Développement formel de systèmes automatisés

Olfa Mosbahi Mosbahi-Khalgui

Institut National
Polytechnique de Lorraine

Faculté Des Sciences de Tunis
Université Tunis-El Manar

Département de formation doctorale en informatique

École doctorale IAEM Lorraine

Développement formel des systèmes automatisés

THÈSE

présentée et soutenue publiquement le 21 février 2008

pour l'obtention du

Doctorat de
l'Institut National Polytechnique de Lorraine
l'Université Tunis-El Manar
(spécialité informatique)

par

Olfa MOSBAHI

Composition du jury

Président :	Stephan Merz	Directeur de recherche, INRIA, LORIA
Rapporteurs :	Jacques Julliand	Professeur, Université de Franche-comté
	Riadh Robbana	Maître de conférence, HDR, Ecole polytechnique de Tunis
Examinateurs :	Jacques Jaray	Professeur, Institut National Polytechnique de Lorraine
	Samir Ben Ahmed	Professeur, Faculté des Sciences de Tunis
	Nejib Ben Hadj Alouane	Maître de conférence, HDR, ENSI de Tunis

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thloria.

Remerciements

J'adresse mes plus vifs remerciements à mes directeurs de thèse M. Jacques Jaray, professeur à l'INPL, et M. Samir Ben Ahmed, professeur à la FST, pour leur encadrement de qualité dont il m'ont fait bénéficier aimablement, pour leur disponibilité et leur soutien permanents, pour la patience avec laquelle ils ont suivi et corrigé mon travail. Qu'ils trouvent ici l'expression de ma profonde gratitude et de ma reconnaissance.

Je tiens également à remercier M. Dominique Mery, professeur à l'UHP, responsable de l'équipe MOSEL pour m'avoir accueillie au sein de son équipe, pour m'avoir procuré d'excellentes conditions de travail ainsi que pour ses encouragements.

J'exprime mes sincères remerciements à M. Stephan Merz, directeur de recherche INRIA, pour ses conseils judicieux, ainsi que ma gratitude pour son aide et son soutien qu'il a bien voulu manifester à mon égard, me prodiguant les bienveillantes directives dont j'avais besoin pour mener à bien ce travail.

Je tiens à remercier Mme Leila Jemni Ben Ayed d'avoir participé à l'encadrement de la thèse et dont je suis reconnaissante, pour son soutien, ses conseils et ses remarques intéressantes.

Je remercie vivement M. Jacques Julliand et M. Riadh Robbana pour leur judicieux travail d'évaluation de la thèse, leurs commentaires qui ont contribué à l'amélioration de ce document et leurs remarques très constructives. Je remercie aussi M. Nejib Ben Hadj-Alouane d'avoir examiné ce travail et pour ses intéressantes remarques dont je dois prendre en compte dans mes futurs travaux.

Je tiens à remercier tous les membres de l'équipe MOSEL, avec qui travailler fut toujours un plaisir en raison de leur disponibilité et de leur convivialité. Merci à Dominique Cansell, Joris Rehm, Loïc Fejoz et Nazim Benaïssa. Merci aussi aux assistantes Josiane Reffort et Roxane Auclair pour leurs services administratifs de qualité et à Nadine Beurné pour son aide dans le règlement des problèmes administratifs.

Toutes les personnes m'ayant permis de mener à bien ce travail sont assurées de ma gratitude.

Je dédie ce travail,
à mes parents...
à mon mari Mohamed...
à mon Cher Mohamed Aziz...
à toute la famille...

Résumé

Le travail de thèse présente une méthode de développement de systèmes automatisés basée sur les méthodes formelles B et TLA$^+$. Le développement par raffinement est au cœur de la méthode proposée. Un système automatisé est modélisé par deux composants, un contrôlé formé par le dispositif physique et son environnement et un contrôleur pilotant ce dernier. Il est exprimé par un produit synchronisé sur les actions de ces deux composants.

La première contribution de la thèse concerne la proposition d'une approche qui combine le B événementiel et le langage de modélisation TLA$^+$ pour la vérification des propriétés de vivacité. Nous définissons une extension syntaxique et sémantique du B événementiel permettant d'exprimer des propriétés de vivacité. Nous développons un prototype pour la transformation d'un modèle B en un module TLA$^+$ sur lequel nous effectuons la preuve des propriétés de vivacité avec le model checker TLC. Pour la vérification de ce type de propriétés sur des systèmes infinis, nous proposons l'utilisation des diagrammes de prédicats qui sont des abstractions des systèmes modélisés en TLA$^+$.

La deuxième contribution est la proposition d'une technique pour représenter explicitement le temps en B événementiel. Cette technique s'appuie sur la réalisation d'un entrelacement entre un processus qui gère le temps avec les autres processus du système. Le temps modélisé est discret et son écoulement est modélisé par des événements. Cette approche est assez différente des systèmes temporisés où l'on considère que le temps s'écoule indépendamment du système.

Dans la troisième contribution, nous proposons une approche de développement des systèmes automatisés en utilisant la technique de composition où il s'agit de développer conjointement le contrôleur et le composant physique qu'il contrôle et appliquer le raffinement aussi bien sur le contrôleur que le contrôlé. Le raffinement est une technique de base des méthodes que nous proposons et si notre objectif est de construire des contrôleurs corrects, le critère de correction porte sur le comportement du système automatisé qui résulte de la composition du contrôleur et du contrôlé. Nous présentons également un théorème de compositionnalité qui indique sous quelles conditions il est possible de déduire que le composé des raffinements des contrôleur et contrôlé est un raffinement du composé des contrôleur et contrôlé abstraits.

La dernière contribution porte sur la définition, la preuve et l'utilisation d'un patron de raffinement pour les processus continus dans des systèmes de production manufacturière. Ce type de patron prouvé permet d'utiliser l'abstraction discrete de l'effet d'un processus continu agissant pendant un certain temps.

Mots-clés: Modélisation, Vérification, Raffinement, Composition, Patrons de conception, Systèmes automatisés, B événementiel, TLA$^+$, Diagrammes de prédicats.

Abstract

This thesis deals with the development of automated systems while following the formal methods B and TLA$^+$. We propose a formal methodology based on the refinement paradigm to specify and verify the system that we model by two components :the controlled system representing the physical device and its environment, and the controller that controls the system. A synchronised product on the actions of these two components is applied to specify the automated system.

As a first contribution, we propose an approach combining the event B method and the language TLA$^+$ in order to verify liveness properties defined in user requirements. Inspired by the temporal logic of actions TLA, we first extend the event B notation to specify liveness properties and we give semantics of this extended syntax over traces. Second, we give transformation rules from a temporal B model into a TLA$^+$ module. We present, in particular, our prototype system called B2TLA$^+$, that we have developed to support this transformation. To consider infinite systems, we use predicate diagrams as abstractions of systems modelled with TLA$^+$.

To consider the real-time concept in automated systems, we propose as a second contribution a technique explicitly representing time in B event systems. This technique is based on an interleaving between any event handling time and the other system events.

By considering the well known co-design technique, we propose as a third contribution a refinement-based composition technique keeping a separation between controller and controlled systems in order to build correct automated systems satisfying user requirements. We prove a compositionality theorem with respect to refinement to get an efficient approach to verify the refinement of a synchronized composition between components. We verify the refinement of a synchronized composition by verifying separately the refinement of each component.

Finally, we define, prove and use in a case study as a fourth contribution the concept of a refinement pattern for continuous processes in manufacturing systems. Such proven pattern allows us to use the discrete abstraction of the effect of continuous processes operating for a while.

Keywords: Modelling, Verification, Refinement, Composition, Design patterns, Automated systems, Event B method, TLA$^+$, Predicate diagrams.

Table des matières

Introduction

1 Contexte du travail 15
2 Les méthodes formelles de développement de systèmes informatiques 16
2.1 Spécification formelle 16
2.2 Vérification formelle 16
3 Problématique 18
4 Contributions 18
5 Plan du document 19
6 Publications 20

Chapitre 1
Les systèmes automatisés

1.1 Introduction 23
1.2 Les systèmes hybrides 24
1.2.1 Définition 24
1.2.2 Exemple de systèmes hybrides 24
1.2.3 Modélisation des systèmes hybrides 24
1.3 Les systèmes automatisés ou de contrôle-commande 26
1.3.1 Définition 26
1.3.2 Problématique du contrôle 28
1.3.3 Méthode de développement des systèmes automatisés 29
1.4 Méthodes formelles pour le développement de systèmes automatisés 29
1.4.1 Méthodes formelles et cycle de vie 36
1.5 Besoins d'expression 37
1.6 Conclusion 38

Chapitre 2
La Méthode B et ses extensions

2.1 Introduction 39

2.2 Méthode B 39
2.2.1 Fondements de la méthode B 39
2.2.2 Raffinement 45
2.3 Méthode B-Événementiel 46
2.3.1 Différences avec B 46
2.3.2 Définition d'un modèle B événementiel 46
2.3.3 Raffinement 48
2.3.4 Exemple 49
2.3.5 Avantages de la méthode B-Événementiel 50
2.3.6 Limites de la méthode B-Événementiel 50
2.3.7 Outil support de B : l'Atelier B 50
2.4 Les extensions apportées à la méthode B 51
2.4.1 Extensions de la méthode B - Traitement du temps 51
2.4.2 Extensions du B événementiel - Traitement des propriétés de vivacité . . . 53
2.5 Conclusion 58

Chapitre 3
Le langage de modélisation TLA$^+$ et les diagrammes de prédicats

3.1 Introduction 61
3.2 La logique TLA 61
3.2.1 Exemple introductif 61
3.2.2 Présentation de TLA 62
3.3 Le langage de modélisation TLA$^+$ 66
3.3.1 Exemple 66
3.3.2 Syntaxe 69
3.3.3 Composition dans TLA$^+$ 70
3.3.4 Outil support de TLA$^+$: Model checker TLC 72
3.4 Les diagrammes de prédicats 73
3.4.1 Définition 73
3.4.2 Vérification de systèmes en utilisant les diagrammes de prédicats 74
3.4.3 Raffinement des diagrammes de prédicats 76
3.4.4 Outil support des diagrammes de prédicats DIXIT 79
3.5 Conclusion 80

Chapitre 4
Raffinement et composition dans le développement de systèmes complexes

4.1 Introduction 83

4.2 Raffinement et vérification 83
4.2.1 Simulation de la relation de raffinement 84
4.2.2 Exemples de raffinement 84
4.3 Composition 88
4.3.1 Compositions classiques des systèmes de transitions 88
4.3.2 Mécanisme de composition en B classique 93
4.3.3 Travaux existants autour de la composition en B événementiel 94
4.3.4 Conclusion 98

Chapitre 5
B événementiel et les propriétés de vivacité

5.1 Introduction 99
5.2 Comparaison du B événementiel et du langage TLA$^+$ 100
5.3 Approche proposée 101
5.3.1 Présentation de l'approche 102
5.3.2 Syntaxe et sémantique de l'extension 103
5.3.3 Règles de preuves des propriétés de vivacité 110
5.4 Transformation d'un modèle B temporel en un module TLA$^+$ 112
5.4.1 Les règles de traduction d'un modèle B temporel en un module TLA$^+$ . . 112
5.4.2 Le Prototype B2TLA$^+$ 117
5.5 Validation d'un module TLA$^+$ à l'aide du prouveur de théorèmes Isabelle/TLA . 118
5.5.1 Description informelle de l'exemple 119
5.5.2 Illustration de la méthode proposée 119
5.6 Validation d'un module TLA$^+$ à l'aide du model checker TLC 124
5.6.1 Description informelle de l'exemple 124
5.6.2 Illustration de la méthode proposée 124
5.6.3 Statistiques sur les preuves 139
5.6.4 Raffinement des propriétés de vivacité 139
5.6.5 Utilisation des diagrammes de prédicats 140
5.7 Conclusion 144

Chapitre 6
Représentation du temps en B événementiel

6.1 Introduction 147
6.2 Le temps dans les processus de développement 147
6.2.1 Etude de cas du timeout avec Esterel et B 149
6.3 Spécification de systèmes automatisés 152

6.4 Etude de cas : Brûleur à gaz 153
6.4.1 Description informelle du problème 153
6.4.2 Description des exigences du système automatisé 153
6.4.3 Modélisation du système automatisé 154
6.5 Conclusion 160

Chapitre 7
La composition en B pour le développement de systèmes automatisés

7.1 Introduction 161
7.2 La composition dans le développement de contrôleurs 162
7.2.1 Présentation de la démarche 162
7.2.2 Communication entre contrôleur et contrôlé 163
7.2.3 Modélisation du contrôleur et du contrôlé 163
7.2.4 Construction du système automatisé 165
7.3 Raffinement et composition 170
7.4 Etude de cas : tri des paquets postaux 174
7.4.1 Modèle abstrait du système 174
7.4.2 Premier raffinement du système de tri 178
7.4.3 Deuxième raffinement du système de tri 181
7.5 Conclusion 184

Chapitre 8
Modélisation des processus continus et patrons de conception

8.1 Introduction 187
8.2 Patrons de conception prouvés 188
8.2.1 Patron de raffinement 189
8.2.2 Exemple d'un patron de raffinement 190
8.3 Modélisation d'un système automatisé de production manufacturière 191
8.3.1 Description du système 191
8.3.2 Les contraintes du problème 192
8.3.3 Modélisation du système 193
8.3.4 Statistiques sur les preuves 199
8.3.5 Vérification des propriétés de vivacité 199
8.4 Décomposition et composition en B pour la modélisation des systèmes automatisés complexes 201
8.4.1 Décomposition du système automatisé 203
8.4.2 Application de la technique de décomposition au cas du mixeur 204

8.4.3 Premier raffinement . . . 207
8.4.4 Deuxième raffinement . . . 210
8.4.5 Troisième raffinement . . . 211
8.4.6 Quatrième raffinement . . . 213
8.4.7 Statistiques sur les preuves des modèles du contrôleur et contrôlé . . . 213
8.5 Conclusion . . . 214

Conclusion et Perspectives

1 Synthèse et bilan . . . 215
2 Perspectives . . . 216

Bibliographie **217**

Annexe A
Prototype B2TLA$^+$

1 Prototype . . . 225
2 Extension de la grammaire de B . . . 226
3 Symboles en B et TLA$^+$ . . . 226

Annexe B
Les modèles B du système de tri des paquets postaux

1 Modèle abstrait . . . 229
2 Modèle du premier raffinement . . . 230
3 Modèle du deuxième raffinement . . . 230
4 Modèle du troisième raffinement . . . 232

Annexe C
Les modules TLA$^+$ du système de tri des paquets postaux

1 Model checker TLC . . . 235
2 Module abstrait . . . 235
3 Module du premier raffinement . . . 236
4 Module du deuxième raffinement . . . 238
5 Module du troisième raffinement . . . 241

Annexe D
Les modèles B des contrôleur et contrôlé du système de tri des paquets postaux

1 Niveau abstrait . . . 245
1.1 Modèle du contrôlé . . . 245

1.2 Modèle du contrôleur 246
2 Premier raffinement 247
2.1 Modèle du contrôlé 247
2.2 Modèle du contrôleur 248
3 Deuxième Raffinement 250
3.1 Modèle du contrôlé 250
3.2 Modèle du contrôleur 251

Annexe E
Les modèles B du système de production manufacturière

1 Les modèles B 253
1.1 Modèle du premier raffinement 253
1.2 Modèle du deuxième raffinement 254
1.3 Modèle du troisième raffinement 256
1.4 Modèle du quatrième raffinement 261

Annexe F
Les modèles B des contrôleurs et contrôlés du système de production manufacturière

1 Niveau abstrait du système 263
1.1 Modèle du contrôlé 263
1.2 Modèle du contrôleur 264
2 Premier raffinement 265
2.1 Modèle du contrôlé 265
2.2 Modèle du contrôleur 267
3 Deuxième raffinement 269
3.1 Modèle du contrôlé 269
3.2 Modèle du contrôleur 271
4 Troisième raffinement 273
4.1 Modèle du contrôlé 273
4.2 Modèle du contrôleur 277
5 Quatrième raffinement 281
5.1 Modèle du contrôlé 281

Annexe G
Les modules TLA$^+$ du système de production manufacturière

1 Module abstrait 283

2 Module du premier raffinement 284

3 Module du deuxième raffinement 286

4 Module du troisième raffinement 288

5 Module du quatrième raffinement 295

Introduction

1 Contexte du travail

Le travail de la thèse porte sur l'application de la méthode B pour le développement des systèmes automatisés. Dans ces systèmes on distingue deux composants : le système à contrôler, appelée par la suite contrôlé (ou partie opérative), dont le comportement de nature hybride (discrète et continue), formé par le dispositif physique et son environnement dépend de commandes émises par l'autre partie, le contrôleur dont le comportement est de nature discrète. Ces systèmes exigent un niveau de sûreté et de fiabilité élevé.

Le dispositif physique est fourni. L'environnement, en l'absence d'influence externe, a son propre comportement temporel et évolue sous l'effet de processus naturels (gravitation naturelle, échanges thermiques,...). L'utilisateur est intéressé par le comportement du système automatisé formé par la composition des parties contrôle et opérative et non pas par la partie contrôle, elle même. Son souhait est que le système automatisé agisse sur l'environnement de façon que ce dernier se comporte en respectant les propriétés exigées.

Nous nous proposons de concevoir une méthode formelle pour construire le système de contrôle ainsi que le système à contrôler. Nous avons ainsi à modéliser l'environnement et son évolution, le contrôleur, et prouver que l'action du contrôleur produit sur l'environnement l'effet souhaité par le client.

Deux types d'approches ont été adoptées dans la thèse pour construire un contrôleur pour un système automatisé : la première consiste à commencer par la modélisation concrète du contrôlé, et ensuite construire le contrôleur d'une manière incrémentale, l'étape finale consiste à séparer le contrôleur du contrôlé. Dans la deuxième démarche, nous avons été amenés à nous intéresser à l'interaction entre un contrôleur et un contrôlé. Elle consiste dans une première étape à construire un modèle abstrait du contrôlé ainsi qu'un modèle abstrait du contrôleur. Par la suite, nous procédons à des raffinements successifs des deux parties jusqu'à obtenir un modèle implémentable du contrôleur et un modèle du contrôlé suffisamment concret pour tenir compte de tous les aspects du composant physique. A chaque étape nous composons les deux modèles pour obtenir un modèle du système automatisé sur lequel nous effectuons la preuve des propriétés de comportement souhaitées par l'utilisateur.

La modélisation du comportement des systèmes qui nous intéressent nécessite des constructions absentes de la méthode B. Plutôt que d'utiliser une méthode plus adaptée aux applications temps réel, nous avons choisi de conserver B qui est applicable à un certain niveau d'abstraction pour un certain nombre de raisons. Aux niveaux où il faut intégrer des propriétés temporelles nous avons recours à d'autres outils comme ceux qui sont associés à TLA^+.

Dans les systèmes automatisés, nous sommes confrontés à la modélisation de comportements continus qui nécessitent l'usage d'équations aux dérivées partielles. A un niveau abstrait nous n'avons pas besoin de prendre en compte le caractère continu des processus que nous modélisons,

seuls certains états distingués par lesquels les processus passent nous intéressent. Au niveau concret, nous modélisons de façon discrète la progression des processus continus pour permettre d'exprimer qu'ils peuvent atteindre ces états distingués. La relation entre le niveau abstrait et le niveau concret a donné lieu à la définition d'un patron de conception qui s'applique de façon récurrente dans les études de cas que nous avons traitées.

Le raffinement est une technique de développement qui est au cœur de la méthode B, c'est un outil pour maîtriser la complexité des systèmes, nous l'avons associée à la composition. La composition a été appliquée pour former le système automatisé à partir des composants que sont le contrôleur et le contrôlé mais aussi dans le cas où un système automatisé est clairement composé de sous-systèmes.

2 Les méthodes formelles de développement de systèmes informatiques

Les méthodes formelles reposent sur des fondements mathématiques qui permettent de raisonner sur des propriétés et de construire des preuves. Ces preuves montrent que les propriétés exprimées sur un programme sont bien respectées.

Nous présentons deux techniques de modélisation des systèmes complexes : le raffinement et la composition ainsi que deux techniques de preuve : le model checking et la preuve.

2.1 Spécification formelle

La spécification formelle consiste à établir une description formelle d'un système et de ses propriétés de comportement. Elle utilise un langage formel dont la syntaxe et la sémantique sont définies mathématiquement. L'intérêt des méthodes formelles est la possibilité de vérifier de manière rigoureuse des propriétés sur un modèle formel. Pour assurer qu'un système respecte certaines propriétés, on lui associe un model formel qu'il est possible de vérifier par preuve.

