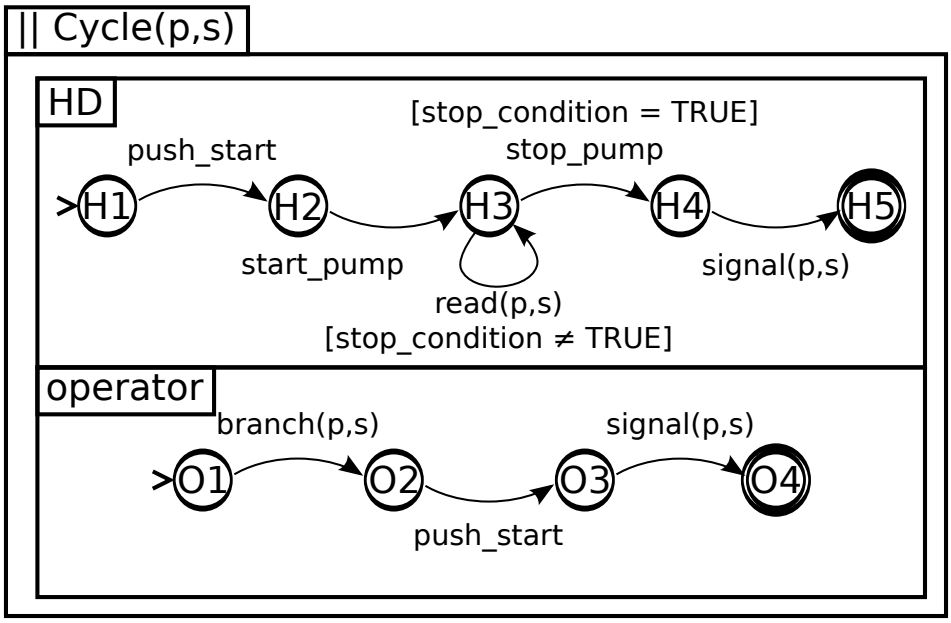


figure 5.19 – Spécification de l'ASTD *cycle*

dans le fonctionnement de la machine : l'initialisation (*initialization*), la thérapie (*therapy*) et la fin de thérapie (*ending*). Ces trois phases sont représentées par une variable $p \in \{INIT, THREAPY, ENDING\}$. Chacune de ces phases est séparée en deux étapes : le remplissage du circuit, et la circulation du liquide. Elles sont représentées par une variable $s \in \{FILL, CIRC\}$. Les six ASTD sont alors remplacés par un ASTD unique qui utilise ces deux paramètres pour spécifier les comportements particuliers de chaque ASTD. Les opérations communes à tous les ASTD (*push_start*, *start_pump* et *stop_pump*) ne changent pas, les autres opérations sont paramétrées en fonction de la phase et de l'étape. La spécification se trouve à la figure 5.19.

Afin de simplifier l'écriture de la spécification principale, nous utilisons un opérateur de séquence quantifiée. L'idée de cet opérateur est de permettre d'exécuter une séquence plusieurs fois pour un ensemble de valeurs. Les valeurs pour lesquelles la séquence est exécutée sont données sous la forme d'une liste ordonnée. La spécification est donnée à la figure 5.19. Elle signifie que la séquence constituée de deux ASTD *cycle* doit être exécutée pour trois valeur de la variable p : d'abord pour p valant *INIT* puis *THERAPY* puis *ENDING*.

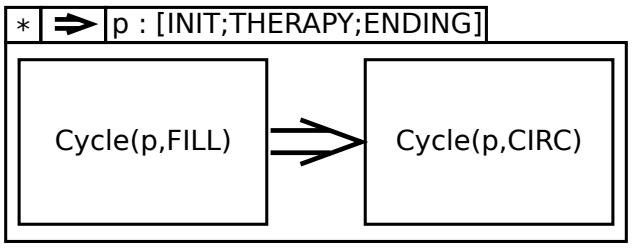


figure 5.20 – Spécification principale

La spécification de données permet de différencier les comportements spécifiques de chaque étape et de chaque phase. La spécification Event-B de la l'événement *branch_act* est donnés à la Figure 5.21 à titre d'exemple. Elle permet de décrire ce qui doit être branché en fonction de l'étape et de la phase. Par exemple, lorsque la machine est dans l'étape de remplissage lors de la phase de thérapie ($p = THERAPY$ et $s = FILL$), l'opérateur ne doit pas changer le branchement de l'extrémité veineuse du système et doit brancher l'extrémité artérielle au patient.