Raffinement : Le raffinement [Abrial and Hallerstede, 2006] est une approche de plus en plus répandue pour construire progressivement des systèmes complexes : il consiste à dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du système préserve bien sa correction vis-à-vis de la spécification précédente.

Composition : La composition [Bellegarde *et al.*, 2002] est une technique de développement horizontal qui permet de maîtriser la complexité des systèmes de grande taille. Elle permet une spécification structurée avec des composants réutilisables. De façon générale, la composition dans le développement consiste à assembler des composants vérifiés en introduisant éventuellement des contraintes pour restreindre le comportement des composants.

2.2 Vérification formelle

L'utilisation de méthodes formelles permet une vérification rigoureuse de systèmes informatiques. Il existe deux techniques de vérification formelle : le *model checking* [Merz, 2001; 2002] et le *theorem proving* [Rushby, 2000].

Vérification par model checking : Le *model checking* [Merz, 2002] est une approche algorithmique qui consiste à effectuer la vérification du modèle d'un système en s'assurant que tout comportement satisfait les propriétés attendues. Cette vérification est entièrement automatisée et consiste à explorer tous les cas possibles. Le résultat de cette analyse est soit la confirmation que chaque propriété est vérifiée par le modèle, soit qu'elle ne l'est pas. Dans le dernier cas, et c'est un des principaux intérêts de cet outil, le model checker renvoie un contre-exemple. La vérification, aussi appelée analyse d'atteignabilité, consiste à générer tous les états du système qui sont atteignables à partir de l'état initial et à vérifier en même temps que la propriété énoncée est toujours satisfaite sur le graphe ainsi obtenu. Il s'agit alors d'effectuer une recherche exhaustive dans l'espace des états. Un résultat n'est possible que si le modèle est fini, et que, de plus, la taille du problème est abordable pour un ordinateur. Sans quoi l'exploration peut prendre un temps très long, voire ne jamais s'exécuter sur la machine par manque de ressources. L'espace des états se définit simplement à l'aide de toutes les variables qui participent au contrôle du comportement du modèle. Si cet espace est fini, il est alors possible d'énumérer tous les états et vérifier si des formules logiques sont valides pour tous ces états. Il y a trois étapes dans la mise en oeuvre d'un model checker :

- modéliser le système par un automate d'états finis,
- définir les propriétés à vérifier sur le modèle dans une logique temporelle,
- vérifier les propriétés définies sur le modèle.

Le model checking présente l'avantage d'être une technique complètement automatisée et rapide et ne requiert aucune aide extérieure pendant l'analyse. Un autre avantage du model-checking est qu'il est capable de fournir des contres-exemples. Cependant, l'inconvénient du model checking reste le problème d'explosion combinatoire des états à couvrir. Il est, en effet, impossible de faire une énumération exhaustive d'un nombre d'états infinis. Et surtout, la représentation de toutes les situations possibles conduit rapidement à un dépassement des capacités de l'ordinateur à stocker toutes ces informations en mémoire. En effet, l'effort de calcul requis pour la vérification d'une propriété sur un ensemble d'états augmente rapidement avec le nombre des variables décrivant ces états. Les méthodes d'états finis sont moins applicables sur des spécifications portant sur des espaces d'états conséquents ou non bornés. Pour ces cas là, les méthodes déductives semblent plus appropriées.

Nous pouvons citer comme exemple de model checker les outils, comme TLC [Lamport and Yu, 2003], UPPAAL [Larsen *et al.*, 1997a; 1997b], SPIN [Holzmann, 1997], SMV [Clarke *et al.*, 1994], Kronos [Bozga *et al.*, 1998; Kro, 1998] et HyTech [Henzinger *et al.*, 1997; Hyt, 1997]. Nous retenons dans la thèse le model checker TLC.

Vérification par preuve : La vérification par preuve s'appuie sur un système formel contenant des axiomes et des règles d'inférence. Elle consiste à dériver un théorème à partir d'axiomes et/ou de théorèmes prouvés par application des règles d'inférence.

Il existe de nombreux prouveurs de théorèmes, tels que Coq [Coq, 2002], Isabelle/HOL [Nipkow, 2000], PVS [Owre *et al.*, 1992; PVS, 1992] et l'Atelier B [ClearSy, 2002a].

Les techniques de preuve ont l'avantage de pouvoir traiter des systèmes à nombre d'états infini contrairement à la technique de *model checking* qui repose sur des systèmes finis et décidables. Néanmoins, l'une des limitations de l'utilisation de la preuve en industrie est le fait que les prouveurs de théorèmes ont besoin d'être assistés par des ingénieurs expérimentés. En effet, quel que soit leur niveau d'automatisation, tous les outils de preuve nécessitent l'intervention d'un spécialiste, car leur usage requiert une très grande connaissance aussi bien au niveau du modèle à vérifier que des techniques de preuves utilisées.

3 Problématique

Le travail de thèse a pour objectif :

- L'utilisation et l'adaptation du B événementiel pour la modélisation de systèmes automatisés, en particulier la prise en compte de propriétés temporelles.
- La vérification des propriétés de vivacité sur de tels systèmes.
- La modélisation des systèmes automatisés en utilisant les techniques de composition et de raffinement.
- La modélisation des comportements continus correspondants à l'activité de processus physiques.

4 Contributions

Nos contributions se situent d'une part au niveau spécification et au niveau vérification des systèmes automatisés d'autre part.

- Dans la première contribution, nous proposons une approche qui combine le B événementiel et le langage de modélisation TLA$^+$ pour la vérification des propriétés d'invariance et de vivacité. TLA$^+$ a prouvé son efficacité dans l'expression et la vérification de propriétés de vivacité. A l'instar de B, elle supporte le raffinement et se base sur la notion d'actions qui s'apparente à celle d'événements ce qui la rend compatible avec B. Nous utilisons, ainsi, l'environnement de spécification de B ainsi que l'outil de preuve associé pour vérifier des propriétés d'invariance et nous utilisons l'environnement de TLA$^+$ pour vérifier les propriétés de vivacité qui ne peuvent être facilement exprimées et vérifiées en B.

  Dans cette contribution, nous définissons une extension du B événementiel permettant d'exprimer des propriétés de vivacité. En particulier, nous donnons la sémantique du B événementiel engendrée par l'ajout des nouvelles constructions. De plus, nous proposons des règles de preuve pour la vérification de telles propriétés dans l'axiomatique de B. Nous développons un prototype pour la transformation d'un modèle B en un module TLA$^+$. Après avoir transformé le modèle B temporel en un module TLA$^+$ et pour la vérification des propriétés de vivacité, nous utilisons le prouveur de théorèmes Isabelle/TLA ou le model checker TLC.

  Nous avons choisi la technique de *model checking* au lieu de la vérification par preuve, car cette dernière démarche de vérification n'est pas automatique, et elle demande à l'utilisateur d'être expert en spécification et dans l'utilisation de cette technique. En particulier celui-ci doit fournir des variants, et doit guider le prouveur en lui définissant des stratégies de preuves. Ce choix limite notre travail au cas des systèmes d'événements à nombre d'états fini. Pour traiter des systèmes infinis, nous proposons l'utilisation des diagrammes de prédicats qui sont des abstractions des systèmes modélisés en TLA$^+$ et l'outil associé (DIXIT).
- Dans la deuxième contribution, nous proposons une technique pour représenter le temps en B événementiel. Cette technique s'appuie sur la réalisation d'un entrelacement entre un processus qui gère le temps avec les autres processus du système. Elle est inspirée de la notion du temps multiforme en Esterel.
- Dans la troisième contribution, nous proposons une approche de développement des systèmes automatisés en utilisant la technique de composition où il s'agit de développer

conjointement le contrôleur et le composant physique qu'il contrôle. Elle consiste à partir d'un modèle abstrait du contrôleur et d'un modèle abstrait du contrôlé puis à effectuer une suite de raffinements successifs jusqu'à obtenir un modèle implémentable du contrôleur et un modèle du contrôlé suffisamment concret pour tenir compte de tous les aspects du composant physique. Cette approche présente plusieurs intérêts tels que la possibilité de réutilisation de la partie opérative et l'obtention plus facile du contrôleur en fin de développement, le raffinement séparé des composants, la vérification de l'interaction du système à contrôler avec le système de contrôle avant sa mise en oeuvre et la réduction de la complexité du système automatisé.

- La dernière contribution porte sur la définition, la preuve et l'utilisation d'un patron de conception que nous avons créé pour abstraire le comportement de processus continus. Le comportement continu est discrétisé pour qu'il puisse être possible d'utiliser l'environnement de B pour prouver ce patron d'application très fréquent dans les systèmes automatiques de production manufacturière.

5 Plan du document

Ce document est organisé de la manière suivante :

Partie I : Contexte scientifique, préliminaires

- Le chapitre 1 est consacré à la présentation des systèmes automatisés et l'utilisation des méthodes formelles pour le développement de tels systèmes. Nous étudions en particulier :
  - Les systèmes hybrides et leurs approches de modélisation,
  - Les systèmes automatisés et leurs approches de modélisation,
  - Les méthodes formelles pour le développement des systèmes automatisés
- Le chapitre 2 nous permet de présenter la méthode B et les extensions apportées autour de cette méthode. Nous étudions en particulier :
  - La méthode B
  - Le B événementiel
  - Les travaux existants autour des extensions de la méthode B
- Le chapitre 3 présente le langage de modélisation TLA$^+$ et les diagrammes de prédicats.
- Le chapitre 4 est consacré à la présentation de deux techniques de développement : le raffinement et la composition. Nous présentons les différentes notions classiques de raffinement, la technique de composition ainsi que les travaux existants autour de cette technique en B.

Partie II : Contributions

- Dans le chapitre 5, nous présentons une approche de spécification et de vérification des propriétés d'invariance et de vivacité des systèmes automatisés en utilisant le B événementiel et le langage TLA$^+$. Nous étendons, en particulier l'expressivité et la sémantique d'un modèle B par de nouvelles clauses pour obtenir un modèle B temporel. Après avoir transformé ce dernier en un module TLA$^+$, nous vérifions les propriétés de vivacité sur des systèmes à états finis par le model checker TLC. Pour la vérification de ces types de propriétés sur des systèmes à états infinis, nous proposons l'utilisation des diagrammes de prédicats. Nous illustrons notre approche sur le système de tri de paquets postaux.

- L'objectif du chapitre 6 est la présentation d'une technique de représentation du temps en B événementiel pour la modélisation de systèmes automatisés.
- Le chapitre 7 est consacré à l'étude de la composition en B pour le développement de contrôleurs des systèmes automatisés. En effet, nous nous intéressons à l'interaction entre un contrôleur et un contrôlé. Pour ce faire, nous proposons une approche qui rappelle la technique du *co-design* où il s'agit de développer conjointement le contrôleur et le composant physique qu'il contrôle. Nous développons deux modèles l'un pour le contrôleur et l'autre pour le système qu'il contrôle. Le modèle du système automatisé est obtenu par composition de ces deux modèles. Par la suite, nous enrichissons les deux modèles qu'on compose à chaque niveau de raffinement pour obtenir le modèle du système automatisé sur lequel nous faisons la preuve des propriétés de comportment. A la fin du cycle de développement nous disposons d'un modèle concret du contrôleur.

  Dans le cadre où nous nous plaçons, les deux modèles à composer sont dissymétriques où les deux composants ne sont pas de même nature ; un composant commande et l'autre réagit. La définition de la composition tient compte de cette dissymétrie
- Dans le chapitre 8, nous proposons l'utilisation des patrons de conception pour la modélisation des processus continus. Ces patrons permettent de réutiliser les specifications existantes dans de nouveaux projets. Nous illustrons notre méthode sur un système de production manufacturière. Nous appliquons aussi, la technique de composition pour la construction de contrôleurs pour des systèmes automatisés complexes. Ces systèmes sont décomposés en un ensemble de sous systèmes communicants. Pour chaque sous système, nous donnons ces parties contrôle et opérative. La communication entre les composants est réalisée à travers leurs contrôleurs.

6 Publications

Ce travail a donné lieu aux publications suivantes :

- dans [Mosbahi and Jaray, 2004a], nous donnons une représentation du temps en B événementiel pour la modélisation des systèmes de contrôle commande.
- dans [Mosbahi and Jaray, 2004b], nous présentons une démarche formelle de développement de systèmes de contrôle-commande que nous appliquons sur l'exemple du brûleur à gaz.
- dans [Mosbahi *et al.*, 2006b], nous proposons une méthode formelle de développement de systèmes automatisés en utilisant le langage de modélisation TLA^+.
- dans [Mosbahi and Jemni, 2006], nous proposons une méthode utilisant conjointement le B événementiel et le langage de modélisation TLA^+ pour la spécification et la vérification des systèmes réactifs.
- dans [Mosbahi *et al.*, 2006a], nous présentons une méthode formelle de développement de systèmes de contrôle-commande en utilisant le B événementiel que nous appliquons sur l'étude de cas de tri de paquets postaux.
- dans [Mosbahi and Jaray, 2007], nous proposons une méthode de spécification et de vérification des propriétés de vivacité dans des systèmes d'événements B. Nous présentons des règles de transformation d'un modèle B vers un module TLA^+ et un prototype supportant cette transformation.
- dans [Mosbahi *et al.*, 2007b], nous proposons une approche formelle pour le développement de systèmes automatisés.

- dans [Mosbahi *et al.*, 2007a], nous présentons une méthode de spécification et de vérification des propriétés de vivacité en B événementiel. Nous proposons une extension syntaxique et sémantique ainsi que des règles de preuve correspondants à l'ajout des propriétés de vivacité.

Chapitre 1

Les systèmes automatisés

1.1 Introduction

Nous nous intéressons dans ce travail de thèse, à l'application de méthodes formelles pour le développement de systèmes automatisés (systèmes de contrôle commande) qui forment une sous-classe de systèmes hybrides [Antsaklis and Koutsoukos, 2003]. Dans ces systèmes nous distinguons deux composants : l'un, le système à contrôler (partie opérative) qui a un comportement essentiellement continu dépendant de commandes émises par l'autre partie (logicielle) de nature discrète (partie contrôle). Nous nous intéressons plus particulièrement lors du développement de ces systèmes automatisés, à la synthèse de contrôleurs qui permet entre autre de raffiner une spécification incomplète de manière à atteindre un certain objectif, comme la satisfaction d'une propriété non encore vérifiée sur le système initial. Le raffinement consiste à contraindre une spécification afin d'atteindre l'objectif fixé.

En théorie, plusieurs techniques ont été proposées pour la conception des contrôleurs. Nous pouvons utiliser les réseaux de Petri [Murata *et al.*, 2006], les langages formels ou l'approche de Ramadge et Wonham [Ramadge and Wonham, 1987; 1989; Wonham, 2003], les logiques temporelles [Ostroff, 1989]. Dans notre travail de thèse, nous nous sommes inspirés des travaux de Wonham et Ramadge pour développer le contrôleur le plus permissif et controllable en utilisant une méthode formelle.

Les méthodes formelles permettent de prouver, à priori, qu'un dispositif physique commandé satisfait un cahier des charges imposé. En effet, l'utilisation de ces méthodes apparaît une des solutions principales pour le développement des systèmes de grande qualité et sûrs de fonctionnement à des coûts et délais raisonnables. Cette nécessité est accentuée par le fait que plusieurs domaines d'applications tels que l'aéronautique, le nucléaire, les télécommunications et les systèmes de production exigent un niveau de sûreté et de fiabilité élevée. L'utilisation de ces méthodes dans le processus de développement permet la vérification et la validation de la spécification et facilite le passage à l'implémentation.

Dans la section 2 de ce chapitre nous présentons les systèmes hybrides. La section 3 présente une sous-classe de systèmes hybrides, les systèmes automatisés. Dans la quatrième section, nous présentons les méthodes formelles et leurs avantages dans le développement de ces systèmes. La section 5 présente les méthodes formelles dans le cycle de vie de ces systèmes. Enfin, la section 6 présente les besoins d'expression de propriétés des systèmes automatisés.

1.2 Les systèmes hybrides

1.2.1 Définition

Les systèmes hybrides sont des systèmes ayant des composantes discrètes et continues. L'état des systèmes à événements discrets prend ses valeurs dans un ensemble fini et évolue de manière discontinue. Les systèmes continus se caractérisent par une évolution continue de leur état dans le temps et sont modélisés par des équations différentielles [Antsaklis and Koutsoukos, 2003].

Il existe deux grandes classes de systèmes hybrides : la première est hybride par nature, c'est-à-dire un système à temps continu comportant des aspects à événements discrets. La seconde classe est hybride de par le contrôleur (ou superviseur) à événements discrets qui commande un processus à temps continu. Cette seconde classe est parfois appelée Hybrid Control System, HCS [Michael Lemmon and Antsaklis, 2006]. Ces deux grandes classes de systèmes hybrides sont fréquemment rencontrées en industrie. Un contrôleur de processus physique est un exemple typique de systèmes hybrides. Le contrôleur est un système à états discrets et contrôle l'évolution d'un système à variables continues. Un exemple d'un tel système est celui d'un contrôleur de température d'une cuve industrielle [Lewis, 2001] ; passée une certaine température, le système doit changer d'état discret pour enclencher un mécanisme de refroidissement. Aussi un système de commande temps-réel avec son environnement continu peut être considéré comme un système hybride ; il doit produire des réponses temps-réel à des stimuli qui changent de façon continue.

1.2.2 Exemple de systèmes hybrides

La figure 1.1 montre un exemple d'un système hybride. La partie continue est formée par les bacs T1, T2, T3 et T4 et la partie discrète est formée par les actionneurs et les capteurs. Les bacs T1, T2 et T4 sont équipés de capteurs de niveau binaires. Ils indiquent si le niveau atteint certaines limites prédéfinies. Le bac T2 a aussi un radiateur et un capteur de température. Les deux contenus de T2 et T4 sont mixés dans T3. Le produit est sorti du bac T3 via une vanne proportionnelle. Ce dernier bac a un capteur continu de niveau. Les phénomènes dynamiques à modéliser dans le procédé sont fondamentalement continus et correspondent aux flux de matière et d'énergie. En même temps, le procédé a des actionneurs et des capteurs dont le comportement est fondamentalement discontinu. Ainsi, le comportement global du système est composé de deux parties : la partie continue et la partie événementielle.

1.2.3 Modélisation des systèmes hybrides

La modélisation d'un même procédé peut être réalisée de différentes manières en fonction du niveau d'abstraction souhaité du modèle. Par exemple, le niveau d'un produit dans une cuve peut être modélisé de manière continue. Il faudra alors connaître son évolution qui sera estimée par une équation différentielle traduisant un bilan d'une quantité conservative (comme la masse par exemple). Alors, le niveau pourra être connu à chaque instant. Ce même niveau peut également être modélisé de manière discrète en ne détectant que des franchissements de seuil par exemple la hauteur maximale et minimale dans la cuve. Cette modélisation beaucoup plus abstraite permet de définir une exigence qui peut assurer que la cuve ne déborde pas et ne soit jamais vide, mais le niveau ne sera pas connu à chaque instant.

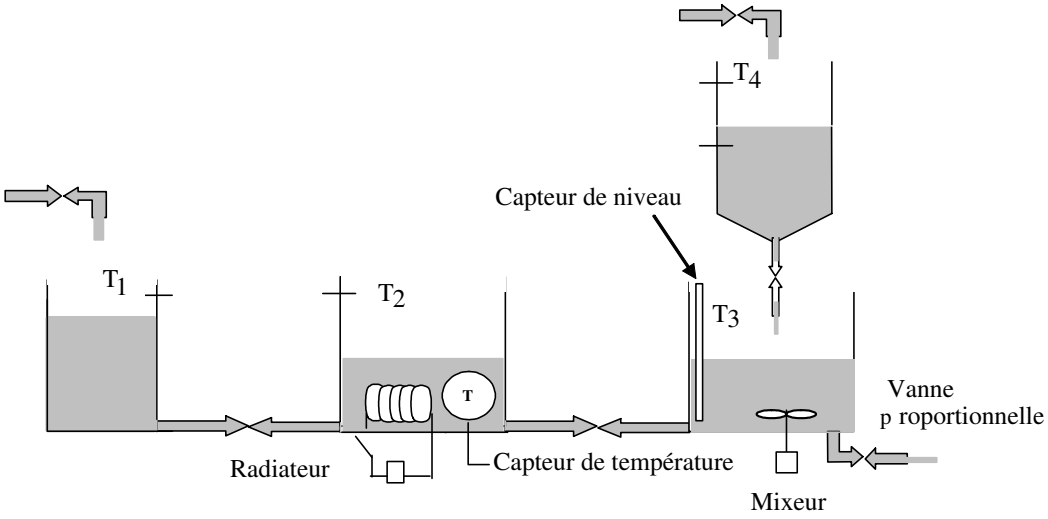


FIG. 1.1 – Exemple de système hybride.