5.2.3 Conclusion

Dans ce chapitre, nous présentons la spécification d'une machine à hémodialyse en utilisant la combinaison des méthodes ASTD et Event-B décrite dans le chapitre 2. Le document qui présentait la machine et qui en donnait les exigences est un document écrit en langage naturel. Il comportait donc des imprécisions et des ambiguïtés. Nous ne pouvions donc pas décrire fidèlement le fonctionnement d'une machine à hémodialyse et vérifier des propriétés de sécurité fortes. Cependant, la spécification ASTD permet de décrire formellement et graphiquement le fonctionnement d'une machine à hémodialyse. De plus, certains opérateurs permettent de simplifier cette spécification en la factorisant. Les spécifications obtenues peuvent permettre ensuite de valider progressivement le fonctionnement d'une machine à hémodialyse en discutant de ces spécifications avec des experts du domaine. Dans ce cas d'étude, les ASTD sont utilisés comme une spécification de haut niveau qui permet de valider des spécifications avec un expert du domaine ne connaissant pas les langages formels.

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

any

p

s

where

gu1 : $p \in PHASES$

gu2 : $s \in STEPS$

then

act1 : $branch_venous, branch_arterial :\mid($

$(p = INIT \wedge s = FILL \Rightarrow$
$branch_venous' = branch_venous \wedge branch_arterial' = WATER) \wedge$
$(p = INIT \wedge s = CIRC \Rightarrow$
$branch_venous' = TUBBING \wedge branch_arterial' = TUBBING) \wedge$
$(p = THERAPY \wedge s = FILL \Rightarrow$
$branch_venous' = branch_venous \wedge branch_arterial' = PATIENT) \wedge$
$(p = THERAPY \wedge s = CIRC \Rightarrow$
$branch_venous' = PATIENT \wedge branch_arterial' = branch_arterial) \wedge$
$(p = ENDING \wedge s = FILL \Rightarrow$
$branch_venous' = branch_venous \wedge branch_arterial' = WATER) \wedge$
$(p = ENDING \wedge s = CIRC \Rightarrow$
$branch_venous' = NOTHING \wedge branch_arterial' = NOTHING))$

end

figure 5.21 – Spécification Event-B de l'événement *branch_act*

De plus, dans cette spécification, plusieurs opérations sont exécutées par un humain. Cette modélisation permet donc d'avoir une spécification formelle de ce que doit faire un opérateur pour utiliser correctement une machine à hémodialyse. Cette spécification formelle peut ensuite être utilisée pour écrire un manuel utilisateur par exemple.

Conclusion

L'objectif de cette thèse est de proposer une méthode de spécification qui permet l'utilisation du langage ASTD pour la spécification de systèmes industriels. Le langage ASTD est un langage de spécification basé sur les états. Il permet donc d'exprimer plus facilement des propriétés d'ordonnancement, mais la spécification du modèle de données est plus difficile. Pour permettre la spécification complète d'un système industriel, le langage ASTD est combiné avec le langage Event-B. Enfin, le langage B permet de vérifier la cohérence de la spécification (c'est-à-dire vérifier que les actions du système s'exécutent dans l'ordre spécifié par l'ASTD et que l'évolution des données est conforme à la spécification Event-B). La méthode est utilisée sur deux cas d'études (la modélisation d'un système ferroviaire et la modélisation d'une machine à hémodialyse).