Trois principales classes d'approches pour la modélisation des systèmes hybrides sont définies de la façon suivante [Chombart, 1997] :

- ***Approche continue***. Il s'agit d'étudier le comportement des systèmes vus comme des systèmes continus comportant des discontinuités, et éventuellement, de définir un modèle " étendu ". L'approche consiste à définir une approximation des dynamiques discrètes du système hybride en utilisant des équations différentielles (ou de différence) pour simuler l'occurrence des événements discrets. Les dynamiques continues sont modélisées par des équations différentielles et les dynamiques discrètes par des équations aux différences.

- ***Approche événementielle***. Contrairement à l'approche continue, il s'agit de définir une approximation des dynamiques continues de telle façon que le système hybride ne soit représenté que par les événements qui les caractérisent. Cette méthode fait appel à la théorie de Ramadge et Wonham pour laquelle il a mis en évidence la synthèse d'un superviseur et l'analyse en boucle fermée.

- ***Approche mixte***. Cette approche conduit à rechercher des modifications des modèles discrets, adaptées à la modélisation des phénomènes hybrides, en particulier en étudiant des variables continues. La modélisation combinée des parties discrètes et continues dans une même structure se veut être une méthode orientée vers la simulation et la vérification du fonctionnement. La sémantique d'un tel modèle est basée sur une extension des graphes de transitions d'états. Nous avons comme approches mixtes les automates hybrides, les réseaux de Pétri hybrides et l'extension de la logique du calcul de durée (Extended Duration Calculus).

Dans la section suivante, nous nous intéressons à une sous-classe de systèmes hybrides, les systèmes automatisés et leur développement en utilisant des méthodes formelles.

1.3 Les systèmes automatisés ou de contrôle-commande

Une caractérisation des différents composants d'un système automatisé est présentée dans cette section. Le but est d'examiner les différents besoins d'expression et de dégager les caractéristiques d'un formalisme adéquat pour la formalisation de comportements de tels systèmes.

1.3.1 Définition

Le terme contrôle-commande recouvre l'ensemble des moyens permettant d'automatiser le contrôle d'un procédé, de le réguler, d'engendrer des actions de protection et de mise dans l'état sûr de l'installation [Nise, 2003]. Ces systèmes constituent une classe particulière de systèmes hybrides. Leur organisation est décrite ainsi : le concepteur d'un système de contrôle-commande ne doit pas considérer les composants du système séparément, il doit considérer le système en sa totalité. Le dispositif doit être contrôlé afin de fournir le produit attendu en satisfaisant certains critères. De manière générale, ces systèmes ont besoin de l'intervention d'un opérateur extérieur pour assurer la communication d'information de régulation ou de contrôle.

Les systèmes de contrôle-commande sont caractérisés par une relation générale entre les composants principaux. La figure 1.3 présente une boucle de contrôle dont les composants sont :

- L'*opérateur* étant une personne qui communique directement avec le contrôleur. Il peut envoyer des informations au contrôleur ou en recevoir. Ainsi, l'opérateur peut agir sur le dispositif physique en passant par le contrôleur. L'agent de conduite d'un train est un exemple d'opérateur.
- La *partie opérative* formée par le dispositif physique et son environnement, étant contrôlée par le contrôleur. Les *capteurs* et les *actionneurs* font partis aussi de la partie opérative. Ce sont des composants matériels qui servent au contrôleur pour connaître (via les capteurs), et pour manipuler (via les actionneurs) le dispositif physique.
- La *partie contrôle* qui est construite pour agir sur le comportement du dispositif physique afin qu'il respecte des contraintes données. Il interagit avec l'opérateur au moyen des boutons et des indicateurs, et avec le dispositif physique au moyen des capteurs et des actionneurs.

La relation générale entre ces composants peut être résumée ainsi : le contrôleur obtient des informations (variables mesurées) sur le dispositif physique qu'il doit contrôler. En fonction des variables mesurées et selon l'algorithme de contrôle, il doit manipuler les variables contrôlées, au moyen des actionneurs, afin de garder le fonctionnement du dispositif dans les limites prédéfinies.

La partie contrôle

La partie contrôle a un comportement permettant le pilotage de la partie opérative. Il suit l'évolution de ses états via les capteurs et réagit à toute évolution jugée significative pour garantir un contrôle permanent de la partie opérative.

Pour piloter la partie opérative, la partie contrôle doit contenir un ensemble d'actions destinées à la partie opérative. Ces actions sont des réactions à des événements instantanés telles que la détection d'une température assez élevée. Elles mettent en route des processus physiques ramenant l'environnement à un état souhaité. Par exemple, le système de contrôle peut activer la fermeture du robinet de gaz si la partie opérative n'a pas détecté l'existence d'une flamme

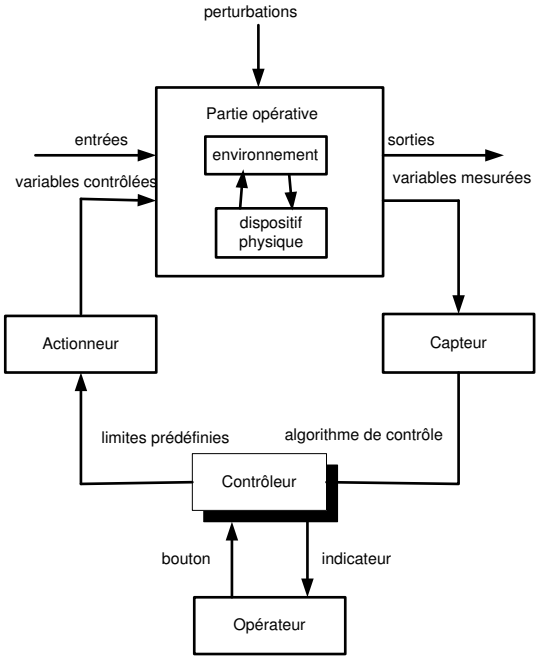


FIG. 1.2 – boucle de contrôle.

pendant un certain temps ; il peut ouvrir une vanne de sortie d'un produit si ce dernier est au niveau N.

Comme il peut agir sur la partie opérative, suite à ce qu'il observe dans l'environnement, le système de contrôle peut aussi recevoir des consignes de l'opérateur ou être piloté par un opérateur. C'est le cas du distributeur de monnaie où l'utilisateur produit un événement par l'introduction d'une carte, l'appui sur un bouton, la confirmation d'un code, ...Ces événements sont observés par le système de contrôle du distributeur qui réagit en envoyant des actions à la partie opérative ayant par exemple l'effet d'avaler la carte, d'afficher le menu, de fournir l'argent, ..etc.

La partie opérative

La partie opérative est composée du dispositif physique et de son environnement.

- **Le dispositif physique.** Via les capteurs, le dispositif physique offre au système de contrôle une image de l'état de l'environnement (mesures et événements), qu'il met sans cesse à jour, et dont la vitesse d'évolution ne dépend que de ses propres dynamiques internes. Si le contrôle consiste en le pilotage du dispositif physique, la réaction du système de contrôle consiste en l'émission de commandes vers les actionneurs ; si le contrôle consiste en le suivi du système cible, la réaction du système de contrôle consiste en l'enregistrement des états de l'environnement (données) observés via les actionneurs.
- **L'environnement.** en général, l'environnement a sa dynamique intrinsèque. Le rôle du système de contrôle est d'observer son évolution et de réagir, pour éviter tout comportement anormal de cette partie. Sans contrôle ce dernier peut évoluer vers des états jugés

intolérables ou non souhaités. L'environnement est soumis à des actions provenant du dispositif physique ou à des influences externes. Il est défini d'une façon abstraite par des variables qui évoluent au cours du temps sous l'action de phénomènes naturels : champs gravitationnels, magnétiques,... ou bien sous l'effet d'une action provenant du dispositif physique.

Par exemple, dans le cas du système de la climatisation, l'environnement est constitué de la pièce à climatiser avec son air ambiant. Ce système peut être représenté par une variable d'état qui représente la température de l'air. La température de la pièce subit l'effet de processus naturels. Elle est soumise à un refroidissement ou à un réchauffement climatique. Sans contrôle, la température peut atteindre une valeur qui sort de la plage idéale. Le système de contrôle doit observer l'évolution de la température et réagir, par exemple, pour activer le réchauffage quand il trouve une valeur qui descend au-dessous d'un seuil minimal.

Dans les chapitres suivants, nous notons par *contrôlé*, la partie opérative et par *contrôleur* la partie contrôle.

1.3.2 Problématique du contrôle

Le problème de la synthèse de contrôleur [Ramadge and Wonham, 1989; Wonham, 2003] consiste, à partir d'une spécification d'un système physique et d'une propriété attendue de celui-ci, à synthétiser un contrôleur qui, placé dans l'environnement du système physique et disposant de la spécification, va garantir que les exécutions possibles du système valident cette propriété (1.3).

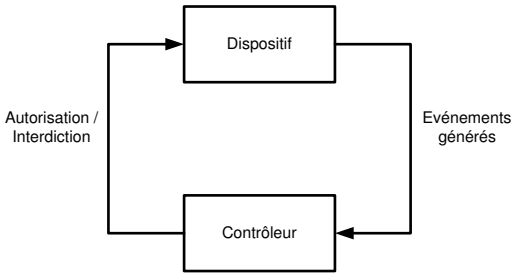


Fig. 1.3 – Schéma de contrôle [Ramadge et Wonham 1987].

La problématique de contrôle repose sur la théorie des langages formels et la théorie des automates. Dans cette approche, un procédé est considéré comme un générateur spontané d'événements et le rôle du contrôleur permet d'interdire l'occurrence de certains événements dans le procédé. En effet, le modèle physique est représenté par un langage dont l'alphabet représente les différentes actions possibles ; un mot de ce langage représente alors une séquence d'actions correspondant à une exécution du système. Dans cette approche, l'action du contrôleur a pour effet d'interdire, à différents moments, un certain nombre de lettres de l'alphabet, appelées événements contrôlables. Cette approche a notamment été mise en oeuvre sur des automates et des réseaux de Petri.

En nous intéressant au développement d'un contrôleur pour un système automatisé, nous

avons adopté une démarche semblable aux travaux de Ramadge et Wonham. Dans notre approche, nous utilisons la technique de raffinement pour construire le contrôleur.

1.3.3 Méthode de développement des systèmes automatisés

En termes de méthodes de développement, il s'agit de construire un système automatisé qui contraigne le comportement du système à contrôler. Nous recherchons ainsi à utiliser les méthodes formelles pour développer le contrôleur (logiciel de contrôle-commande) et vérifier que ce système, composé avec la partie opérative, agit sur l'environnement pour éviter tout comportement anormal.

Utiliser une méthode formelle pour le développement d'un logiciel consiste à spécifier de façon formelle le comportement attendu du logiciel par des propriétés et à prouver par la suite que le logiciel satisfait cette spécification. Généralement, nous remarquons que pour les systèmes automatisés, le processus de développement formel diffère sensiblement de celui qui est utilisé pour un système classique. Nous posons le problème de développement formel d'un système automatisé de la façon suivante : nous avons à développer un système de contrôle. Pour ce faire, il faut construire la spécification formelle de ce système et prouver que le système de contrôle qui vérifie cette spécification, composé avec la partie opérative, forme un système automatisé qui satisfait les besoins de l'utilisateur.

D'une façon générale, un système automatisé a un comportement qui consiste à exercer des commandes ou à interagir avec la partie opérative pour modifier son comportement et le maintenir dans un des états souhaités par l'utilisateur. Le dispositif physique est celui qui a la possibilité d'exercer une contrainte sur l'environnement, il est le plus souvent imposé. L'environnement a un comportement qui lui est propre en l'absence de contrôle, mais il est pilotable par la partie opérative qui modifie son état selon les consignes du système de contrôle.

D'un point de vue logique, nous avons la boucle de contrôle suivante : le système de contrôle observe la partie opérative et prend une décision d'action qu'il demande au composant physique d'effectuer pour qu'il agisse sur l'environnement. Via ces capteurs, le dispositif physique offre au système de contrôle des informations qui permettent d'actualiser l'image de l'environnement. Le rôle du système de contrôle consiste alors à suivre l'évolution de cette image et à réagir à toute évolution jugée significative garantissant que la partie opérative reste toujours contrôlable.

Le système automatisé résulte de l'intéraction entre la partie opérative et la partie contrôle, à développer. Il serait souhaitable d'obtenir ces propriétés comme une sorte de composition de ces deux parties. Étant donné la correspondance entre ces deux systèmes, nous devons avoir une cohérence au niveau de la composition, garantissant que la composition des deux spécifications des deux parties est une spécification du système composite. Cet aspect est garanti du fait que dans l'étape de construction de la spécification du système de contrôle, nous tenons compte de la description de la partie opérative et de ses fonctionnalités.

1.4 Méthodes formelles pour le développement de systèmes automatisés

L'accroissement de la complexité des systèmes automatisés rend de plus en plus nécessaire l'utilisation de méthodes formelles qui permettent de prouver, *a priori*, qu'un procédé commandé satisfait un cahier des charges imposé.

Cette section a pour but d'introduire les méthodes formelles. Ces méthodes sont des techniques qui reposent sur des fondements mathématiques précis qui permettent de raisonner sur des propriétés et de faire des preuves. Ces preuves montrent et démontrent que les propriétés exprimées sur un programme sont bien respectées. Une approche formelle s'appuie sur trois aspects :

- un langage doté d'une sémantique formelle bien définie pour décrire le modèle du système étudié de façon non ambiguë.
- un langage pour exprimer des propriétés sur le système étudié. Ces langages sont généralement basés sur la logique. Le type de propriété qui peut être exprimé dépend du langage utilisé.
- Une méthode, une théorie, pouvant définir (1) de quelle manière écrire des modèles, (2) quelles propriétés peuvent être écrites sur ces modèles et (3) de quelle manière elles peuvent être vérifiées. Il faut également des outils supportant cette méthode : gestion de la bibliothèque des modèles, génération des propriétés de bonne formation des modèles, outils permettant la vérification des propriétés (prouveur, model-checker,...).

Nous proposons une vue synthétique et non exhaustive du monde des méthodes formelles. En nous focalisant sur quelques technologies clés et représentatives, nous tentons de montrer le choix et la diversité des techniques de modélisation aujourd'hui disponibles, et la difficulté que représente ce choix pour un industriel. En effet, ce choix dépend en grande partie des objectifs visés par la modélisation. Outre le choix de la technique de modélisation, la difficulté de mise en oeuvre doit également être prise en compte. Cela comprend les outils de développement et la compétence requise pour les personnes travaillant sur ces technologies. Pour les industriels voulant intégrer la modélisation formelle dans leur cycle de développement, la présence d'une méthodologie est primordiale pour réussir à utiliser les technologies formelles.

Cette section propose de définir ce que nous entendons par la modélisation formelle de systèmes. Nous nous intéressons plus particulièrement à la modélisation de logiciels et de systèmes. Cette partie décrit également une classification des modèles en différents types que nous détaillons. S'offrent alors à nous différentes techniques pour prouver ou assurer que les propriétés du système sont respectées par le modèle construit. Parmi les différentes méthodes formelles disponibles aujourd'hui, nous avons choisi la méthode B pour la réalisation de nos travaux.

La modélisation formelle de systèmes

Un modèle formel est une représentation mathématique d'un système, utilisé pour vérifier ou assurer les propriétés du système considéré. Il est une abstraction mathématique du monde réel obtenue après suppression de certains détails d'implémentation ou après le choix de certaines hypothèses et de caractéristiques essentielles du système.

Certaines méthodes donnent la possibilité de raffiner ce modèle abstrait dans un modèle concret. Ce modèle concret se trouve alors très proche d'une implémentation. Il s'agit alors de traduire ce modèle concret dans un langage de programmation. Après traduction, le code obtenu est exécutable. Il représente l'implémentation formelle du système spécifié, en préservant les propriétés inclues au niveau le plus abstrait. La modélisation mathématique a de nombreux avantages dont les principaux sont :

- une précision dans la spécification formelle supérieure à celle fournie par une description informelle en langage naturel qui est souvent ambiguë,

- des fondements permettant d'énoncer des propriétés et d'étudier le comportement du système dans son ensemble.

La modélisation peut prendre diverses formes, en fonction de la nature des mathématiques retenues pour exprimer et formaliser le système. Dans cette perspective, nous distinguons différents types de modèles comme les modèles mettant en oeuvre des machines d'états abstraits, ceux basés sur des automates, ceux basés sur les logiques ou bien encore les modèles fonctionnels et les langages synchrones.

*** Les modèles basés sur les machines d'états abstraits**

Dans ce type de modèle, le système est représenté par son état et un ensemble d'opérations qui s'appliquent à cet état et le modifient. Chaque opération permet d'effectuer une transition entre les états que peut prendre le système. Plusieurs méthodes formelles utilisent ce formalisme. Parmi lesquelles nous pouvons citer les ASMs [Blass *et al.*, 2006], VDM [Bjorner and Jones, 1978], Z [Abrial *et al.*, 1980] et B [Abrial, 1996a].

- Les **ASMs** (Abstract State Machines), proposées par Gurevich [Blass *et al.*, 2006] sont une méthode formelle pour la spécification et la vérification de systèmes informatiques. Une machine abstraite en ASM est constituée d'un système fini de transitions d'états. Chaque transition d'état est une règle modifiant un état pour obtenir un nouvel état. Ces règles de transitions contiennent une garde et des fonctions de mise à jour. Ainsi, si une règle peut s'appliquer, c'est-à-dire que sa garde est vérifiée, alors la clause relative à la transition est exécutée de manière atomique. L'exécution de cette clause met à jour l'état du système et le processus continue. Il se peut, dans certains cas, que plusieurs règles soient éligibles en même temps. Dans cette condition, une règle parmi celles éligibles est choisie de façon non déterministe. L'état du système est constitué d'une collection de noms représentant diverses parties du système. La mise à jour de l'état consiste à assigner des valeurs à ces noms ou à en modifier la valeur. Les ASMs sont aussi expressifs que les machines de Turing et leur avantage est de pouvoir être utilisés à la fois comme langage de spécification et langage de programmation.
- La méthode **VDM** (Vienna Development Method) [Bjorner and Jones, 1978] développée depuis les années 70 par Cliff Jones est un langage de spécification. Une spécification en VDM consiste à la définition de types, de fonctions, et d'opérations. La spécification commence par une description globale du système, ensuite par raffinements successifs, on arrive à la spécification finale. Chaque spécification VDM, sans considérer son niveau d'abstraction, peut être vue comme un ensemble de descriptions d'états. Chaque description d'états est constituée d'un ensemble de variables d'états, des opérations sur ces variables d'états et des invariants sur ces variables d'états définis comme des pré et post conditions. Ces invariants constituent l'argument de correction de la spécification permettant d'assurer la correspondance avec la spécification précédente.

  La consistance est analysée par la vérification, à chaque niveau d'abstraction, de la consistance des définitions. La validation consiste à prouver que les propriétés sont vérifiées par la spécification.
- La méthode **Z** est développée dans les années 70, en parallèle de VDM, avec notamment la participation de Jean-Raymond Abrial qui, quelques années plus tard, élabore la méthode B [Abrial, 2003]. La méthode Z est basée sur la théorie des ensembles et le calcul des prédicats. La spécification finale est obtenue par raffinements successifs en partant de la

forme la plus abstraite du système. Le schéma est la notion de base des spécifications Z. Les schémas sont utilisés pour décrire les états et les opérations du système. La spécification du système est décomposée en schémas qui permettent de décrire les aspects statiques (états et invariants) et dynamiques (opérations, relations entre les entrées et les sorties, changements d'états) du comportement du système.

La méthode B est une amélioration de la méthode Z grâce notamment à l'apport du langage de programmation parallèle CSP (Communicating Sequential Processes) développé par Hoare [Hoare, 1985].

Les méthodes formelles décrites dans cette section couvrent une importante partie du cycle de développement d'un logiciel. Elles ont comme avantage de se baser sur des mathématiques simples à appréhender : la logique du premier ordre et la théorie des ensembles. Cependant, ce ne sont pas des modélisations simples à mettre en oeuvre. Si elles sont simples à comprendre, elles le sont moins à utiliser, nécessitant alors de l'expérience et de la pratique. Il manque un guide permettant de faciliter leur mise en œuvre.

*** Les modèles basés sur les automates**

Ce type de modélisation s'appuie sur une description comportementale du système. Cette description donne les actions et les réactions du système face à différents stimuli. Un automate est un système de transitions d'états finis. Il est constitué d'un ensemble d'états et d'un ensemble de transitions décrivant le flot de contrôle du système. Parmi ces modèles, nous notons LOTOS et SDL et les statecharts.

- **LOTOS** permet la modélisation de comportements séquentiels, de choix, de comportements concurrents et non déterministes [Leduc and Germeau, 2000]. Ce langage traite des spécifications de très haut niveau d'abstraction avec :
  - des données définies sous forme de types abstraits algébriques,
  - la récursivité,
  - l'interruption de communication entre deux processus,
  - un système de communication qui permet de simuler les rendez-vous multiples et les synchronisations.

  Les spécifications en LOTOS décrivent les comportements observables de systèmes. Un système est décrit par une architecture de processus. Ces processus vont communiquer au moyen d'interactions qui peuvent soit être une communication, soit une synchronisation entre plusieurs processus. Les spécifications dans ce langage sont à la fois exécutables et prouvables. Ce langage a été utilisé en industrie et plusieurs spécifications des applications industrielles ont été écrites avec LOTOS. La vérification automatisée de ces spécifications n'a toutefois pas été encore réalisée.
- **SDL** [Worm, 1999] décrit le comportement d'un système sous la forme de stimulu/réaction, étant admis que les stimuli, aussi bien que les réactions, sont des entités discrètes et contiennent de l'information. En particulier, la spécification d'un système est vue comme étant la séquence de réactions associée à une séquence de stimuli. Le modèle de spécification d'un système est fondé sur la notion de machine à états finis étendue. Un système SDL est composé d'un ensemble de blocs. Les blocs sont connectés entre eux et à l'environnement par des canaux. A l'intérieur de chacun des blocs, il y a un ou plusieurs processus.

Un processus est une machine d'états finis étendue fonctionnant de manière concurrente et autonome avec les autres processus. Les processus communiquent de manière asynchrone par l'intermédiaire de signaux. Aucune synchronisation n'est requise entre l'émetteur d'un signal et son consommateur. Un processus peut envoyer (resp. recevoir) des signaux vers (resp. de) l'environnement ou les autres processus d'un même système. Les signaux reçus par un processus sont stockés dans une file de taille supposée non bornée.

SDL est fondé sur les machines d'états finis étendues et sur les types abstraits de données. Le concept de types abstraits de données (qui est utilisé aussi par LOTOS) permet d'avoir des spécifications de données rigoureuses et avec plusieurs niveaux d'abstraction. SDL fait appel à des concepts structurels qui facilitent la spécification des systèmes complexes. Il est ainsi possible de subdiviser la spécification d'un système en unités faciles à gérer qui peuvent être traitées et comprises de manière indépendante.

- **Statecharts** [Harel *et al.*, 1987; Harel and others, 1988] a été introduit comme une notation visuelle pour la représentation de machines à état complexes sous une forme plus synthétique. Les machines à état complexes sont représentées sous forme de combinaisons de machines de plus en plus simples à l'aide des mécanismes *OR* et *AND*. Définis, donc, comme un moyen de modélisation comportemental, statechart nécessite pour être utilisé comme moyen de spécification complet d'être intégré à un outil *CASE* où les aspects structurel et fonctionnel sont modélisés à l'aide d'autres notations. Cela a été réalisé dans le cadre de STATEMATE [i-Logix, 1987] où l'aspect fonctionnel est modélisé par ACTIVITY-CHART et l'aspect structurel par MODULE-CHART. La simulation permet de produire des spécifications beaucoup plus fiables. La notion de temps est gérée à l'aide de la fonction spéciale *timeout(E,N)* qui devient vraie quand N unités de temps se sont écoulées après l'occurrence de l'événement E. Les Statecharts permettent de représenter l'aspect contrôle et les contraintes temporelles quantitatives des systèmes critiques temps réel, mais elles ne permettent pas de faire la vérification et la validation de ces propriétés.

*** Les langages synchrones**

Les langages synchrones possèdent un modèle d'exécution qui suppose une horloge logique globale. Trois principaux langages synchrones ont été développés : Estérel [Berry and Gonthier, ; Berry, 2000], Lustre [Caspi *et al.*, 1987] et Signal [Benveniste *et al.*, 1991]. Ces langages se basent sur un modèle synchrone à réaction instantanée ayant une hypothèse de synchronisme qui permet de considérer le système de façon isolée de son environnement et, par conséquent, simplifie le développement et l'analyse du système. Cette hypothèse se résume à :

Les sorties sont simultanées aux entrées qui les provoquent ou autrement dit le temps de réaction du système est négligeable par rapport au temps d'action de l'environnement.

Dans ces langages, des primitives ont été introduites permettant de raisonner comme si le programme réagissait instantanément aux événements externes. Chaque événement interne du programme est précisément daté par rapport au flot des événements externes, et le comportement du programme est complètement déterministe, tant du point de vue fonctionnel que temporel. En fait, la notion du temps physique (chronomètre) est remplacée par une simple notion d'ordre entre événements : les seules notions importantes sont celles de simultanéité et de précédence entre événements. Le temps physique n'a pas un statut particulier ; il sera perçu comme un événement externe, au même titre que les autres événements en provenance de l'environnement.

La notion d'instant devra être comprise comme celle d'instant logique : l'histoire d'un système est une succession totalement ordonnée d'instants logiques, à chacun desquels *0, 1* ou plusieurs

événements surviennent. Les occurrences d'événements survenant au même instant logique seront considérées comme simultanées, celles survenant à des instants différents seront considérées comme survenant à l'ordre de leurs instants d'occurrence. Tous les processus intervenant dans le système ont la même perception des événements présents et absents à un instant donné. L'hypothèse de synchronisme revient à supposer que le programme réagisse assez vite pour percevoir tous les événements externes au bon ordre. Ces approches s'adaptent bien aux systèmes réactifs synchrones.

Ces langages, bien qu'ils utilisent la notion du temps, ce temps est physique (chronomètre) et il sera considéré comme un événement externe, provenant de l'environnement. C'est la notion de temps multiforme.

*** Les logiques temporelles**

Ces méthodes permettent de décrire le système à l'aide d'un ensemble de règles logiques permettant de spécifier comment le système doit évoluer à partir de certaines conditions. Elles sont inadéquates pour la représentation des aspects structurels du système, mais sont très appropriées pour la description des propriétés du système. Ainsi, elles contribuent à la définition d'un langage de type assertionnel. La validation consistera à prouver des propriétés de haut niveau, données sous forme logique, à l'aide de démonstrateurs de preuves et de systèmes d'inférences. Il existe deux grandes classes de logiques temporelles [Pnueli, 1988] : la logique temporelle linéaire (comme la logique temporelle des actions TLA [Lamport, 1994a]) et la logique temporelle arborescente (comme CTL). Dans la logique linéaire, un instant n'admet qu'un instant successeur, tandis que, dans la logique temporelle arborescente, un instant admet plusieurs instants successeurs. La logique linéaire permet de spécifier des propriétés de programmes déterministes alors que la logique arborescente est particulièrement bien adaptée à la spécification de propriétés dynamiques de systèmes indéterministes.

*** Les modèles fonctionnels**

Les modèles fonctionnels se basent sur la notion de fonction dans le sens mathématique du terme. Dans d'autres approches adoptant le point de vue impératif, le système est représenté par un ensemble d'états et d'actions qui modifient ces états. Pour décrire la succession d'actions transformant l'état du système, nous disposons par exemple des mécanismes comme les séquences ou les répétitions. Dans l'approche fonctionnelle, seules les expressions manipulant des fonctions sans effet de bord sont considérées. Le système est alors représenté par une série de définitions de fonctions. Chaque fonction $f \in E \rightarrow F$ est définie par son domaine (E) et son co-domaine (F), introduisant la notion de type. Les fonctions sont vues comme des objets et peuvent être arbitrairement manipulées : elles peuvent elles-mêmes être les arguments ou les résultats d'autres fonctions. En effet, elles peuvent être composées et la composition est une fonction. Dans ce formalisme, les compositions et les définitions récursives remplacent les séquences et les répétitions. La notion de variable est identique à la définition mathématique, c'est-à-dire que la variable correspond à un lien entre la valeur et le nom. La valeur d'une expression ne dépend que de son contexte textuel et non de son historique de calcul.

La programmation fonctionnelle est un style de programmation qui met l'accent sur l'évaluation des expressions plutôt que sur l'exécution des commandes. Ainsi, une modélisation fonctionnelle peut directement, et naturellement, être implémentée dans ces langages fonctionnels, réduisant ainsi l'écart entre spécification et implémentation. Ces langages sont généralement

considérés comme de bons outils pour la conception et le prototypage des programmes sûrs et corrects. Plus encore, les modèles fonctionnels, utilisés avec la théorie des types, représentent un cadre général de développement pour quelques ateliers dédiés au développement de preuves formelles comme Coq [Coq, 2002], Isabelle [Merz, 1999] ou PVS [PVS, 1992].

- **Coq** est un assistant de preuve développé au sein de l'INRIA dans le projet *Logical* [Coq, 2002]. Il permet de proposer une spécification d'un système sous la forme d'assertions et de théorèmes. Le cadre de développement de Coq facilite la preuve mécanique de ces théorèmes. La preuve formelle ainsi établie assure que le système valide les propriétés exprimées sous la forme de théorèmes. Par la suite, de ces théorèmes et de leurs preuves complètes peut être extrait le programme correspondant au système modélisé. Le code extrait est du code fonctionnel. Coq est basé sur le lambda calcul [de Groote and Salvati, 2004] (formalisme de représentation des fonctions mathématiques avec des principes de logiques qui permet d'exprimer toutes les fonctions calculables) et la logique d'ordre supérieur (les quantificateurs portent sur les prédicats et les fonctions qui sont considérés comme des variables et pas des fonctions). Son utilisation reste l'apanage de spécialistes. En effet, il est assez difficile pour l'ingénieur de développement de se mettre à Coq directement. Le principal problème réside dans la difficile automatisation de la phase de preuve. L'écriture de tactiques facilite cette phase, mais elle reste complexe et nécessite une très bonne expérience dans ce domaine. Des travaux sont en cours pour simplifier cette phase en la rendant plus automatique. Cependant, Coq reste un assistant de preuve et souffre d'un manque de méthodologie permettant sa mise en oeuvre.
- **Isabelle** est un prouveur de théorème développé entre l'université de Cambridge et l'Université de Munich [Nipkow, 2000]. Ce prouveur repose principalement sur les logiques du premier ordre et d'ordre supérieur ainsi que sur le calcul des séquents [de Groote, 1996] et le lambda calcul [de Groote and Salvati, 2004]. Une syntaxe logique et les règles d'inférence sont spécifiées déclarativement, permettant la construction de la preuve des spécifications en une étape. Le prouveur de théorèmes Isabelle est très utilisé en Europe. Par rapport à Coq il propose une plus grande automatisation de la phase de preuve, ce qui en fait un assistant de preuve préféré. Néanmoins, il ne reste pas simple à appréhender par des ingénieurs de développement logiciel. De nombreux travaux sont en cours pour améliorer et fournir une méthodologie simple de mise en oeuvre du prouveur et de sa formalisation.
- **PVS** (*Prototype Verification System*) est un système pour spécifier et vérifier des propriétés sur des systèmes logiciels [PVS, 1992]. Il repose sur la logique d'ordre supérieur classique. De plus les spécifications sont élaborées pour la construction de la preuve de leur validité plutôt que pour l'efficacité du code généré à partir de ces spécifications. Les spécifications en PVS sont organisées en théories qui peuvent être paramétrables. PVS est un prouveur de théorème hautement automatisé, qui facilite son utilisation et le rend très populaire. Cela représente le principal avantage de PVS par rapport à ses concurrents directs en logique d'ordre supérieur que sont Coq et Isabelle.

La modélisation à l'aide des modèles fonctionnels est très attirante grâce à la forme que prend les spécifications et à la force d'expressivité des logiques employées. En effet, le risque avec les techniques comme Z ou B est que la logique employée n'est pas suffisamment expressive pour parvenir à un modèle satisfaisant. Ce risque n'existe plus avec les modèles fonctionnels fondés sur la logique d'ordre supérieur. De plus, ces prouveurs de théorèmes permettent d'exprimer des propriétés de très haut niveau et d'en obtenir une preuve formelle. Ces assistants de preuve offrent des tactiques qui sont des commandes utilisées par l'utilisateur pour guider le système pour une compilation automatique des objets de preuves. Un des principaux avantages de ces

systèmes est que l'utilisateur peut voir et manipuler la preuve, comme c'est le cas pour Coq. Malheureusement, cela ne va pas sans quelques inconvénients. Le premier est la traduction de ces spécifications formelles et de ces modèles fonctionnels en du code exécutable. Cette traduction est difficile voire impossible dans certains cas et des problèmes d'efficacité apparaissent. Le second concerne la haute technicité et l'expérience à acquérir pour pouvoir manipuler ces outils.

1.4.1 Méthodes formelles et cycle de vie

Les méthodes formelles peuvent être utilisées à différentes étapes du cycle de vie, et pour des objectifs différents. Nous listons ces utilisations possibles en cherchant, dans la mesure du possible, à donner des critères sur l'impact sur le cycle de vie.

Phase de spécification

Au niveau de la spécification, c'est à dire du passage des besoins au cahier des charges, l'intérêt des méthodes formelles est clair. Elles obligent à un effort de *formalisation* qui, *a priori*, fournit une aide à la modélisation. En effet, la formalisation peut obliger à un certain degré de questionnement, d'autant plus si cette formalisation est assistée par des outils. Cette phase est souvent la plus critique.

Conception et codage

Partant d'une spécification, les méthodes formelles peuvent être utilisées pour passer progressivement de la spécification au code. Suivant la distance entre les deux, cette étape peut essentiellement prendre deux formes :

- la production automatique de code ;
- l'assistance à la transformation de la spécification au code.

Processus automatique. Le premier type d'outils est adapté lorsqu'il existe un processus systématique de traduction qui, de plus, fournit un code ayant les qualités requises. Ceci signifie d'une part que la spécification est, d'une certaine manière, déjà exécutable et d'autre part qu'elle décrit effectivement le comportement désiré au niveau de l'exécution.

Lorsque l'automatisation est possible, le gain est indéniable puisqu'il supprime un niveau de développement. Ceci revient à offrir au développeur un langage de plus haut niveau.

Processus assisté. Lorsque la spécification est plus abstraite, ou bien n'a pas été conçue comme décrivant le comportement attendu à l'exécution, le processus de transformation peut être assisté formellement. C'est la technique de raffinement. Le développeur doit alors écrire lui-même les spécifications plus précises et la correction sera vérifiée formellement. Le gain est ici moins tangible car l'activité de développement devient plus coûteuse. Néanmoins, si cette activité est mise en place dans une approche globale de la qualité, ce gain peut être évaluable. Ceci a été le cas, par exemple, dans le projet Météor où les tests unitaires ont été supprimés pour les composants développés formellement.

1.5 Besoins d'expression

Dans le cadre de développement des systèmes automatisés, nous disposons d'une partie opérative pilotée par une partie contrôle de manière à ce que le système automatisé satisfait les besoins de l'utilisateur présentés sous forme de propriétés. Les propriétés à spécifier et à vérifier sur les systèmes automatisés sont : sûreté, vivacité, atteignabilité, absence de blocage et équité.

Les propriétés de sûreté ou d'invariance

Une propriété de sûreté énonce que : "quelque chose de mauvais ne se produit jamais". Les propriétés d'invariance ou d'inatteignabilité font partie des propriétés de sûreté. Dans la logique temporelle, les propriétés de sûreté sont exprimées par une formule sous la forme de : $\Box F$, où F est un prédicat d'état. Par exemple, pour le système de climatisation d'une pièce, une propriété souhaitée peut être le maintient de la température dans un intervalle de référence.

Les propriétés de vivacité

Une propriété de vivacité énonce que : "quelque chose de bien arrivera nécessairement". Ce type de propriétés se vérifie en analysant l'ensemble de toutes les traces possibles du système. Nous avons comme propriétés de vivacité les propriétés de fatalité.

Une **propriété de fatalité** énonce que : "sous certaines conditions quelque chose de bien finira par avoir lieu au moins une fois à partir d'un certain état". Dans une logique temporelle elle s'exprime sous la forme de : $F \Rightarrow \Diamond G$ ou $F \rightsquigarrow G$.

Les propriétés d'équité

Une propriété d'équité énonce que, sous certaines conditions, quelque chose aura lieu un nombre infini de fois. Dans la logique TLA, on distingue deux types d'équité : équité faible et équité forte. L'équité faible exprime que si une transition est continuellement activable alors elle est infiniment souvent activée. L'équité forte exprime que si une transition est infiniment souvent activable alors elle sera infiniment souvent activée.

Les propriétés d'atteignabilité

Ces propriétés énoncent qu'une certaine situation peut être atteinte. Nous pouvons exprimer aussi que quelque chose n'est jamais atteignable et nous parlons alors d'inatteignabilité. Dans ce cas, on se situe dans la classe des propriétés de sûreté.

Les propriétés d'absence de blocage

Ces propriétés énoncent que le système ne se trouve pas dans une situation où il lui est impossible de progresser.

1.6 Conclusion

Dans ce chapitre, nous avons présenté les systèmes automatisés ainsi que l'avantage d'utiliser des méthodes formelles pour leur développement. Un système automatisé est un système formé par un composant opératif qui est la partie physique et son environnement et un composant de contrôle (composant logiciel). L'objectif est de développer un système de contrôle de façon que, quand il est composé avec la partie opérative, il satisfait les propriétés exigées par l'utilisateur. Les méthodes formelles permettent de prouver, a priori, qu'un dispositif physique satisfait un cahier des charges imposé. En effet, il s'agit de construire un système automatisé qui contraigne le comportement du système à contrôler.

Les approches formelles telles que la théorie de la supervision [Ramadge and Wonham, 1987] et les techniques de synthèse de la commande qui en découlent, reposent sur des modèles du procédé à commander et des objectifs à satisfaire parfaitement établies. Ces approches se révèlent insuffisantes en pratique lorsque les modèles des objectifs et du procédé à commander sont plus au moins raffinés au cours du cycle de vie, en particulier lors de la conception d'un procédé et de son système de commande. Ainsi, l'objectif de la thèse est donc de mettre en place une méthode formelle permettant de spécifier d'une manière progressive les propriétés attendues d'un système automatisé. De plus, de vérifier l'interaction du système à contrôler avec le système de contrôle avant sa mise en œuvre. Dans les travaux classiques, il y a la modélisation du contrôleur et par la suite il y a son interfacage avec le système à contrôler et le test d'intégrité se fait après la mise en place du contrôleur. Certaines erreurs peuvent être détectées au moment de l'exécution, ce qui peut engendrer des pertes en temps et en argent. Ces erreurs auraient pu être évitées ou réduites si l'interaction entre les deux composants étaient prise en considération à l'étape de spécification et vérification. Pour ce faire, nous proposons dans cette thèse une solution qui permet de synthétiser un système de contrôle en utilisant une méthode formelle et en modélisant aussi, le système à contrôler pour vérifier l'interaction avant la mise en oeuvre du système de contrôle.

Nous avons choisi d'utiliser dans notre travail le B événementiel pour le développement de ce type de systèmes. Le choix de cette méthode répond à un certain nombre de critères, notamment concernant l'applicabilité d'une telle méthode en milieu industriel. Elle a été utilisée dans le projet Météor qui a servi de projet pilote pour cette technologie [Behm *et al.*, 1999]. De plus, elle couvre le cycle complet de développement et repose, comme nous le détaillerons dans le chapitre suivant, sur des fondements assez simples comme la logique des prédicats et la théorie des ensembles. Le fait de couvrir le cycle complet de développement permet en outre de mieux l'intégrer à un cycle industriel déjà existant. Cela rassure les industriels en leur montrant que les méthodes formelles ne sont pas uniquement réservées à un usage académique. Ces deux avantages, l'expérience industrielle et les fondements théoriques de la méthode B, en font un candidat préférentiel pour mener nos travaux.

En plus, vu la complexité des systèmes automatisés et l'interactivité entre ces composants (parties contrôle et opérative), la méthode que nous allons proposer pour le développement de ce genre de systèmes doit se baser sur les techniques de raffinement et de composition. Ces deux paradigmes de spécification permettent de construire le système automatisé d'une manière incrémentale. D'autre part, les systèmes automatisés doivent satisfaire des propriétés de vivacité. De ce fait, l'approche proposée doit prendre en considération la satisfaction de ce type de propriétés. Dans la méthode B, seules les propriétés de sûreté sont considérées, d'où nous proposons d'utiliser un autre formalisme pour la spécification et la vérification des propriétés de vivacité, qui est le langage de modélisation TLA^+ vu sa compatibilité avec le B événementiel. En effet, les deux méthodes se basent sur des systèmes à état-transitions .