Synthèse des contributions

La principale contribution de cette thèse est la définition d'une méthode formelle de spécification combinant un langage de spécification basé sur les états et un langage de spécification basé sur les événements. Les propriétés que cette méthode doit vérifier sont formellement définies : il s'agit de vérifier que les gardes des événements nécessaires à la préservation de invariants dans le modèle de données sont vraies lorsque la spécification du comportement dynamique du système impose l'exécution d'un événement. De plus, un raffinement combiné du langage ASTD et du langage Event-B permet de spécifier les systèmes incrémentalement (chapitre 2).

La vérification de la cohérence des spécifications nécessite de traduire les spécifications ASTD dans le langage B. Pour que la méthode définie dans cette thèse soit

formelle, l'implémentation d'une partie des règles de traduction a été prouvée avec l'assistant de preuve Coq. Cette traduction permet de générer un outil de traduction validé d'une sous-partie du langage ASTD vers le langage B (chapitre 3).

Pour faciliter l'utilisation de la méthode définie dans ce document, des règles de simplification des ASTD sont définies. Les règles de simplification sont un catalogue non exhaustif de règles jugées utiles pour les spécifications. Elles sont accompagnées d'une esquisse de leur preuve de validité, contenant les arguments principaux pour s'assurer de la correction de ces règles (chapitre 4).

Enfin, la méthode est mise en œuvre pour la spécification d'un système ferroviaire et pour la modélisation d'une machine à hémodialyse. La spécification du système ferroviaire est la spécification qui a permis de construire la méthode. Elle a fait l'objet d'une publication pour le workshop Refine2015 [FFLG16]. La spécification de la machine à hémodialyse est une réponse au cas d'étude proposé dans le cadre de la conférence ABZ2016 [Mas15, FFGL16] (chapitre 5).

La modélisation du système ferroviaire a également permis de mesurer les apports de la méthode. Une étude comparative de ces apports est détaillée dans la section 5.1.4, et notamment dans le tableau 5.1. Cette comparaison montre qu'avec les améliorations proposées dans cette thèse, l'utilisation de la méthode n'augmente pas significativement le nombre d'obligations de preuve générées, alors qu'elle apporte plus de garanties.

Perspectives

Dans cette thèse, nous avons défini une méthode de spécification qui permet l'utilisation des ASTD pour la spécification de systèmes industriels. Le principal avantage du langage ASTD est de permettre de spécifier des systèmes complexes en utilisant un langage graphique. Cependant, pour le moment, aucun éditeur graphique n'est disponible pour le langage ASTD. De même, seule une partie de l'outil de traduction des ASTD vers le langage B est actuellement prouvée avec Coq. Dans l'ensemble donc, pour être mise en place dans le cadre de projets industriels, un meilleur outillage de la méthode de spécification est nécessaire.

Ensuite, la méthode définie permet l'utilisation du raffinement, afin de pouvoir spécifier incrémentallement les systèmes modélisés. Ce raffinement est basé sur un raffinement combiné du langage ASTD et du langage Event-B. La preuve du raffinement Event-B est pris en charge par l'outil Rodin, mais aucun outil ne permet générer les obligations de preuve du raffinement des ASTD. Une solution envisagée pourrait être d'utiliser la traduction en B des ASTD pour vérifier ce raffinement, en générant automatiquement les invariants de collage à partir de l'éditeur des ASTD.

De plus, l'objectif de cette thèse est définir une méthode de spécification pour la modélisation de systèmes industriels. Deux cas d'études ont été réalisés pour mesurer l'intérêt de cette méthode de spécification. Cependant, ces deux cas de spécification ne correspondent pas à des cas réellement industriels. Pour valider véritablement la méthode de spécification, il serait intéressant de l'utiliser pour la modélisation d'un cas industriel réel.

Enfin, la méthode telle qu'elle est définie actuellement ne permet de modéliser qu'un système avec un ASTD et sa spécification Event-B associée. Afin de faciliter la modélisation de système plus complexes, il serait intéressant de pouvoir découper le système à modéliser en plusieurs parties grâce à l'utilisation de la modularité. Il faudrait pour cela mesurer l'impact de la modularité sur les preuves de cohérence de la méthode.

Annexe A

Sémantique opérationnelle des ASTD

La sémantique opérationnelle des ASTD [FGLF08] est définie à l'aide d'un système de transition étiquetté. Un système de transition étiquetté est un sous ensemble de $\mathsf{State} \times \mathsf{Event} \times \mathsf{State}$, où State est l'ensemble des états d'un ASTD et Event est l'ensemble des événements possible dans un ASTD, écrit $s \xrightarrow{\sigma}_A s'$. $s \xrightarrow{\sigma}_A s'$ signifie qu'un événement étiquetté σ permet de passer de l'état s à l'état s' dans l'ASTD A. Si le contexte est suffisamment clair, l'indice A peut être retiré.

A.1 Automates

Un automate est décrit par un nom *name*, un ensemble d'événement Σ, un ensemble N de nom correspondant aux états de l'automate, une fonction ν qui associe à chaque nom d'état l'ASTD correspondant, un ensemble δ de transitions, deux ensembles *SF* et *DF* qui contiennent l'ensemble des états finaux et l'ensemble des états finaux profonds ainsi qu'un nom d'état initial *init*. Il est noté

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

Les états d'un automate sont décrits par un nom d'état n, une fonction h qui associe un état historique à chaque nom d'état et un sous-état s correspondant à l'état du sous-astd. Il est noté

$$\langle \mathsf{aut}_\circ, n, h, s \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle des automates sont les suivantes :

$$\mathsf{aut}_1 \ \frac{\delta((\mathsf{loc}, n_1, n_2), \sigma', g, \mathit{final}?) \qquad \Psi}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', \mathit{init}(\nu(n_2)))}$$

$$\mathsf{aut}_2 \ \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, n_{2_\flat}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} \notin \{\mathsf{H}, \mathsf{H}^*\} \\ \Psi\end{array}}{\begin{array}{l}(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} \\ \qquad (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, \mathit{init}(\nu(n_{2_\flat}))))\end{array}}$$

$$\mathsf{aut}_3 \ \frac{\begin{array}{c}\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}), \sigma', g, \mathit{final}?) \\ n_{2_\flat} = \mathit{name}(h(n_2)) \\ \Psi\end{array}}{\begin{array}{l}(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} \\ \qquad (\mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, \mathit{init}(\nu(n_{2_\flat}))))\end{array}}$$

$$\mathsf{aut}_4 \ \frac{\delta((\mathsf{tsub}, n_1, n_2, \mathsf{H}^*), \sigma', g, \mathit{final}?) \qquad \Psi}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', h(n_2))}$$

$$\mathsf{aut}_5 \ \frac{\begin{array}{c}\delta((\mathsf{fsub}, n_1, n_{1_\flat}, n_2), \sigma', g, \mathit{final}?) \\ \mathit{name}(s) = n_{1_\flat} \\ \Psi\end{array}}{(\mathsf{aut}_\circ, n_1, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n_2, h', \mathit{init}(\nu(n_2)))}$$

$$\mathsf{aut}_6 \ \frac{s \xrightarrow{\sigma,\Gamma}_{\nu(n)} s'}{(\mathsf{aut}_\circ, n, h, s) \xrightarrow{\sigma,\Gamma} (\mathsf{aut}_\circ, n, h, s')}$$

avec

$$\Psi \triangleq ((\mathit{final}? \Rightarrow \mathit{final}(s)) \wedge g \wedge \sigma' = \sigma \wedge h' = h \lhd\!\!+ \{n_1 \mapsto s\})$$

A.2 Sequence

Un ASTD de type séquence est décrite par un nom *name* et par deux ASTD *fst* et *snd*. L'ASTD *fst* est l'ASTD exécuté en premier dans la séquence, l'ASTD *snd* est l'ASTD exécuté en second dans la séquence. Une séquence est notée :

$$\langle \fatsemi, name, fst, snd \rangle$$

L'état d'une séquence est décrit par son état, qui peut être fst si l'ASTD est en train d'exécuter des transitions dans le premier ASTD ou snd si l'ASTD est en train d'exécuter des transitions dans le deuxième ASTD, et par l'état courant du sous-ASTD. Il est noté :