Chapitre 2

La Méthode B et ses extensions

2.1 Introduction

Nous étudions à travers ce chapitre l'utilisation d'une méthode formelle particulière : la méthode B dans le but de formaliser et vérifier les systèmes automatisés. Nous nous intéressons en particulier aux extensions apportées à cette méthode pour la spécification et la vérification des propriétés de vivacité.

2.2 Méthode B

La méthode B [Abrial, 1996a], se présente comme une méthode formelle couvrant toutes les phases d'un cycle de développement. En effet, les modèles B (Figure 2.1) expriment différents niveaux d'abstraction allant de la phase de conception préliminaire jusqu'à la phase de codage. Le passage d'une phase à une autre dans un processus de développement en B correspond à un incrément de spécifications en cours de développement.

Depuis quelques années, la méthode B a été utilisée avec succès dans le domaine du transport ferroviaire, notammment dans le cadre du métro sans conducteur METEOR [Behm *et al.*, 1999] réalisé par *Matra Transport International*. Actuellement, ses applications industrielles touchent divers domaines, tels que les cartes à puce [Casset, 2002], le diagnostic automobile [Pouzancre, 2003; Pouzancre and Pitzalis, 2003] et les circuits électroniques [Hallerstede, 2003].

2.2.1 Fondements de la méthode B

La méthode B est une méthode formelle de modélisation et de construction de logiciels proposée par Jean-Raymond ABRIAL ; elle est décrite dans le livre, the B Book [Abrial, 1996a]. Elle s'appuie sur les concepts mathématiques de la théorie des ensembles. Elle propose une démarche qui couvre toutes les étapes de développement d'un logiciel, depuis la spécification jusqu'à l'implementation. Cette méthode est basée sur les notions de machine abstraite, de raffinement vérifiée par la preuve. Le cycle de développement commence par la construction d'une machine abstraite. Cette machine abstraite B est constituée, d'une part, de variables d'état, d'un invariant exprimant des propriétés sur les variables et d'autre part, d'opérations qui décrivent les transformations d'états correspondant à des changements de valeur des variables. Le développement d'un système complexe ne peut se faire en une seule étape [Back and K-Sere, 1989a;

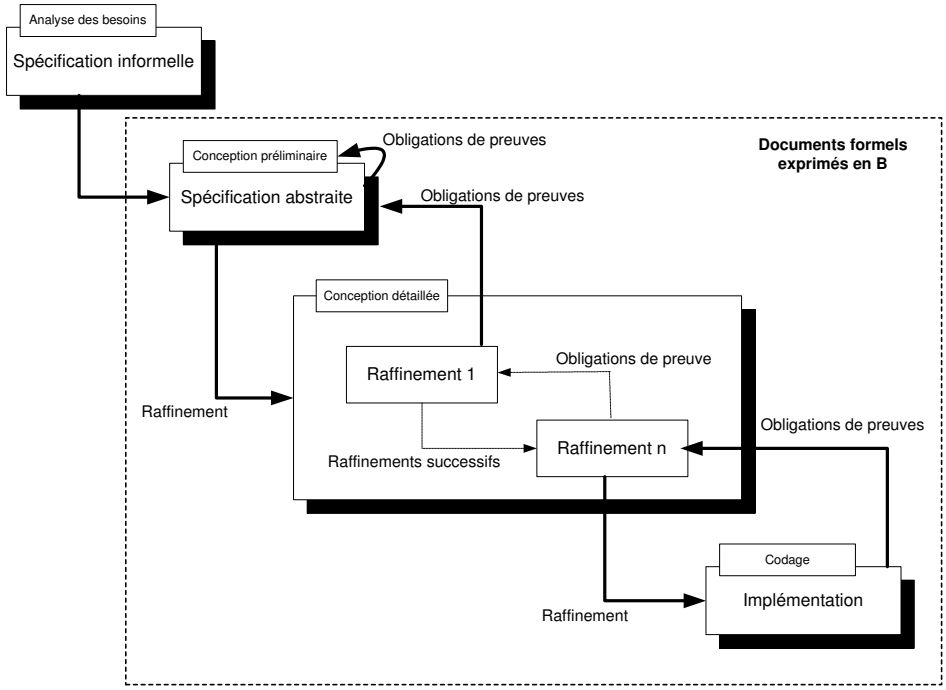


FIG. 2.1 – Processus de développement en B.

Chandy and Misra, 1988] et B propose une méthode de construction incrémentale par raffinements successifs.

L'abstraction est une technique générale employée pour maîtriser la complexité des systèmes, un développement en B consiste, alors, à construire un premier modèle abstrait, dans lequel certains détails ne sont pas pris en compte. Les étapes suivantes consistent à compléter le modèle abstrait pour obtenir un modèle concret. A l'issue de chaque étape, la vérification consiste à prouver des obligations de preuves engendrées par l'outil de développement, l'Atelier B [ClearSy, 2002a].

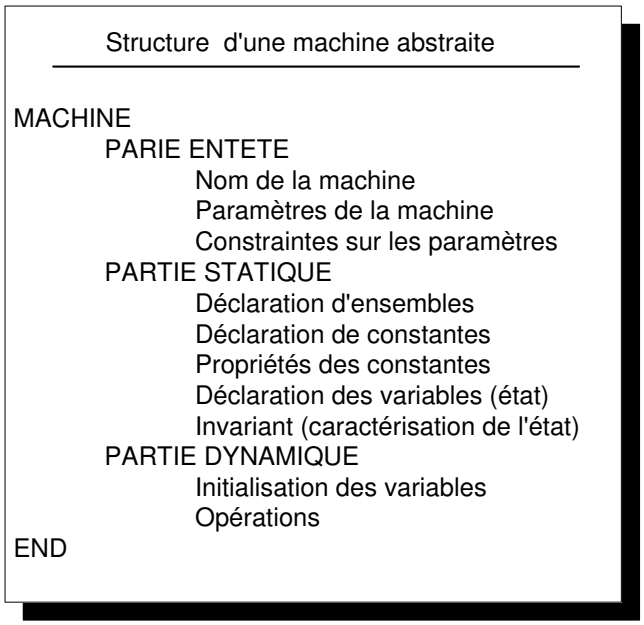
* Machine abstraite

J.-R Abrial compare la notion de machine abstraite en B à une boite noire ayant une mémoire et un certain nombre de boutons. Les valeurs stockées dans cette mémoire représentent l'état de la machine et les boutons sont les opérations mises à la disposition de l'utilisateur et qui ont pour effet de modifier les valeurs stockées en mémoire. Les opérations constituent donc la seule interface de la boite noire pour l'environnement extérieur, tant en entrée qu'en sortie.

Cette notion de machine abstraite est une notion fondamentale en B. Elle correspond à l'élément de structuration de base des spécifications et elle est composée de trois parties : la partie entête, la partie statique et la partie dynamique.

- La partie entête. elle permet l'identification de la machine abstraite. Elle contient la clause MACHINE, décrivant le nom de machine suivi éventuellement de paramètres, ainsi que la clause CONSTRAINTS qui donne les propriétés des paramètres.

- La partie statique. Elle regroupe les déclarations d'ensembles (clause SETS), de



constantes (clause CONSTANTS) et de variables (clause VARIABLES). Ces déclarations sont complétées par un ensemble de prédicats décrivant les propriétés des constantes (clause PROPERTIES) ainsi que les invariants (clause INVARIANT) qui explicitent précisément les propriétés qui doivent être toujours satisfaites par l'état de la machine. Les données définies au niveau de ces clauses sont spécifiées en utilisant des formules de la logique du premier ordre et des notations mathématiques de la théorie des ensembles.

- La partie dynamique. Elle décrit l'évolution de l'état de la machine. Ceci comprend son initialisation et les opérations offertes par la machine. Les transitions entre états peuvent s'effectuer de manière non-déterministe. Pour ce faire, le langage défini est celui des substitutions généralisées qui est une notion spécifique à B. Dans d'autres techniques de spécification formelle orientée modèle telles que Z et VDM, la partie opérationnelle est décrite par des prédicats avant/après spécifiant les états avant et après l'activation d'une opération. En B, nous retrouvons ces notions ; en effet, les substitutions généralisées en B peuvent être réécrites en termes de prédicats avant/après.

*** Substitutions généralisées**

Dans une machine B, une substitution est un moyen de représenter une transition sur l'état du modèle B. La machine est initialement dans un "état pré" et se retrouvera, après la substitution dans un "état post". Le langage des substitutions généralisées est défini pour exprimer ce mécanisme d'évolution de données.

Deux substitutions élémentaires existent, l'une *skip* pour conserver l'état courant et l'autre pour le modifier : la substitution simple (affectation) $x := E$ a pour effet d'affecter à la variable x la valeur de l'expression E. Les substitutions complexes sont construites à partir de ces deux substitutions élémentaires et des opérateurs de composition tel que le séquencement (;), le parallélisme (||), le conditionnel (if ... the ... end), la boucle (while ... do ... invariant ... variant), etc.

Toutes les substitutions généralisées ne sont pas nécessairement déterministes. Une substitution déterministe spécifie une seule transition, alors qu'une substitution non-déterministe spécifie un ensemble de transitions potentielles.

Une substitution généralisée est considérée comme un transformateur de prédicats. Les prédicats auxquels nous nous référons portent sur les variables d'état de la machine considérée. Étant donné une substitution S, on peut transformer tout prédicat P en un prédicat Q de la manière suivante : $Q = [S]P$ (la substitution S établit le prédicat P). Conformément à la terminologie de Dijkstra en 1975 [Dijkstra, 1975], le prédicat Q est défini comme étant la plus faible pré-condition qui garantit que l'execution de S termine dans un état où le prédicat P est vrai. Par exemple, dans le cadre de l'instruction particulière d'affectation, la formule $[x := e]P$ dénote la plus faible pré-condition qui garantit que le prédicat P est vrai après la substitution uniforme de la variable x par l'expression e. En d'autres termes, $[x := e]P$ dénote la formule obtenue après remplacement de toutes les occurrences libres de x dans P par e. Ceci sera aussi noté : $P[e/x]$.

Les règles d'application d'une substitution à un prédicat pour calculer la plus faible précondition sont présentés dans le tableau 2.1. Ces substitutions correspondent à :

- La substitution simple : correspond à l'affectation usuelle.
- L'affectation multiple : permet la spécification de substitution multiple où les variables x et y sont distinctes.
- La substitution pré-conditionnée : fixe les pré-conditions sous lesquelles une opération est appelée. En effet, si la pré-condition P est fausse, alors le prédicat $P \wedge [S]R$ est faux et la substitution S peut ne jamais se terminer.
- La substitution gardée : définit différents comportements possibles en fonction de la validité de gardes. Une spécification gardée spécifie une opération qui est activée sous une hypothèse correspondant à la garde.
- La substitution choix borné : permet de définir un nombre fini (ou borné) de comportements possibles sans préciser lequel sera effectivement implanté.
- La substitution choix non-borné : permet d'utiliser dans la substitution s la donnée abstraite z. Cette substitution définit un comportement no-déterministe dépendant du choix de z. On peut alors dire que la substitution choix non-borné est une généralisation de la substitution choix borné.
- La substitution séquencée : correspond à l'application en séquence de deux substitutions. $[S;T]P \Leftrightarrow [S][T]P$, signifie que le prédicat obtenu par application de la substitution séquence $S;T$ sur le prédicat P est le prédicat obtenu par application de S sur le résultat de l'application de T sur P.
- La substitution parallèle : correspond à l'exécution simultanée de deux substitutions. Le caractère de simultanéité dénote le fait que les substitutions doivent pouvoir se réaliser indépendamment l'une de l'autre.
- La boucle : réalise la substitution S tant que le prédicat P reste vrai. I est l'invariant de la boucle qui permet de prouver qu'à chaque pas la boucle est possible et qu'elle donne bien le résultat produit à la sortie. V est le variant de la boucle. C'est une expression entière qui permet de démontrer que la boucle se termine au bout d'un nombre fini d'itérations. Pour cela, il faut prouver que V est une expression entière positive qui décroît strictement à chaque itération.

Nom de la substitution	**Notation mathématique**	**Sémantique (calcul du WP)**
sans effet	$skip$	$[skip]R \Leftrightarrow R$
affectation	$x := e$	$[x := e]R \Leftrightarrow R[e/x]$
affectation multiple	$x, y := e, f$	$[x, y := e, f]R \Leftrightarrow [z := f][x := e][y := z]R$
pré-conditionnée	$P \mid S$	$[P \mid S]R \Leftrightarrow P \wedge [S]R$
gardée	$P \Longrightarrow S$	$[P \Longrightarrow S]R \Leftrightarrow (P \Rightarrow [S]R)$
choix borné	$T [] S$	$[T [] S]R \Leftrightarrow [T]R \wedge [S]R$
choix non bornée	$@z.S$	$[@z.S]R \Leftrightarrow \forall z.[S]R$
séquence	$S; T$	$[S; T]R \Leftrightarrow [S][T]R$
boucle	While P do S	$I \wedge P \Rightarrow [S]I$
	VARIANT V	$I \Rightarrow V \in \mathbb{N}$
	INVARIANT I	$I \wedge P \Rightarrow [n := V][S](V < n)$
	END	$I \wedge \neg P \Rightarrow R$
Avec : x et y des variables ; e et f des expressions ; s et T des substitutions ; P et R des prédicats. Et où z est non-libre dans R.		

TAB. 2.1 – substitutions généralisées et sémantique en transformateur de prédicat.

Pour toutes les substitutions indiquées par le tableau 2.1, il s'agit de la plus faible précondition, sauf pour la boucle. En ce qui concerne celle-ci, les formules indiquées servent à vérifier que l'invariant est suffisamment expressif pour vérifier P. Il s'agit donc d'une précondition suffisamment expressive, plutôt qu'une plus faible précondition.

Nous introduisons également les prédicats permettant de vérifier certaines propriétés des substitutions. Une substitution peut être caractérisée par sa terminaison, sa faisabilité et l'état avant et après son exécution. Pour ce faire, B définit le prédicat de terminaison, le prédicat de faisabilité et le prédicat avant/après pour une substitution S comme suit :

- Prédicat de terminaison - trm(S)
Ce prédicat représente l'espace des états pour que S puisse établir une postcondition. Dans le cas où S ne termine pas, l'état post ne vérifie aucun prédicat. Le prédicat $trm(S)$ est défini par : $trm(S) \equiv [S]btrue$ ou encore $trm(S) \equiv [S](x = x)$. La terminaison des substitutions généralisées énoncées précédemment se déduit de cette définition.

- Prédicat avant/après - prd_x(S)
Ce prédicat caractérise toutes les valeurs possibles des variables d'état après l'exécution d'une substitution. La valeur avant d'une variable est notée x et la valeur après est notée x'. Le prédicat $prd_x(S)$ est défini par : $prd_x(S) \equiv \neg[S](x' \neq x)$.

- Prédicat de faisabilité - fis(S)
Ce prédicat décrit le domaine de prd_x(S). En effet, si S n'est pas faisable, alors l'ensemble des états post est vide (les éléments vérifient n'importe quel caractère). Il est défini comme suit : $fis(S) \equiv \neg[S](bfalse) \equiv < S > (btrue)$.

Une substitution s portant sur une variable x est faisable si et seulement si une valeur après x' peut être calculée à partir de la valeur avant x. Ce qui revient à définir la faisabilité en termes de prédicats avant/après : $fis(S) \Leftrightarrow \exists x'.(prd_x(S))$.

Toute substitution peut être décrite en termes de *trm*, *fis* et *prd*. Pour pouvoir faire le lien entre le langage des substitutions généralisés (GSL) et la syntaxe B usuelle, nous indiquons dans le tableau 2.2 la manière dont s'écrivent les substitutions de B avec les GSL.

Dans la substitution "devient tel que", $x\$0$ dénote la valeur de la variable x avant l'application de la substitution.

Nom de la substitution	**Substitutions B**	**Substitutions généralisées**
Substitution précondition	PRE P THEN S END	$P \mid S$
Substitution choix borné	CHOICE S OR T END	$S[]T$
Substitution conditionnelle	IF P THEN S ELSE T END	$P \Rightarrow S[]\neg P \Rightarrow T$
Substitution sélection	SELECT WHEN P THEN S ... WHEN Q THEN T ELSE R	$P \Rightarrow S[]...[]Q \Rightarrow T[]R$
substitution variable locale	VAR x IN S END	$@x.S$
Substitution choix non borné	ANY x WHERE P THEN S END	$@z.(P \Longrightarrow S)$
Substitution devient tel que	$x : (P)$	$@x'.(x' \in E \Longrightarrow x := x')$
Substitution devient élément de	$x :\in E$	$@x'.([x, x\$0 := x', x]P \Longrightarrow x := x')$

TAB. 2.2 – Correspondance des substitutions B avec les substitutions généralisées.

Enfin, notons que toute substitution peut s'écrire sous la forme normalisée suivante :

$$S \equiv @x'.(prd_x(S) \Longrightarrow x := x')$$

Où x est l'ensemble des variables de la machine et x' représente l'ensemble des valeurs de x après l'exécution de S.

* Obligations de preuve d'une machine abstraite

Définition. Une obligation de preuve est une formule mathématique à démontrer afin de vérifier la correction d'un modèle B. La théorie B indique quelles sont les obligations de preuve à vérifier pour assurer la correction d'un modèle B donné. Dans cette optique, les obligations de preuve sont une aide au processus de vérification.

La génération des obligations de preuve s'appuie sur le calcul de la plus faible pré-condition. Il s'agit de vérifier que l'invariant est respecté aussi bien par l'initialisation que par les opérations de la machine.

Obligations de preuve de l'initialisation. Soit I l'invariant (clause Invariant), R les propriétés (clause PROPERTIES) et C les contraintes (clause CONSTRAINTS) de la machine, et soit S la substitution définissant l'initialisation de la machine alors l'initialisation est correcte si et seulement si :

$$R \wedge C \Rightarrow [S]I$$

En d'autres termes, il s'agit de démontrer que pour n'importe quel état s satisfaisant les propriétés les contraintes de la machine, l'exécution de la substitution d'initialisation S à partir de s aboutit à un état qui satisfait l'invariant I.

Obligations de preuve des opérations. Les obligations de preuve des opérations ont pour objectif de vérifier que l'invariant de la machine est vrai après application de la substitution de l'opération sachant que l'invariant était vrai avant l'appel de l'opération. Étant donné que la forme générale d'une opération en B est : $r \longleftarrow nom\_op(p)$ =PRE P THEN S END ; alors l'obligation de preuve associée s'exprime par la formule suivante :

$$I \wedge R \wedge C \wedge P \Rightarrow [S]I$$

2.2.2 Raffinement

Le raffinement en B est une technique de développement incrémental permettant de préciser progressivement les données et les opérations de la spécification abstraite de départ. L'objectif de cette technique est de passer progressivement d'une spécification non-déterministe abstraite à une spécification concrète déterministe, automatiquement traduisible dans un language de programmation.

*** Fondements**

Le raffinement d'une machine abstraite est une transformation produisant une machine qui conserve la même interface (opérations) et le même comportement que la machine abstraite initiale. Elle permet d'une part, de reformuler la machine en une expression de plus en plus concrète, et d'autre part, de l'enrichir en y ajoutant de nouvelles données et invariants. En B, le raffinement porte sur :

- les données : où les données ou variables abstraites définies dans la clause VARIABLES peuvent être conservées, disparaître ou modifiées et de nouvelles variables être introduites. Dans ce dernier cas, un invariant, dit de liaison (ou de collage), est défini en vue de spécifier la relation entre données abstraites et données concrètes (celles du raffinement).
- les opérations : chaque opération raffinée doit réaliser ce qui est spécifié dans l'abstraction, à l'aide des données du raffinement et de substitutions plus concrètes et plus déterministes ;

*** Obligations de preuve du raffinement**

Comme pour les obligations de preuve d'une machine abstraite, nous donnons les obligations de preuve de l'initialisation et celles des opérations du raffinement. Soit I l'invariant de composant raffiné (ou abstrait), R ses propriétés et C ses contraintes, et soit S la substitution définissant son initialisation.

Obligations de preuve de l'initialisation.
Nous désignons par J l'invariant du raffinement, et par R_r les propriétés du raffinement. Soit S_r la substitution définissant l'initialisation du raffinement alors l'obligation de preuve de l'initialisation du raffinement revient à vérifier que le prédicat suivant est satisfait :

$$\boxed{R \wedge C \wedge R_r \Rightarrow [S_r]\neg[S]\neg J}$$

Il s'agit, de vérifier que l'initialisation du raffinement S_r établit l'invariant du raffinement J sans contredire l'initialisation S du composant raffiné, et ce, sous l'hypothèse formée par la conjonction des prédicats R, C et R_r.

Obligations de preuve des opérations.

- Preuve de la pré-condition : soit P la pré-condition de l'opération abstraite et Q la pré-condition de l'opération concrète alors la preuve de la pré-condition de l'opération concrète se ramène à :

$$\boxed{R \wedge C \wedge R_r \wedge I \wedge J \wedge P \Rightarrow Q}$$

- Preuve de l'opération : soit L la substitution de l'opération concrète et K celle de l'opération abstraite alors l'obligation de preuve de l'opération est la suivante :

$$R \wedge C \wedge R_r \wedge I \wedge J \wedge P \Rightarrow [L]\neg[K]\neg J$$

2.3 Méthode B-Événementiel

Le B événementiel est une extension de B [Abrial, 1996b] conçue pour modéliser des systèmes réactifs [Abrial, 2003]. Dans cette extension, les concepts de machine abstraite et d'opérations sont respectivement remplacés par ceux de système abstrait et d'événements. Ces systèmes abstraits peuvent être vus comme des systèmes fermés qui modélisent le système et son environnement, et dont l'état peut évoluer par l'application des événements. Ceux-ci sont décrits en terme d'actions gardées, et ils sont susceptibles d'être déclenchés quand leur garde devient vraie. L'un des événements déclenchables peut alors être appliqué et l'état du système change en fonction de l'action associée à l'événement. Cette notion de système à événements est inspirée des actions de Back [Back and K-Sere, 1989b], ou des commandes gardées de Dijkstra [Dijkstra, 1975].

En B événementiel, les événements sont constitués d'un prédicat d'état appelé garde et d'une substitution ayant pour but de modifier la valeur de variables. Dans un état donné, plusieurs des gardes du système peuvent être vraies. L'une d'elles est alors déclenchée et la substitution associée effectuée.

2.3.1 Différences avec B

Les avantages du B événementiel par rapport au B classique sont les suivants :

- Un modèle B n'est plus "appelable" : il est désormais un système autonome,
- Chaque modèle est composé de plusieurs actions activables lorsque les gardes de celles-ci sont déclenchables. Ces actions gardées sont appelées *événements*
- Un modèle fonctionne tant qu'un de ses événements est activable. Lorsque le contraire arrive, selon le point de vue, soit le calcul fait par le système est terminé, soit le système est bloqué. Cette dernière propriété n'est pas désirable lorsque le système est conçu pour justement ne pas s'arrêter,

Les substitutions autorisées en B événementiel sont moins nombreuses. La raison de cette réduction du nombre de substitutions est liée au fait que le B événementiel oblige à raisonner en termes d'activation d'événements plutôt qu'en termes d'appel de sous-routines. Ceci a pour conséquence que toutes les substitutions décrivant le flot de contrôle en B classique ne se retrouvent pas en B événementiel.

2.3.2 Définition d'un modèle B événementiel

Un système B ou modèle B est représenté comme suit :

La clause **SETS** permet de définir des ensembles (énumérés ou non) que va manipuler le modèle. La clause **CONSTANTS** définit les constantes du modèle et la clause **VARIABLES** définit les variables du modèle (l'état du système). La clause **PROPERTIES** permet d'exprimer des propriétés (typage, domaine, etc.) à propos des constantes du modèle. Les variables sont contraintes par un invariant, c-à-d. une conjonction de propriétés statiques qui doit toujours

Structure d'un modèle B

```
MODEL
        PARIE ENTETE
                <Nom du modèle>
        PARTIE STATIQUE
                SETS                <Déclaration d'ensembles>
                CONSTANTS           <Déclaration de constantes>
                PROPERTIES          <Propriétés des constantes>
                VARIABLES           <Déclaration des variables (état)>
                INVARIANT           <Invariant (caractérisation de l'état)>
        PARTIE DYNAMIQUE
                INITIALISATION      <Initialisation des variables>
                EVENTS              <événements>
END
```

être vérifiée par le modèle. L'initialisation et les événements sont décrits à l'aide du langage des *substitutions généralisées*, une sorte d'extension du langage des commandes gardées de Dijkstra. La clause **INITIALISATION** initialise les variables du modèle. La clause **EVENTS** donne la liste des événements réalisés par le modèle. Chaque événement est en fait une substitution généralisée, avec une garde et modifiant les variables concernées par son action.

*** Événements**

Un événement modélise une transition discrète du système dynamique modélisé. Il correspond à un changement d'état et il se produit instantanément. A chaque événement E sera associé une garde et une substitution. La garde et la substitution permettent de définir le prédicat "avant-après" de l'événement E.

Les événements peuvent avoir l'une des trois formes figurant dans le tableau ci-dessous. La sémantique formelle d'un événement s'exprime à partir de la sémantique de la substitution autrement dit de la relation qui existe entre la valeur des variables avant et après le déclenchement de l'événement. On note $BA(x, x')$ un tel prédicat où x et x' désignent respectivement la valeur des variables avant et après l'exécution de la substitution.

Evénement	*Prédicat"avant-après"*	*Garde*
$BEGIN\ x\ :\ P(x_0, x)\ END$	$P(x, x')$	TRUE
$SELECT\ G(x)\ THEN\ x : Q(x_0, x)$	$G(x) \wedge Q(x, x')$	$G(x)$
$ANY\ t\ WHERE\ G(t, x)$ $THEN\ x\ :\ R(x_0, x, t)\ END$	$\exists t.(G(t, x) \wedge R(x, x', t))$	$\exists t.G(t, x)$

Avec : G désigne un prédicat, t un terme et P, R des substitutions généralisées.

TAB. 2.3 – Formes des événements.

*** Les obligations de preuve pour un modèle B événementiel**

Deux types d'obligations de preuve existent en B. D'une part, les obligations de preuve de correction du modèle : il s'agit de vérifier que les invariants spécifiés dans le modèle sont bien respectées et d'autre part il y a des obligations de preuve de raffinement permettant de s'assurer que le système est un raffinement correct du système abstrait spécifié dans la clause REFINES. Il peut y avoir d'autres obligations de preuve provenant de la clause ASSERTIONS. Dans ce cas les obligations de preuve consistent à prouver les assertions avec hypothèses les propriétés de la clause PROPERTIES et les invariants du modèle.

Preuve des assertions. Une assertion est une formule sur les variables et les constantes du système qui doit être prouvée sous l'hypothèse de l'invariant et des propriétés des constantes et des ensembles abstraits (clause PROPERTIES). En notant $I(x)$ l'invariant et $P(s,c)$ le contenu de la clause PROPERTIES pour chaque assertion $A(x,s,c)$, on doit prouver la propriété ci-dessous. Les variables du modèle sont x, les ensembles abstraits sont s et les constantes sont c.

$$\boxed{P(s,c) \wedge I(x) \Rightarrow A(x,s,c)}$$

Preuve de correction de l'invariant. L'invariant du système $I(x)$ doit être satisfait quelle que soit l'évolution du système. L'invariant contient entre autre le type des variables. L'évolution du système étant modélisé par les événements, il faut s'assurer que tous les événements préservent l'invariant. Il faut aussi s'assurer que l'initialisation établit l'invariant. Pour ce faire, les deux obligations de preuve suivantes sont générées automatiquement par les outils supportant le B événementiel.

- L'événement d'initialisation est une substitution généralisée notée $x : Init(x)$ où $Init$ est le prédicat vérifié par les variables x à l'issue de l'initialisation. L'obligation de preuve de l'initialisation s'exprime de la manière suivante :

$$\boxed{Init(x) \Rightarrow I(x)}$$

- Chaque événement doit préserver l'invariant. Si $BA(x,x')$ est le prédicat *"avant-après"* de l'événement alors l'obligation de preuve est la suivante :

$$\boxed{I(x) \wedge BA(x,x') \Rightarrow I(x')}$$

2.3.3 Raffinement

Le raffinement d'un système est une opération du développement qui consiste à créer un nouveau système (plus concret) qui "simule" le système (abstrait). Le système "raffiné" qu'on appelle aussi raffinement (*refinement*) est relié au système qu'il raffine par des propriétés qui lient les variables des deux systèmes et qui constituent ce qu'on appelle l'*invariant de collage* et qu'on note $J(x,y)$, où x sont des variables du système abstrait et y des variables du système concret. A tout événement du système raffiné correspond un événement du système abstrait, la garde d'un événement concret étant plus forte et sa substitution plus déterministe que celles de l'événement correspondant du système abstrait. Le raffinement induit les obligations de preuves suivantes :

- Raffinement de l'Initialisation
Il faut que l'initialisation concrète $y : Init(y)$ corresponde à une initialisation abstraite $x : Init(x)$ et il faut qu'elle établisse l'invariant. L'obligation de preuve générée est la suivante :

$$Init(y) \Rightarrow \exists x.(Init(x) \wedge J(x, y))$$

- Raffinement des événements
Soit un événement abstrait dont le prédicat "avant-après" abstrait est $BAA(x, x')$ et le prédicat "avant-après" concret est $BAC(y, y')$, l'obligation de preuve générée est la suivante :

$$I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow \exists x'.(BAA(x, x') \wedge J(x', y'))$$

Cette obligation dit bien que sous couvert de l'invariant abstrait $I(x)$ et concret $J(x, y)$, un pas concret $BAC(y, y')$ peut être réalisé ($\exists x'$) par un pas abstrait $BAA(x, x')$. De plus cette obligation de preuve prouve l'invariant concret $J(x, y)$ car le pas concret $BAC(y, y')$ préserve cet invariant.

- Raffinement de nouveaux événements
Les nouveaux événements doivent raffiner *skip*. Donc l'événement abstrait a comme prédicat "avant-après" $x' = x$, l'obligation de preuve est la suivante :

$$I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow J(x, y')$$

- Raffinement de modèles

1. Le modèle concret ne doit pas se bloquer plus souvent que le modèle abstrait afin de garantir que le raffinement n'introduit pas de nouveaux comportements bloquants :

   $$I(x) \wedge J(x, y) \wedge grds(AM) \Rightarrow grds(CM)$$

   où $grds(AM)$ est la disjonction des gardes des événements abstraits et $grds(CM)$ est la disjonction des gardes des événements concrets.

2. Les nouveaux événements ne doivent pas garder le contrôle indéfiniment afin de garantir que le système raffiné ne diverge pas plus que le système abstrait. Pour cela, il y a ajout d'une clause VARIANT qui spécifie un *variant* $V(y)$ (naturel), qui doit décroître strictement à chaque activation de chaque nouvel événement afin de garantir l'absence de cycles formés uniquement de nouveaux événements. Les obligations de preuve sont les suivantes :

   $$I(x) \wedge J(x, y) \Rightarrow V(y) \in N$$
   $$I(x) \wedge J(x, y) \wedge BAC(y, y') \Rightarrow V(y') < V(y)$$

2.3.4 Exemple

Soit l'exemple d'une horloge repris de Leslie Lamport [Lamport, 1996]. Ce système représente un modèle formel B d'une horloge. Le premier modèle (modèle *Clock*) modélise les heures (variable *hour*). Cette variable est comprise entre 0 et 23. L'événement *update_h* incrémente l'heure et l'événement *reset_h* met l'heure à zéro.

MODEL *Clock*
VARIABLES
hour
INVARIANT
hour ∈ *0..23*
INITIALISATION
hour :∈ *10..16*
EVENTS
update_h = *WHEN hour <23*
THEN hour :=hour+1
END ;
reset_h = *WHEN hour =23*
THEN hour :=0
END
END

REFINEMENT *Clockhm*
REFINES *Clock*
VARIABLES *h,minute*
INVARIANT
minute ∈ *0..59* ∧ *h=hour*
INITIALISATION
h :=11 || *minute* :∈ *0..45*
EVENTS
update_h = *WHEN hour <23* ∧ *m = 59*
THEN hour :=hour+1 || *m := 0*
END ;
reset_h = *WHEN hour =23* ∧ *m = 59*
THEN hour :=0 || *m := 0*
END ;
update_m = *WHEN m <59*
THEN m :=m+1
END
END

Dans le raffinement (*Clockhm*), nous ajoutons l'événement *update_m* qui modélise l'incrémentation des minutes. Le modèle obtenu produit un raffinement de notre modèle des heures en ajoutant les pas de calcul pour les minutes.

2.3.5 Avantages de la méthode B-Événementiel

Un des avantages de la méthode B-événementiel est qu'elle permet plus de flexibilité et d'expressivité que les langages d'entrées des model-checkers tout en nécessitant moins d'interaction avec l'utilisateur que les prouveurs de théorèmes tel que PVS. Aussi, aucune supposition n'est faite quant à la taille du système à modéliser contrairement au model checking. Le raffinement, nous permet d'enrichir notre modèle graduellement, et de renforcer peu à peu notre invariant, alors qu'en général en model-checking on part directement d'un modèle très complexe du système à vérifier. La complexité du système est donc distribuée, et les preuves étapes par étapes sont plus faciles. Le raffinement permet d'ailleurs d'assurer un meilleur dialogue entre la personne concevant le modèle et son mandataire, et de réparer les erreurs en cours de développement. Ainsi on valide les modèles étape par étape, plutôt que sur un unique modèle final construit directement.

2.3.6 Limites de la méthode B-Événementiel

La méthode B permet d'établir des preuves de propriétés d'invariance, par conséquent, nous pouvons vérifier la correction d'une spécification, mais aussi sa complétude. Ces propriétés peuvent être exprimées en faisant appel à la théorie des ensembles. En revanche, la méthode B ne permet pas d'exprimer explicitement des propriétés temporelles (fatalité, équité, etc.).

2.3.7 Outil support de B : l'Atelier B

La méthode B est supportée par l'Atelier B [ClearSy, 2002b]. Il implante des fonctionnalités telles que la gestion de projet, la génération des obligations de preuve et leur preuve partielle, la traduction automatique en un langage de programmation, l'animation de spécification et la

génération de documents. L'existence de cet outil a permis une utilisation de la méthode dans le milieu industriel. Les outils sont relativement puissants mais ne contournent pas les résultats de décidabilité et d'indécidabilité liés aux théories logiques ; si une obligation de preuve reste non prouvée à la suite de l'activation du prouveur automatique, alors, soit elle est vraie mais restée non prouvée, soit elle est fausse et non prouvable. L'Atelier B supporte également une grande partie de la méthode B-événementiel. Il est commercialisé par la société Stéria.

L'outil fait le type-checking, qui est une vérification de la cohérence des "types" utilisés. Il génère également, des obligations de preuve qu'il prouve d'une manière automatique ou intéractive. Les preuves "automatiques" sont effectuées selon divers niveaux de force. Plus la force est élevée (entre 0 et 3), et plus le prouveur passe de temps sur chaque obligation de preuve, tout en utilisant plus de règles. L'ensemble des règles du prouveur a été entièrement prouvé. Le prouveur interactif, par contre, requiert un temps d'apprentissage et d'assimilation. Il permet d'ajouter des hypothèses intermédiaires (ou lemmes) à l'ensemble des hypothèses considérées pour faire la preuve. Il permet également de raisonner par contradiction, d'instancier des hypothèses quantifiées universellement et de raisonner par cas.

Click_n_Prove est une interface d'aide à la preuve interactive pour l'atelier B ou B4free. *ProB* [Leuschel and Butler, 2003a] est un outil de model-checking de spécifications B.

2.4 Les extensions apportées à la méthode B

Plusieurs extensions ont été apportées à la méthode B. Il y a globalement deux manières d'étendre B :

- étendre la sémantique de B, ou bien
- associer B à un autre formalisme pour bénéficier des avantages de chacun.

Ces mécanismes d'extension sont détaillés dans les sections ultérieures. l'objectif étant ici de donner simplement un point de vue global.

Extensions sémantiques. Les extensions sémantiques faites à B peuvent se faire à plusieurs niveaux : les substitutions (la substitution parallèle ||) [Bodeveix *et al.*, 1999], la logique utilisée ou au niveau des machines B en les contraignant à avoir des traces d'exécution précises [Lano *et al.*, 1996c].

Extensions par d'autres formalismes. Les extensions par d'autres formalismes se placent plutôt dans la catégorie des adjonctions à B. Nous citons les travaux de Treharne et Schneider [Schneider and Treharne, 2002] qui montrent qu'il est possible d'utiliser CSP pour décrire le contrôle d'exécution des opérations de machines B. Les travaux de Bodeveix [Bodeveix *et al.*, 2004] qui utilise un fragment TLTL [Zhang and Mackworth, 1995] (Timed linear temporal logic) pour exprimer des propriétés temporelles simples. Les travaux de Hammad [Hammad *et al.*, 2003] qui montre qu'il est possible d'associer un automate temporisé à un modèle B événementiel pour exprimer des contraintes temporelles (progression, changement d'état, durée).

2.4.1 Extensions de la méthode B - Traitement du temps

Les diverses extensions présentées ci-après présentent plusieurs manières d'associer un flot de contrôle à un projet B ou event B. Elles ne se veulent pas exhaustives, mais donnent un aperçu

assez complet des techniques utilisées pour y parvenir.

Étendre la logique
Lano et Dick [Lano *et al.*, 1996c] proposent d'étendre les clauses de B par les suivantes :

- Une clause TRACES permettant d'exprimer la trace d'exécution des operations de la machine. Ces traces sont par exemple de la forme $(a; b; c)^*$, qui signifient que la machine n'a le droit que d'exécuter les operations a, b et c, consécutivement, et ce un nombre arbitraire de fois.
- Une clause GUARDS permettant d'empêcher l'exécution de certaines operations si elles ne vérifient pas une certaine contrainte. Cette clause semble a priori superflue, puisqu'il est déjà possible de spécifier des gardes en B. En fait, la différence se situe dans le fait que cette garde peut aussi contenir un connecteur **count** énumérant le nombre d'exécutions de l'opération. Cela permet ainsi d'exprimer des contraintes simples à propos de l'historique de l'opération ainsi gardée.
- Une clause THREADS associant à des opérations des gardes et des commandes supplémentaires : lorsqu'une opération est appelée, la garde doit être vérifiée à ce moment, sinon l'appelant est bloqué tant qu'elle n'est pas vérifiée. Puis l'opération est exécutée, et lorsqu'elle est finie elle rend la main à la machine appelante (qui était bloquée tant que l'opération s'exécutait). Ensuite la commande correspondante est exécutée. De cette manière, la commande fonctionne en même temps que la suite des opérations dans la machine appelante. Ce mécanisme permet de définir un opérateur pouvant faire fonctionner deux substitutions en parallèle.
- Une clause HISTORY contenant des formules de logique temporelle discrète, pour exprimer des contraintes d'équité et de vivacité sur les opérations de la machine.

Ils montrent ensuite que cette extension préserve le raffinement de B. Ils présentent une étude de cas (la cellule de production) reprenant toutes les nouvelles notions présentées avant. Pour résumer, cette extension définit une forme de concurrence en B, avec la possibilité d'exprimer des contraintes sur les traces d'exécution, tout en conservant les mécanismes de raffinement de B.

Associer un autre formalisme
Schneider et Treharne [Schneider and Treharne, 2002] utilisent l'algèbre de processus CSP pour décrire le contrôle d'exécution des opérations de machines B. Les opérations des machines B deviennent des événements pour les termes de CSP, et l'enchaînement des différents événements est décrit en utilisant la syntaxe de CSP. La sûreté d'un tel système est obtenue en vérifiant un *invariant de boucle de contrôle* : cette formule spécifie les conditions selon lesquelles le système ne peut pas se bloquer. Cet invariant modélise les schémas d'exécution selon lesquels les machines B appelées devront fonctionner, indéfiniment (d'où la dénomination *boucle de contrôle*). Plus précisément, la modélisation est séparée en trois étapes :

- Le comportement du modèle, et la façon dont il évolue au cours du temps. À cet effet, une machine introduisant une variable de temps ainsi que des opérations de consultation/avancement du temps est modélisée. Une machine peut donc "avancer le temps" pour indiquer ses propriétés temporelles. En effet, les opérations du modèle devraient être "soumises" au temps plutôt qu'être capables de le faire avancer. Les machines B qui décrivent le comportement du modèle font appel à cette "machine horloge" pour spécifier comment elles évoluent, temporellement parlant.
- Les contraintes du système, spécifiées via des machines non-temporisées.

- Un modèle CSP décrivant les interactions entre les différentes machines, où les passages de messages correspondent aux opérations des machines.

De cette manière, les différentes classes de spécification du système sont bien séparées :

- Ce que font les éléments du système (machines B à la base du fonctionnement du système)
- Les contraintes temporelles sur l'évolution du système (les machines B "temporisées")
- Le comportement global du système (ordonnancement de l'activation des différents éléments du système).

Valider un tel projet revient à valider d'abord les machines B, puis à générer les traces d'exécution depuis le modèle CSP et de vérifier la correction de ces traces d'exécution par rapport à des contraintes temporelles sur le système dans son ensemble. Les traces d'exécution correspondent aux séquences d'appels d'opérations B autorisées par le modèle CSP.