$$\langle \fatsemi_\circ, \mathsf{fst}, s \rangle \text{ ou}$$
$$\langle \fatsemi_\circ, \mathsf{snd}, s \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle des ASTD de type séquence sont les suivantes :

$$\fatsemi_1 \frac{s \xrightarrow{\sigma,\Gamma}_{fst} s'}{(\fatsemi_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\fatsemi_\circ, \mathsf{fst}, s')}$$

$$\fatsemi_2 \frac{final(s)[\Gamma] \qquad init(snd) \xrightarrow{\sigma,\Gamma}_{snd} s'}{(\fatsemi_\circ, \mathsf{fst}, s) \xrightarrow{\sigma,\Gamma} (\fatsemi_\circ, \mathsf{snd}, s')}$$

$$\fatsemi_3 \frac{s \xrightarrow{\sigma,\Gamma}_{snd} s'}{(\fatsemi_\circ, \mathsf{snd}, s) \xrightarrow{\sigma,\Gamma} (\fatsemi_\circ, \mathsf{snd}, s')}$$

A.3 Choice

Un ASTD de type choix est décrit par un nom *name* et par deux ASTD *left* et *right*. L'un de ces deux ASTD au choix est exécuté. Un ASTD de type choix est noté :

$$\langle |, name, left, right \rangle$$

L'état d'un astd de type choix est décrit par son état, qui peut être left si l'astd est en train d'exécuter des transitions dans l'astd de gauche, right si l'astd est en train d'exécuter des transitions dans l'astd de droite, ou $\perp$ si aucun astd n'a encore été choisi, et par l'état courant du sous-astd (ou $\perp$ si aucun astd n'a été choisi). Il est noté :

$$\langle |_\circ, \perp, \perp \rangle \text{ ou}$$
$$\langle |_\circ, \mathsf{left}, s \rangle \text{ ou}$$
$$\langle |_\circ, \mathsf{right}, s \rangle \text{ ou}$$

Les règles d'inférence qui définissent la sémantique opérationnelle des astd de type choix sont les suivantes :

$$|_1 \frac{init(l) \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \perp, \perp) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')}$$
$$|_2 \frac{init(r) \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \perp, \perp) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$
$$|_3 \frac{s \xrightarrow{\sigma,\Gamma}_l s'}{(|_\circ, \mathsf{left}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{left}, s')}$$
$$|_4 \frac{s \xrightarrow{\sigma,\Gamma}_r s'}{(|_\circ, \mathsf{right}, s) \xrightarrow{\sigma,\Gamma} (|_\circ, \mathsf{right}, s')}$$

A.4 Kleene closure

Une fermeture de Kllene est décrite par un nom *name* et par un astd a, qui peut être exécutée zéro, une ou plusieurs fois. Un astd de type fermeture de Kleene est noté :

$$\langle \star, name, a \rangle$$

L'état d'une fermeture de Kleene est décrit par une valeur booléenne *started*?, qui est vraie si la fermeture de Kleene a démarré une exécution, et un état qui correspond à l'état courant du sous-ASTD. Il est noté :

$$\langle \star_\circ, \mathit{started}?, s \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle des fermetures de Kleene sont les suivantes :

$$\star_1 \frac{(\mathit{final}(s)[\Gamma]_\vee \neg \mathit{started}?) \qquad \mathit{init}(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ, \mathit{started}?, s) \xrightarrow{\sigma,\Gamma} (\star_\circ, \mathbf{true}, s')}$$

$$\star_2 \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\star_\circ, \mathbf{true}, s) \xrightarrow{\sigma,\Gamma} (\star_\circ, \mathbf{true}, s')}$$