2.4.2 Extensions du B événementiel - Traitement des propriétés de vivacité

Les propriétés de *sûreté*, ou propriétés invariantes, décrivent les conditions en dehors desquelles le système ne peut pas fonctionner correctement. Les propriétés de *vivacité* ou dynamiques décrivent ce qui doit se passer de bien lorsque le système fonctionne. Il existe beaucoup de formalismes pour exprimer les propriétés dynamiques, en particulier les logiques temporelles, linéaires ou arborescentes. La preuve de ces propriétés sur des systèmes finis est réalisée à l'aide de la technique de model checking, c'est-à-dire en examinant tous les états du système. Des langages ou formalismes de spécification, tels que TLA [Lamport, 1991] ou UNITY [K-M. Chandy and J. Misra, 1989], permettent également d'exprimer des propriétés de sûreté et des propriétés dynamiques. Dans ces cas-là, des systèmes logiques ont été définis dans le but de prouver les propriétés par déduction.

Lorsque les systèmes sont décrits par des événements ou des actions de manière non déterministes, les comportements possibles peuvent être divers. Pour assurer les propriétés de vivacité, il faut alors, le plus souvent, imposer des hypothèses d'équité.

L'équité peut s'exprimer sous différentes formes (faible, forte, progrès minimal,etc). L'équité faible est définie de la manière suivante : si un événement (ou action) est continûment activable, alors il est infiniment souvent activé. L'équité forte est définie de la manière suivante : si un événement (ou une action) est infiniment souvent activé, alors il exécuté infiniment souvent. Dans le progrès minimal, la seule condition pour le déclenchement d'un événement est que sa garde soit vraie.

Dans le B événementiel, il n'y a pas de possibilité d'imposer des hypothèses d'équité. En revanche, les exemples montrent souvent la nécessité d'introduire un ordonnanceur abstrait dans les événements. Les conditions de raffinement des systèmes événementiels B assurent que les propriétés de vivacité du niveau supérieur sont préservées. Les formalismes tels que TLA et UNITY donnent la possibilité de tenir compte des hypothèses d'équité et définissent les obligations de preuve de propriétés de vivacité sous ces hypothèses.

Dans cette section, nous présentons les travaux qui portent sur l'extension de B pour traiter des propriétés de vivacité. Nous présentons en particulier les travaux de Abrial et Mussat, les travaux de Julliand et ceux de Bert.

*** Les travaux de Abrial et Mussat**

Jean Raymond Abrial et Louis Mussat ont proposé dans [Abrial, 1996b; Abrial and Mussat, 1998] d'introduire des propriétés dynamiques dans les systèmes d'événements B ainsi que les obligations de preuve associées, basées sur celles de la terminaison d'une boucle. Ils introduisent trois schémas de propriétés dynamiques : les invariants dynamiques et les modalités *leadsto* et *until*.

- L'invariant dynamique est défini par une formule de la forme : **Dynamics** $P(V, V')$.
  Où $P(V, V')$ désigne un prédicat avant-après et V' est l'ensemble des valeurs des variables obtenu après l'application d'un événement. Il exprime un invariant sur toute transition (événement) du système.
- La modalité *leadsto* est définie par une formule de la forme :

  Select p **leadsto** q **while** $e_1, ..., e_n$ **Invariant** J **Variant** V_a **End** .

  Où p, q, J sont des prédicats définies sur les variables V du système, les e_i sont des événements et V_a une formule arithmétique appelée variant. Ce variant V_a décrit la raison pour laquelle les événements du système conduisent inévitablement à un état qui satisfait un prédicat q. L'invariant de boucle J est une propriété complétant l'invariant du système qui est nécessaire à la démonstration de la terminaison des portions de chemins conduisant à un état où q est vérifié.
- La modalité *Until* est définie par une formule de la forme :

  Select p **Until** q **while** $e_1, ..., e_n$ **Invariant** J **Variant** V_a **End**

Dans [Bellegarde *et al.*, 2001], les auteurs indiquent que les contraintes temporelles B d'Abrial et Mussat (sans la clause while) peuvent être ré-exprimées en PLTL, parce que la logique PLTL s'interprète sur des systèmes de transition non étiquetés alors que les modalités B ont besoin des étiquettes. Soient p et q, deux formules propositionnelles. Un invariant dynamique pourra s'exprimer en PLTL par une formule de la forme $\Box(p \Rightarrow \bigcirc q)$. En ce qui concerne les modalités, elles s'expriment par les formules PLTL $\Box(p \Rightarrow \Diamond q)$, pour la modalité *leadsto*, et $\Box(p \Rightarrow pUq)$, pour la modalité *until*.

Vérification de l'invariant dynamique. L'obligation de preuve à vérifier pour assurer q'un invariant dynamique $\rho(X, X')$ est satisfait par le système est : $I \wedge prd_V(e_i) \Rightarrow \rho(V, V')$ pour tout événement e_i du système. Dans cette expression, $prd_V(e_i)$ désigne le prédicat avant-après associé à l'événement e_i, tel que défini dans le B book [Abrial, 1996a].

Vérification des contraintes dynamiques. Afin de vérifier les modalités exprimant des contraintes dynamiques, l'utilisateur doit exhiber, en plus de la propriété, un *invariant de boucle* et une fonction décroissante appelée *variant*.

L'utilisateur a également la possibilité de définir une liste d'événements intitulée $While$, facultative. Cette liste sert à restreindre la propriété aux exécutions qui n'appliquent que ces événements. Ainsi, la signature intuitive d'une modalité p *Leadsto* q incluant une liste While $e_1, ..., e_n$ est qu'un état où p est vérifié n'est obligé de mener à un état où q est vérifié que si les événements $e_1, ..., e_n$ sont appliqués. Dans le cas où l'on veut vérifier qu'un état où p est vérifié mène à un état où q est vérifié quels que soient les événements appliqués, on peut omettre la liste While, ce qui revient à inclure tous les événements du système dans cette liste.

*** Les travaux de Julliand et Bellegarde**

Ces travaux [Bellegarde *et al.*, 2000a; Mountassir *et al.*, 2000; Julliand *et al.*, 1999] se placent dans le cadre de la spécification de systèmes réactifs sous forme d'événements B. A une spécification B événementielle, ils intègrent l'expression de propriétés dynamiques décrites en termes de formules de logique temporelle linéaire propositionnelle (PLTL). La vérification de ces propriétés par une technique de preuve n'est pas automatisable, ainsi ils proposent d'effectuer leur vérification par model- checking.

Afin de faire face au problème de l'explosion combinatoire lié à l'utilisation du model-checking, ils proposent de découper par partition le graphe d'accessibilité issu de la spécification en un ensemble de modules de petite taille, et de mener la vérification sur chacun des modules de manière séparée. Cette méthode est appelée *Vérification Modulaire*. Ils veulent être en mesure, à partir de la vérification séparée d'une propriété sur chacun des modules, de décider si la propriété est vérifiée globalement. Toutes les propriétés ne se prêtent pas à une telle vérification car certaines d'entre elles peuvent être fausses globalement bien que vraies sur chacun des modules. Ils définissent alors les propriétés qui se prêtent à une telle vérification de la manière suivante (ces propriétés sont dites modulaires) : une propriété est modulaire si et seulement si le fait qu'elle est vraie sur chaque module implique qu'elle est vraie globalement.

Il faut noter également que tous les découpages n'auront pas la même efficacité relativement à la vérification des propriétés. En effet, la propriété a besoin d'être vraie sur chaque module pour pouvoir conclure. Or, un découpage aléatoire aurait de fortes chances de voir échouer la vérification d'une propriété dans un ou plusieurs modules, du fait que certains chemins qui rendent la propriété vraie pourraient être coupés. Le problème est donc de produire un découpage modulaire en accord avec les propriétés qu'ils cherchent à vérifier. En résumé, ils tentent de répondre aux deux questions suivantes :

- Quelles sont les propriétés vérifiables modulairement ?
- Comment produire un "bon" découpage modulaire ?

En réponse à la première question, ils prouvent formellement que les propriétés PLTL issues des 3 schémas de contraintes dynamiques introduits par J.-R. Abrial et L. Mussat sont vérifiables de manière modulaire. Ils prouvent également que toute une classe de propriétés PLTL, incluant les 3 schémas précités, est vérifiable de manière modulaire. Ils exhibent une condition suffisante de modularité d'une propriété. Cette condition de modularité repose sur l'analyse syntaxique de l'automate de Büchi qui code la négation d'une propriété PLTL. Elle est donc facilement automatisable.

Afin de répondre à la deuxième question, Ils proposent de guider la décomposition modulaire par le raffinement B, ce qui offre l'avantage de produire un découpage sémantique du graphe d'accessibilité à chaque niveau du raffinement. Ils font alors la distinction entre les nouvelles propriétés, introduites à un niveau donné de raffinement et les anciennes propriétés, établies aux niveaux précédents de raffinement et conservées par celui-ci. La vérification modulaire porte sur les nouvelles propriétés.

*** Les travaux de Bert**

Dans cette section, nous présentons une méthode de spécification et de vérification des propriétés dynamiques sous hypothèses d'équité en B, ce travail est présenté dans [Barradas and Bert, 2002]. Dans [Barradas and Bert, 2005], les auteurs proposent une méthode de spécification

et de vérification des propriétés de vivacité sous hypothèses d'équité dans les systèmes d'événements B. Ils se sont inspirés de la méthode UNITY afin de spécifier et prouver ce type de propriétés. L'équité prise en compte par ce travail est l'*équité faible* et le *progrès minimal*. Les propriétés de vivacité étudiées sont de la forme "P *LeadsTo* Q" qui signifie qu'à partir d'un état s vérifiant P le système va évoluer inévitablement vers un état s' où Q est vraie.

Selon Bert, les propriétés de vivacité sont divisées en propriétés de base et propriétés générales. Les propriétés de vivacité de base permettent de spécifier des transitions particulières effectuées par l'activation d'un événement, tandis que les propriétés de vivacité générales spécifient des transitions réalisées par l'exécution de plusieurs événements. Les auteurs ont donné des obligations de preuve fondées sur le calcul des plus faibles préconditions pour prouver les propriétés de base et les propriétés de vivacité générales sont prouvées par les définitions et théorèmes de la logique UNITY. Ils ont donné aussi, les obligations de preuve de la préservation des propriétés de vivacité par le raffinement.

- Propriétés de vivacité dans UNITY. UNITY est une méthode de développement de programmes parallèles. Elle comprend : une logique pour la spécification et la preuve des propriétés des programmes, un langage de programmation abstrait pour décrire le comportement dynamique des programmes et un ensemble de règles permettant de traduire ce langage abstrait vers une notation concrète, proche de l'architecture matérielle où les programmes seront mis en oeuvre.

La logique UNITY dispose de deux relations pour spécifier les propriétés de vivacité : la relation *ensures* et la relation *"Leads to"*, notée $\leadsto$. La formule p *ensures* q signifie que dans toute exécution σ d'un système, si p est satisfait sur un certain état s_i de σ, alors il sera satisfait sur tous les états s_j de σ tel que $j \geq i$ tant que q n'est pas satisfait, et inévitablement q sera satisfait dans σ. Dans la sémantique d'une propriété de $P \leadsto Q$ est similaire à celle de p *ensures* q, cependant en $P \leadsto Q$ nous ne pouvons pas assurer que p reste vrai tandis que q ne l'est pas.

La relation *ensures* dépend de certaines hypothèses d'équité : le "*progrès minimal*" MP et l'*équité faible* WF. Dans un système sous les hypothèses d'équité MP, si dans un état il y a un ou plusieurs événements activables, alors un événement est choisi pour être activé. Dans un système sous les hypothèses d'équité WF, tout événement continuellement activable sera infiniment souvent activé.

- Intégration des propriétés de vivacité de base dans B. Pour spécifier les propriétés de vivacité dans B, les auteurs proposent l'utilisation de deux nouvelles clauses dans les modèles B : FAIRNESS et MODALITIES. L'équité est définie dans la clause FAIRNESS par $Fairness_{MP}$ ou $Fairness_{WF}$ pour indiquer le *"progrès minimal"* MP et l'*équité faible* WF. Les propriétés à vérifier sont définies dans la clause MODALITIES.

Soit ES un système d'événements B. Soient I l'invariant de ES et E l'ensemble de ses événements. Les obligations de preuve associées à la propriété P *ensures* Q sous hypothèses d'équité *"progrès minimal"* MP sont :

- $MP_0 : I \wedge P \wedge \neg Q \Rightarrow [[]_{e \in E}\ e]Q$
- $MP_1 : I \wedge P \wedge \neg Q \Rightarrow grd([[]_{e \in E}\ e])$

$[]_{e \in E}\ e$ signifie le choix indéterministe d'un événement e dans l'ensemble des événements E.

- MP_0 indique que si un état satisfait $I \wedge P \wedge \neg Q$, alors n'importe quel événement activable à partir de cet état conduit vers un état satisfaisant Q.

- MP_1 indique que si un état satisfait $I \wedge P \wedge \neg Q$, alors il existe toujours un événement activable à partir de cet état (l'activation de cet événement est assurée par les hypothèses d'équité MP).

Les obligations de preuve associées à la propriété P *ensures* Q sous hypothèses d'*"équité faible"* WF sont :

- $WF_0 : I \wedge P \wedge \neg Q \Rightarrow [[]_{e \in E}\ e](P \vee Q)$
- $WF_1 : I \wedge P \wedge \neg Q \Rightarrow \exists e.(e \in E \wedge grd(e) \wedge [e]Q)$

- WF_0 indique que si un état satisfait $I \wedge P \wedge \neg Q$, alors n'importe quel événement activable à partir de cet état conduit vers un état satisfaisant $P \vee Q$.
- WF_1 indique que si un état satisfait $I \wedge P \wedge \neg Q$, alors il existe toujours un événement activable à partir de cet état qui conduit vers Q (l'activation de cet événement est assurée par les hypothèses d'équité WF).

Ces obligations de preuve sont correctes sous hypothèses d'équité (on suppose que l'équité est satisfaite par le système).

- Intégration des propriétés de vivacité générales dans B. Les propriétés de vivacité générales sont spécifiées par des formules de la forme $P \rightsquigarrow Q$ (P conduit à Q). Cette formule spécifie qu'un système atteint fatalement un état où Q est vrai lorsque son exécution arrive dans un état où P est vrai. Il y a trois différences fondamentales entre ce type de propriété et une propriété de vivacité de base :

- La transition d'états de P à Q peut se faire par l'activation d'un ou plusieurs événements,
- Une propriété générale ne garantit pas que le prédicat P est préservé tant que le système n'a pas atteint l'état où Q est vrai,
- La définition des propriétés générales ne dépend pas directement des hypothèses d'équité comme les propriétés de base.

Afin de prouver la propriété $P \rightsquigarrow Q$ sur un système d'événements B, on doit montrer que cette propriété peut être obtenue en appliquant un nombre fini des règles suivantes définies dans la méthode UNITY :

	ANTECEDENT	*CONSEQUENT*
BRL	P *ensures* Q	$P \rightsquigarrow Q$
TRA	$P \rightsquigarrow R,\ R \rightsquigarrow Q$	$P \rightsquigarrow Q$
DSJ	$\forall m.(m \in M \Rightarrow P_m \rightsquigarrow Q)$	$\exists m.(m \in M \Rightarrow P_m \rightsquigarrow Q)$

Dans la règle DSJ, M dénote n'importe quel ensemble dont les éléments servent à indexer une famille de prédicats P_m. En utilisant les mêmes définitions des propriétés de vivacité de la méthode UNITY, les auteurs peuvent tirer profit des théorèmes de cette méthode afin de raisonner sur les propriétés de vivacité dans le cadre du B événementiel.

Les spécifications des propriétés de vivacité dans un système B événementiel seront données par des relations $\rightsquigarrow$. Lorsque les auteurs veulent prouver que de telles propriétés sont vraies dans un système donné, ils montrent qu'il existe des propriétés de base qui sont vraies dans le système et ils appliquent les règles nécessaires pour montrer la propriété souhaitée. Les propriétés de base doivent être prouvées en utilisant les obligations de preuve WFO et WF1 s'ils supposent des hypothèses d'équité faible.

- Raffinement des propriétés de vivacité. Pour le raffinement des propriétés de vivacité sous hypothèses d'équité, les auteurs proposent deux types de raffinement : un raffinement qui ne préserve pas les propriétés de vivacité prouvées au niveau abstrait, et un raffinement qui les préserve. Cette observation est justifiée par le fait qu'au niveau raffiné, le besoin d'équité faible du niveau abstrait peut devenir un besoin d'équité forte du niveau raffiné.

L'approche de vérification de la préservation des propriétés de vivacité sous hypothèses d'équité par le raffinement, rentre dans le contexte où, un système A avec n événements a_i au niveau abstrait est raffiné en un système C avec n événements concrets c_i et m nouveaux événements h_j, tel que c_i raffine l'événement abstrait a_i.

Soit la propriété P *ensures* Q sous hypothèses d'équité MP. Les obligations de preuve MP_0 et MP_1 assurent que cette propriété est satisfaite par le système abstrait A. Afin de préserver la propriété dans le système concret, il faut que MP_0 et MP_1 soient satisfaites par C. Afin de faire cette preuve, il suffit de prouver que MP_0 est satisfaite par les nouveaux événements, les anciens la satisfont parce que leurs gardes sont renforcées au niveau raffiné.

Pour une propriété *ensures* sous l'équité faible WF, il faut vérifier que WF_0 est satisfaite par les nouveaux événements. Par contre, il faut vérifier que WF_1 est satisfaite par tous les événements de C.

Une propriété $P \rightsquigarrow Q$ du niveau abstrait, est dérivée d'une ou plusieurs propriétés de base (*ensures*). Afin de vérifier si elle est préservée au niveau raffiné, il suffit de vérifier que les propriétés de base qui la composent sont préservées.

Les points traités dans ces travaux sont :

- la spécification et la preuve des propriétés de vivacité sous hypothèses d'équité faible WF et le progrès minimal MP,
- l'intégration des règles de la logique UNITY dans B,
- l'expression de la propriété *ensures* par des obligations de preuve,
- la preuve de la relation *Leads To* par des règles d'inférence,

Les points non traités dans les travaux sont :

- la sémantique des propriétés de vivacité basée sur les traces d'exécution,
- l'étude de la preuve de propriétés dynamiques sous hypothèses d'équité forte

2.5 Conclusion

La méthode B présentée dans ce chapitre est définie comme une méthode de développement formelle mettant en valeur aussi bien les aspects statiques que dynamiques d'un système. D'un coté, les aspects statiques sont définis au moyen de concepts mathématiques de la théorie des ensembles et la logique des prédicats, et d'un autre coté, les aspects dynamiques sont modélisés en utilisant le langage des substitutions généralisées.

Nous pouvons retenir que toutes les extensions apportées à la méthode B ou le B événementiel sont des extensions utilisant les mécanismes de B ou des extensions avec d'autres sémantiques. Nous nous sommes intéressés surtout au traitement des propriétés de vivacité et au traitement du temps au niveau de la méthode B. Concernant le traitement des propriétés de vivacité, nous avons évoqué les travaux d'Abrial, de Julliand et de Bert. Bert a traité les propriétés de vivacité

en se basant sur les transformateurs de prédicats et pas sur les traces d'exécution. De plus, il n'a pas traité les propriétés de vivacité sous hypothèses d'équité forte. D'un autre coté, Julliand s'est intéressé surtout à la vérification des propriétés PLTL par model checking.

L'approche que nous proposons dans ce travail (partie contribution) permet de spécifier et de vérifier des propriétés de vivacité sous hypothèses d'équité faible et forte en utilisant le langage de modélisation TLA$^+$ et son model checker TLC pour la vérification des propriétés de vivacité sur des systèmes à états finis. De plus, nous utilisons les diagrammes de prédicats et l'outil Dixit associé pour la vérification des propriétés de vivacité sur des systèmes à états infinis.

Nous continuons, dans le chapitre suivant, notre partie présentant le contexte scientifique de la thèse et les préliminaires, en étudiant le langage de modélisation TLA$^+$ et les diagrammes de prédicats, ainsi que leurs outils associés.

Chapitre 3

Le langage de modélisation TLA$^+$ et les diagrammes de prédicats

3.1 Introduction

Dans ce chapitre, nous présentons dans un premier temps la logique TLA (Temporal Logic of Actions) introduite par Leslie Lamport, en nous appuyant sur [Lamport, 1991; 1994a], ainsi que le langage de modélisation TLA$^+$ [Lamport, 2002] qui étend TLA pour faciliter la structuration des modèles. Nous présentons par la suite les diagrammes de prédicats qui représentent les abstractions des systèmes à états infinis. Ces diagrammes permettent la vérification des propriétés de sûreté et de vivacité. Les obligations de preuve non temporelles établissent la correspondance entre les spécifications, alors que la technique de model checking peut être utilisée pour vérifier des propriétés de comportement. Nous présentons la notion de raffinement entre diagrammes qui justifie le développement *top-down* de systèmes. La méthode est illustrée par un nombre d'algorithmes d'exclusion mutuelle.