A.5 Parameterized synchronization

Une synchronisation est décrite par un nom *name*, par un ensemble d'étiquette de synchronisation Δ et par deux ASTD *left* et *right*. Les deux ASTD sont exécutés en entrelacement mais doivent se synchroniser pour exécuter toute les transitions dont l'étiquette est dans l'ensemble Δ. Il est noté :

$$\langle |[]|, \mathit{name}, \Delta, \mathit{left}, \mathit{right} \rangle$$

L'état d'une synchronisation est décrit par les états courant des deux sous-ASTD à gauche et à droite. Il est noté :

$$\langle |[]|_\circ, s_l, s_r \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle de la synchronisation sont les suivantes :

$$|[]|_1 \frac{\alpha(\sigma) \notin \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s'_l}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s'_l, s_r)}$$

$$|[]|_2 \frac{\alpha(\sigma) \notin \Delta \qquad s_r \xrightarrow{\sigma,\Gamma}_r s'_r}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s_l, s'_r)}$$

$$|[]|_3 \frac{\alpha(\sigma) \in \Delta \qquad s_l \xrightarrow{\sigma,\Gamma}_l s'_l \qquad s_r \xrightarrow{\sigma,\Gamma}_r s'_r}{(|[]|_\circ, s_l, s_r) \xrightarrow{\sigma,\Gamma} (|[]|_\circ, s'_l, s'_r)}$$

A.6 Quantified choice

Un choix quantifié est décrit par un nom *name*, par une variable de quantification x, un ensemble de quantification T et par un astd a. L'astd a est exécuté pour une valeur de x choisie dans T. Un astd de type choix quantifié est noté :

$$\langle |:, name, x, T, a \rangle$$

L'état d'un choix quantifié est décrit par la variable choisie et par l'état courant du sous-astd. Si aucune variable n'a été choisie, la variable choisie et l'état courant sont à $\perp$. L'état d'un choix quantifié est noté :

$$\langle |:_\circ, \perp, \perp \rangle \text{ ou}$$
$$\langle |:_\circ, v, s \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle du choix quantifié sont les suivantes :

$$|:_1 \frac{init(b) \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b s' \qquad v \in T}{(|:_\circ, \perp, \perp) \xrightarrow{\sigma,\Gamma} (|:_\circ, v, s')}$$

$$|:_2 \frac{s \xrightarrow{\sigma,([x:=v])\lhd\Gamma}_b s' \qquad v \neq \perp}{(|:_\circ, v, s) \xrightarrow{\sigma,\Gamma} (|:_\circ, v, s')}$$

A.7 Quantified Synchronization

Une synchronisation quantifiée est décrite par un nom *name*, par une variable de quantification x, par un ensemble de quantification T, par un ensemble de quantification Δ et par un astd a. L'astd a est exécuté en entrelacement pour chaque

variable x de l'ensemble de quantification T. Les instances de l'ASTD se synchronisent cependant pour exécuter les transitions dont l'étiquette est dans l'ensemble Δ. Une synchronisation quantifiée est notée :

$$\langle |[]|_{:}, name, x, T, \Delta, a \rangle$$

L'état d'une synchronisation quantifiée est une décrit par une fonction f qui associe un état de du sous-ASTD a à chaque variable de l'ensemble de quantification. Il est noté :

$$\langle |[]|_{:\circ}, f \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle de la synchronisation quantifiée sont les suivantes :

$$|[]|_{:1} \frac{\alpha(\sigma) \notin \Delta \qquad f(v) \xrightarrow{\sigma,([x:=v])\triangleleft\Gamma}_b s'}{(|[]|_{:\circ}, f) \xrightarrow{\sigma,\Gamma} (|[]|_{:\circ}, f \lhd\!\!+ \{v \mapsto s'\})}$$

$$|[]|_{:2} \frac{\alpha(\sigma) \in \Delta \qquad \forall v : T \cdot f(v) \xrightarrow{\sigma,([x:=v])\triangleleft\Gamma}_b f'(v)}{(|[]|_{:\circ}, f) \xrightarrow{\sigma,\Gamma} (|[]|_{:\circ}, f')}$$

A.8 Guard

Un ASTD de type garde est décrit par un nom *name*, par un prédicat g et par ASTD a. L'ASTD a ne peut être exécuté que si la garde est vérifiée. Il est noté :

$$\langle \Rightarrow, name, g, a \rangle$$

L'état d'un ASTD de type garde est décrit par une valeur booléenne *started ?* qui décrit si l'ASTD a démarré et par l'état courrant du sous-ASTD. Il est noté :

$$\langle \Rightarrow_\circ, started?, s \rangle$$

Les règles d'inférence qui définissent la sémantique opérationnelle des ASTD de type garde sont les suivantes :

$$\Rightarrow_1 \frac{g[\Gamma] \qquad init(b) \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ, \mathbf{false}, init(b)) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathbf{true}, s')}$$

$$\Rightarrow_2 \frac{s \xrightarrow{\sigma,\Gamma}_b s'}{(\Rightarrow_\circ, \mathbf{true}, s) \xrightarrow{\sigma,\Gamma} (\Rightarrow_\circ, \mathbf{true}, s')}$$

Bibliographie

[Abr96] J.-R. Abrial. The B-book : assigning programs to meanings. Cambridge University Press, 1996.

[Abr07] J.-R. Abrial. The Event-B Book. Cambridge University Press, 2007.

[Art] R Arthan. Proof power reference page. http://www.lemma-one.com/ProofPower/index/index.html.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor : A Successful Application of B in a Large Project. In JeannetteM. Wing, Jim Woodcock, and Jim Davies, editors, FM'99 — Formal Methods, volume 1708 of Lecture Notes in Computer Science, pages 369–387. Springer Berlin Heidelberg, 1999.

[Bjø06] D. Bjørner. Software Engineering, vol. 3 : Domains, Requirements and Software Design. Theoretical Computer Science. Springer, 2006.

[Boi14] E. A. Boiten. Introducing extra operations in refinement. Formal Aspects of Computing, 26(2) :305–317, 2014.

[But00] Michael Butler. csp2B : A practical approach to combining CSP and B. Formal Aspects of Computing, 12(3) :182–198, 2000.

[CEN01] CENELEC. EN-50128 : Applications ferroviaires - système de signalisation, de télécommunication et de traitement - logiciels pour systèmes de commande et de protection ferroviaire. 2001.

[Cle] Clearsy. B Language Reference Manual, 1.8.8 edition.

[Coq] The Coq Proof Assistant. http://coq.inria.fr.

[CSW03] A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for circus. Formal Aspects of Computing, 15(2-3) :146–181, 2003.

[DB01] John Derrick and Eerke A. Boiten. Refinement in Z and Object-Z. Springer London, 2001.

[Dol06] D. Dollé. Vital software : Formal method and coded processor. 2006. http://www.clearsy.com/pdf/VitalSoftware.pdf.

[Fay14] T. Fayolle. Specifying a Train System Using ASTD and the B Method. Technical report, 2014. http://www.lacl.fr/~tfayolle.

[FC06] A. Freitas and A. Cavalcanti. Automatic translation from circus to java. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, FM 2006 : Formal Methods, volume 4085 of Lecture Notes in Computer Science, pages 115–130. Springer Berlin Heidelberg, 2006.

[FFGL16] Thomas Fayolle, Marc Frappier, Frédéric Gervais, and Régine Laleau. Modelling a Hemodialysis Machine Using Algebraic State-Transition Diagrams and B-like Methods, pages 394–408. Springer International Publishing, Cham, 2016.

[FFLG16] Thomas Fayolle, Marc Frappier, Régine Laleau, and Frédéric Gervais. Formal refinement of extended state machines. In John Derrick, Eerke Boiten, and Steve Reeves, editors, Proceedings 17th International Workshop on Refinement, Oslo, Norway, 22nd June 2015, volume 209 of Electronic Proceedings in Theoretical Computer Science, pages 1–16. Open Publishing Association, 2016.

[FGL+08] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. Saint-Denis. Extending Statecharts with Process Algebra Operators. Inovation in System Software Engeneering, Volume 4, Number 3 :285–292, 2008.

[FGLF08] M. Frappier, F. Gervais, R. Laleau, and B. Fraikin. Algebraic State Transition Diagrams. Technical report, Université de Sherbrooke, 2008. http://www.dmi.usherb.ca/~frappier/Papers/astd.pdf.

[FGLM14] M. Frappier, F. Gervais, R. Laleau, and J. Milhau. Refinement patterns for ASTDs. Formal Aspects of Computing, 26(5) :919–941, 2014.

[FSD03] M. Frappier and R. St-Denis. EB3 : an entity-based black-box specification method for information systems. Software and Systems Modeling, 2(2) :134–149, July 2003.

[FSMM14] A. Ferrari, G.. Spagnolo, G. Martelli, and S. Menabeni. From commercial documents to system requirements : an approach for the engineering of novel CBTC solutions. International Journal on Software Tools for Technology Transfer, pages 1–21, 2014.

[GH08] C. George and A. E. Haxthausen. The Logic of the RAISE Specification Language. In Dines Bjørner and MartinC. Henson, editors, Logics of Specification Languages, Monographs in Theoretical Computer Science, pages 349–399. Springer Berlin Heidelberg, 2008.

[Har87] D. Harel. Statecharts : A Visual Formalism for Complex Systems. Science of Computer Programming, 8 :231–274, 1987.

[HJ98] C.A.R. Hoare and H. Jifeng. Unifying Theories of Programming. Prentice Hall series in computer science. Prentice Hall, 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. Commun. ACM, 21(8) :666–677, August 1978.

[HPK11] A. E. Haxthausen, J. Peleska, and S. Kinder. A formal approach for the construction and verification of railway control systems. Formal Aspects of Computing, 23(2) :191–219, 2011.

[IEC14] IEC. Railway applications - Urban guided transport management and command/control systems - Part 1 : System principles and fundamental concepts, 2014.

[IEE] IEEE Standard 1474. IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements : IEEE Std 1474.1-2004 ; IEEE Standard for User Interface Requirements in Communications-Based Train Control (CBTC) Systems : IEEE Std 1474.2-2003 ; IEEE Recommended Practice for Communications-Based Train Control (CBTC) System Design and Functional Allocations : IEEE Std 1474.3-2008.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. Commun. ACM, 52(7) :107–115, July 2009.

[Mas15] Atif Mashkoor. The hemodialysis case study. 2015.

[MGLF12] J. Milhau, F. Gervais, R. Laleau, and M. Frapier. Refinement pattern for ASTD. In Workshop on UML and Formal Methods, 2012.

[Mil11] J. Milhau. A formal integration of access control policies into information systems. Theses, Université Paris-Est ; Université de Sherbrooke, December 2011.

[Pnu81] A. Pnueli. The Temporal Semantics of Concurrent Programs. Theoretical Computer Science, 13 :45–60, 1981.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In Tools and Algorithms for the Construction and Analysis of Systems, pages 151–166. Springer Nature, 1998.

[RHB97] A. W. Roscoe, C. A. R. Hoare, and R. Bird. The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[SB06] Colin Snook and Michael Butler. Uml-b : Formal modeling and design aided by uml. ACM Trans. Softw. Eng. Methodol., 15(1) :92–122, January 2006.

[Sil12] R. Silva. Thesis, chapter Chapter 6 : Case Study, pages 121–160. 2012. http://eprints.soton.ac.uk/340237/.

[Spi89] J. M. Spivey. The Z Notation : A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[ST02] S. Schneider and H. Treharne. Communicating B machines. In Proceedings of the 2Nd International Conference of B and Z Users on Formal Specification and Development in Z and B, ZB '02, pages 416–435, London, UK, UK, 2002. Springer-Verlag.

[STW14] S. Schneider, H. Treharne, and H. Wehrheim. The behavioural semantics of Event-B refinement. Formal Aspects of Computing, 26(2) :251–280, 2014.

[WC02] J. Woodcock and A. Cavalcanti. The semantics of Circus. In Didier Bert, JonathanP. Bowen, MartinC. Henson, and Ken Robinson, editors, ZB 2002 :Formal Specification and Development in Z and B, volume 2272 of Lecture Notes in Computer Science, pages 184–203. Springer Berlin Heidelberg, 2002.