3.2 La logique TLA

Avant de présenter la logique TLA [Lamport, 1994b; Merz, 2003], nous commençons par introduire un exemple [Lamport, 1994b].

3.2.1 Exemple introductif

Considérons le programme suivant, écrit dans un pseudo-langage :

initially x=0 ;
loop forever x :=x+1 **end loop**

Ce programme peut être décrit par la formule suivante :
$\Pi \triangleq (x = 0) \wedge \Box[x' = x + 1]_x \wedge WF_x(x' = x + 1)$

Cette formule est composée de trois parties : la première exprime que la valeur initiale de x est 0, la seconde que la nouvelle valeur de x est toujours ou bien égale à l'ancienne plus un ou

bien identique à l'ancienne ; enfin, la dernière exprime que si l'incrémentation peut toujours se faire, elle se fera. Π est une représentation en TLA des comportements possibles du programme. Une formule TLA va décrire un ensemble d'exécutions en exprimant en particulier les transitions autorisées.

3.2.2 Présentation de TLA

Pour introduire la logique temporelle TLA, nous commençons par définir les concepts de variables, d'états et de traces.

* Variables et valeurs

Nous nous donnons une collection *Val* de valeurs. Nous ne précisons pas cette collection et il faut simplement qu'elle soit suffisamment grande pour contenir toutes les valeurs dont nous avons besoin en pratique. *Val* pourrait être une collection d'ensembles de la théorie des ensembles de Zermelo-Fraenkel (ZF) [Shoenfield, 1978]. Les entiers, les chaînes de caractères, etc., sont considérés comme des valeurs. Les booléens TRUE et FALSE sont considérés comme ne faisant pas partie de *Val*.

Nous nous donnons aussi un ensemble infini dénombrable *Var* de variables. La valeur de chaque variable sera interprétée comme un élément de *Val*.

Nous distinguons les *variables rigides* et les *variables flexibles*. Les *variables rigides* sont des variables qui peuvent avoir une valeur quelconque, mais qui ne doit pas changer entre un ancien et un nouvel état. Une variable rigide pourra donc avoir une valeur initiale quelconque mais devra la conserver. Une variable flexible peut changer de valeur entre un ancien et un nouvel état. Les variables rigides et flexibles peuvent être quantifiées. La quantification rigide est notée $\exists$ et la quantification flexible $\exists$. La sémantique de la quantification est donnée dans la figure 3.2.

* États et traces

Un état s représente une *valuation* des variables, c'est une fonction de **Var** vers **Val** ($s \in [\mathbf{Var} \rightarrow \mathbf{Val}]$). L'état s affecte une valeur notée $s[|x|]$ à la variable x. L'ensemble de tous les états possibles est noté **St**. La signification $[|x|]$ est une fonction de l'ensemble des états vers l'ensemble des valeurs. Une trace est une suite infinie d'états ; on notera $s_0 s_1 ...$ la suite des états $s_0, s_1,$ etc. La sémantique d'une spécification TLA est basée sur le comportement des variables à travers une séquence d'états.

* Fonctions d'état et prédicats

Une *fonction d'état* est une expression non booléenne construite à partir de variables et de symboles constants. La signification $[|f|]$ d'une fonction d'état f est une fonction de **St** vers **Val**. Par exemple, $[|x^2 + y - 3|]$ est une fonction associant à l'état s la valeur $(s[|x|])^2 + s[|y|] - 3$.

$s[|f|]$ est défini sémantiquement comme suit :
$s[|f|] \triangleq f(\forall\ 'v' : s[|v|]/v)$

où f($\forall$ $'v'$: s[|v|]/v) représente la valeur obtenue à partir de f en substituant s[|v|] à v, pour toutes les variables v. Une variable est un cas particulier de fonction d'état.

Un *prédicat* (ou un prédicat d'état) est une expression booléenne construite à partir de variables et de symboles de constantes. La signification [|P|] d'un prédicat P est une fonction des états vers les booléens, c'est-à-dire que pour chaque état s, s[|P|] est un booléen. s satisfait P si et seulement si s[|P|] s'évalue à vrai.

*** Action**

Une *action* est une expression à valeur booléenne pouvant être formée de variables, de variables primées et de symboles de constantes. Un exemple d'action est $x' + y = 1$.

Une action représente une relation entre deux états : un ancien état et un nouvel état. Les variables non primées font référence à l'ancien état et les variables primées font référence au nouvel état. Par exemple, $x' + y = z$ signifie que la somme de la valeur de x dans le nouvel état et de la valeur de y dans l'ancien état est égale à la valeur de z dans l'ancien état.

Formellement, la signification de [|A|] d'une action A est une relation entre deux états, c'est-à-dire une fonction qui affecte un booléen s[|A|]t à une paire d'états (s, t). s[|A|]t est obtenu à partir de A en remplaçant chaque variable non primée v par s[|v|] et chaque variable primée v' par t[|v|] :

$$s[|A|]t \triangleq f(\forall' v' : s[|v|]/v, t[|v|]/v')$$

Par exemple, $s[|x' + y' = x + y|]t$ est égal à $t[|x|] + t[|y|] = s[|x|] + s[|y|]$

Prédicat Enabled. Pour une action A, *Enabled* A est le prédicat qui est vrai pour un état ssi il est possible de réaliser une étape à partir de cet état. Sémantiquement, *Enabled* A est défini par : $S[|\text{Enabled } A|] \equiv \exists t \in St : s[|A|]t$

Opérations sur les actions. Nous pouvons définir plusieurs opérations intéressantes sur les actions TLA. :

- composition séquentielle des actions $\bullet$:
  La composition séquentielle des actions est en fait la composition des relations. Elle revient à faire se succéder deux actions et à dissimuler l'état intermédiaire par une quantification.
  Exemples de composition séquentielle :
  $(x' = x + 1) \bullet (x' = x + 1) \triangleq \exists \$x : (\$x = x + 1) \wedge (x' = \$x + 1)$, ce que l'on peut récrire
  $(x' = x + 1) \bullet (x' = x + 1) \triangleq (x' = x + 2) \wedge (\exists \$x : \$x = x + 1)$.
- composition parallèle $\wedge$: il s'agit simplement de la conjonction logique $\wedge$;
  Exemple de composition parallèle : $(x' = x + 1) \wedge (y' = y)$
- composition alternative $\vee$: il s'agit simplement de la disjonction logique $\vee$, avec $A \vee B \triangleq \neg(\neg A \wedge \neg B)$.
  Exemple : $((x' = x + 1) \wedge (y' = y)) \vee ((y' = y + 1) \wedge (x' = x))$
- Enabled : pour toute action A, Enabled(A) est un prédicat sur l'ensemble des états.

*** La logique temporelle TLA**

La logique temporelle de TLA est fondée sur des traces, c'est-à-dire des suites d'états. Une trace représente un comportement ou une exécution. Un programme est représenté par l'ensemble

de ses exécutions possibles.

Comme chaque état a un seul état successeur, TLA est une *logique temporelle linéaire discrète*. Nous allons tout d'abord définir la notion de formule temporelle, puis la sémantique d'une telle formule.

Une formule temporelle est construite à partir de formules élémentaires en utilisant des opérateurs booléens et des opérateurs temporels. Les formules élémentaires de TLA sont les prédicats et les formules de la forme $\Box[A]_f$, où $[A]_f \triangleq A \vee (f' = f)$.

Des exemples de formules temporelles sont : $\neg E_1 \wedge \Box(\neg E_2)$ et $\Box(E_1 \Rightarrow \Box(E_1 \vee E_2))$ avec (E_1 et E_2 sont formules élémentaires).

$\sigma[|F|]$: dénote la valeur booléenne que la formule F assigne au comportement σ et σ satisfait F ssi $\sigma[|F|]$ s'évalue à true. Les autres formules se définissent de la manière suivante :

$\sigma[|F \wedge G|] \triangleq \sigma[|F|] \wedge \sigma[|G|]$

$\sigma[|\neg F|] \triangleq \neg\sigma[|F|]$

$[|\Box F|]$ est défini en termes de $[|F|]$. Soit $s_0\ s_1\ s_2$,... une trace dont le premier état est s_0, le second s_1, etc., alors $s_0\ s_1\ s_2$,... $[|\Box F|] \triangleq \forall n \in Nat : s_0\ s_1\ s_2$,... $[|F|]$

Pour chaque formule temporelle F, $\Diamond F$ est définie comme suit : $\Diamond F \triangleq \neg\Box\neg F$.

Cette formule asserte que ce n'est pas toujours le cas que F est toujours fausse. $\Diamond F$ asserte que F est inévitablement vraie. Nous avons :

$s_0\ s_1\ s_2$,... $[|\Diamond F|] \triangleq \exists n \in Nat : s_n\ s_{n+1}\ s_{n+2}$,... $[|F|]$.

La figure 3.1 présente "simple TLA" qui est la logique TLA sans la quantification.

*** Propriétés**

En TLA, spécification et propriété sont des formules TLA. Les propriétés d'un système qui nous intéressent le plus souvent sont les propriétés d'invariance, d'équité et de fatalité. L'invariance est une propriété de sûreté, l'équité et la fatalité sont des propriétés de vivacité.

Sûreté. Les formules TLA permettent d'exprimer les propriétés de sûreté.
Vivacité. Les formules TLA permettent d'exprimer que les comportements infinis doivent satisfaire certaines propriétés. On exprime la vivacité dans TLA en fonction de la *fatalité* et *l'équité (fairness)*. On distingue *l'équité faible* et *l'équité forte*.

Les *équités faible* et *forte* sont définies respectivement par les formules suivantes :
$WF_f(A) \triangleq \Diamond\Box$Enabled $\langle A\rangle_f \Rightarrow \Box\Diamond\ \langle A\rangle_f$ et
$SF_f(A) \triangleq \Box\Diamond$Enabled $\langle A\rangle_f \Rightarrow \Box\Diamond\ \langle A\rangle_f$

*** Spécification TLA d'un système**

En exprimant la spécification sous forme d'une formule TLA, un système est spécifié par la formule correspondant à un ensemble de comportements autorisés. Une spécification en TLA a la forme suivante :

$\bigwedge$	Init
$\bigwedge$	$\Box[\text{Next}]_x$
$\bigwedge\bigwedge$	$SF_x(A)$ avec $A \in A_{SF}$
$\bigwedge\bigwedge$	$WF_x(A)$ avec $A \in A_{WF}$

Syntaxe

$\langle$*Formule*$\rangle$ $\triangleq$ $\langle$prédicat$\rangle$ | $\Box$ [$\langle$action$\rangle$]$\langle$*fonction d'état*$\rangle$ | $\neg$ $\langle$*Formule*$\rangle$
| $\langle$*Formule*$\rangle$ $\wedge$ $\langle$*Formule*$\rangle$ | $\Box$ $\langle$*Formule*$\rangle$

$\langle$*Action*$\rangle$ $\triangleq$ expression à valeur booléenne contenant des symboles constants, des variables et des variables primées

$\langle$*Prédicat*$\rangle$ $\triangleq$ $\langle$*Action*$\rangle$ sans variables primées | Enabled $\langle$*Action*$\rangle$

$\langle$*fonction d'état*$\rangle$ $\triangleq$ expression non booléenne contenant des symboles constants et des variables

Sémantique

$s[|f|] \triangleq f(\forall' v' : s[|v|]/v)$

$s[|A|]t \triangleq A(\forall' v' : s[|v|]/v, t[|v|]/v')$

$\models A \triangleq \forall s, t \in St : s[|A|]t$

$s[|\text{Enabled } A|] \triangleq \exists t \in St : s[|A|]t$

$\sigma[|F \wedge G|] \triangleq \sigma[|F|] \wedge \sigma[|G|]$

$\sigma[|\neg F|] \triangleq \neg\sigma[|F|]$

$\models F \triangleq \forall \sigma \in St\infty : \sigma[|F|]$

$\langle s_0 , s_1 ,...\rangle [|\Box F|] \triangleq \forall n \in Nat : \langle s_n, s_{n+1}, ...\rangle [|F|]$

$\langle s_0, s_1, ...\rangle [|\Diamond F|] \triangleq \exists n \in Nat : \langle s_n, s_{n+1}, ...\rangle [|F|]$

$\langle s_0, s_1, ...\rangle [|A|] \triangleq s_0[|A|]s_1$

$\langle s_0, s_1, ...\rangle [|\Box A|] \triangleq \forall n \in Nat : \langle s_n, s_{n+1}, ...\rangle [|A|] \triangleq \forall n \in Nat : s_n[|A|]s_{n+1}$

$\langle s_0, s_1, ...\rangle [|P|] \triangleq s_0[|P|]$

$\langle s_0, s_1, ...\rangle [|\Box P|] \triangleq \forall n \in Nat : s_n[|P|]$

Opérateurs sur les prédicats et les actions

$p' \triangleq p(\forall' v' : v'/v)$

$[A]_f \triangleq A \vee (f' = f)$

$\langle A_f\rangle \triangleq A \wedge (f' \neq f)$

Unchanged $f \triangleq f' = f$

Enabled A $\triangleq$ un pas A est possible

Opérateurs temporels

$\Box F \triangleq$ (F est toujours vrai)

$\Diamond F \triangleq \neg\Box\neg F$ (Fatalement F)

$F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$ (F conduit à G)

$WF(A) \triangleq \Box\Diamond \langle A\rangle \vee \Box\Diamond\neg$Enabled $\langle A\rangle$ (Équité faible)

$SF(A) \triangleq \Box\Diamond \langle A\rangle \vee \Diamond\Box\neg$Enabled $\langle A\rangle$ (Équité forte)

$\exists x : F \triangleq$ Quantification existentielle temporelle

$\forall x : F \triangleq$ Quantification universelle temporelle

Où :

f est une fonction d'état,
A est une action,
F et G sont des formules,
P est une fonction d'état ou un prédicat,
$s, s0, s1, ...$ sont des états,
σ est un comportement (séquence d'état),
St ensemble de tous les états,
$(\forall' v' : ...../v, .../v')$ dénote la substitution pour toutes les variables v,
St est l'ensemble de tous les états possibles.
St^∞ représente l'ensemble des traces.

FIG. 3.1 – Simple TLA (Syntaxe et sémantique de TLA sans quantificateur (d'après Lamport)).

Où Init spécifie les états initiaux des traces d'exécution, $\Box[\text{Next}]_x$ décrit les traces possibles d'exécution, A_{SF} spécifie les actions exécutées sous hypothèse d'équité forte pour les actions de Next concernées et A_{WF} spécifie les actions exécutées sous hypothèse d'équité faible pour les actions de Next concernées.

*** Raffinement**

Dans TLA, la spécification du système est décrite par une formule TLA caractérisant l'ensemble des traces pour lesquelles elle est vrai. Dans TLA, l'implication tient compte du bégaiement, c.à.d. qu'une spécification est satisfaite par un comportement σ si elle est satisfaite par tout comportement obtenu à partir de σ en ajoutant ou en enlevant des étapes qui laissent l'état inchangé. Le raffinement s'exprime alors directement en termes d'implication. Il s'agit d'un bégaiement (*stuttering*).

Le bégaiement désigne une suite d'états où certaines variables conservent la même valeur. Il apparaît naturellement si l'on cache certaines variables et il joue par conséquent un rôle essentiel dans le raffinement. Deux traces qui ne diffèrent que par le bégaiement satisfont les mêmes formules TLA$^+$. Le raffinement s'identifie à l'implication logique.

L'outil TLC (TLA$^+$ Model checker) [Lamport and Yu, 2003] peut être utilisé pour vérifier des spécifications écrites en TLA$^+$ à différents niveaux de raffinements .

La figure 3.2 présente la sémantique de la quantification et du bégaiement (*stuttering*) dans TLA.

*** Preuves TLA**

TLA offre un ensemble de règles de preuve pour démontrer des propriétés de raffinement ou la bonne formation des spécifications. Elle est munie d'un système de preuve. Les preuves en TLA, ou la vérification de preuves TLA, ont donné lieu à plusieurs travaux, à chaque fois en liaison avec un prouveur de théorèmes particulier. Les preuves TLA sont organisées hiérarchiquement selon un format défini dans [Lamport, 1994a]. Chaque pas à prouver est décomposé en pas intermédiaires jusqu'à être justifié soit par l'emploi d'une règle du système de preuve TLA (figure 3.3), soit par l'emploi d'un raisonnement élémentaire en logique.

3.3 Le langage de modélisation TLA$^+$

TLA$^+$ est un langage de spécification due à Lamport [Lamport, 2002] étendant la logique temporelle des actions TLA à l'aide de la structure de module et la théorie des ensembles.

3.3.1 Exemple

Le module de la figure 3.4 introduit brièvement TLA$^+$. La structure du module encapsule diverses définitions classifiées selon leur "type temporel" et leur valeur (booléenne ou non). Les paramètres (x, y) peuvent être instanciés par d'autres modules. La première définition, *Init*, est

Syntaxe

$\langle$*formule générale*$\rangle$ $\triangleq$ $\langle$*formule*$\rangle$ | $\exists$ $\langle$*variable*$\rangle$: $\langle$*formule générale*$\rangle$
| $\exists$ $\langle$variable rigide$\rangle$: $\langle$*formule générale*$\rangle$
| $\neg$ $\langle$*formule générale*$\rangle$
| $\langle$*formule générale*$\rangle$ $\wedge$ $\langle$*formule générale*$\rangle$

$\langle$*formule*$\rangle$ $\triangleq$ une formule TLA simple

Sémantique

$\langle s_0, s_1, ...\rangle =_x \langle t_0, t_1, ...\rangle \triangleq \forall n \in \text{Nat} : \forall' v' \neq' x' : s_n[|v|] = t_n[|v|]$

$\natural \langle s_0, s_1, ...\rangle \triangleq$ if $\forall n \in \text{Nat} : s_n = s_0$
then $\langle s_0, s_0, ...\rangle$
else if $s_1 = s_0$ then $\natural \langle s_1, s_2, ...\rangle$
else $\langle s_0\rangle \circ \natural \langle s_1, s_2, ...\rangle$

$\sigma[|\exists x : F|] \triangleq \exists \rho, \tau \in \text{St}^\infty : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \tau[|F|]$

$\sigma[|\forall x : F|] \triangleq \sigma[|\neg\exists x : \neg F|]$

$\sigma[|\exists c : F|] \triangleq \exists c \in \text{Val} : \sigma[|F|]$

Règles de preuve

$E_1.\ \models F(f/x) \Rightarrow \exists x : F$

$F_1.\ \models F(e/c) \Rightarrow \exists c : F$

$$\frac{E_2.F \Rightarrow G \quad \text{x n'apparat pas libre dans G}}{(\exists\ x{:}F) \Rightarrow G}$$

$$\frac{F_2.F \Rightarrow G \quad \text{c n'apparat pas libre dans G}}{(\exists c{:}F) \Rightarrow G}$$

où
- x est une $\langle$*variable*$\rangle$
- f est une $\langle$*fonction d'état*$\rangle$
- c est une $\langle$*variable rigide*$\rangle$
- e est une $\langle$*expression constante*$\rangle$
- F est une $\langle$*formule générale*$\rangle$
- G est une $\langle$*formule générale*$\rangle$
- s, s_0, t_0, s_1, t_1 ... sont des états
- σ est un $\langle$*comportement*$\rangle$
- $\circ$ dénote la concaténation de suites

FIG. 3.2 – Syntaxe et sémantique de TLA avec quantificateur (d'après Lamport).

Règles de la logique temporelle simple

STL1. $\dfrac{\text{F prouvable par la logique propositionnelle}}{\Box \mathrm{F}}$ STL4. $\dfrac{\mathrm{F} \Rightarrow \mathrm{G}}{\Box \mathrm{F} \Rightarrow \Box \mathrm{G}}$

STL2. $\vdash \Box \mathrm{F} \Rightarrow \mathrm{F}$ STL5. $\vdash \Box(\mathrm{F} \wedge \mathrm{G}) \equiv (\Box \mathrm{F}) \wedge (\Box \mathrm{G})$

STL3. $\vdash \Box\Box \mathrm{F} \equiv \mathrm{F}$ STL6. $\vdash (\Diamond\Box \mathrm{F}) \wedge (\Diamond\Box \mathrm{G}) \equiv \Diamond\Box(\mathrm{F} \wedge \mathrm{G})$

LATTICE. $\succ$ *un ordre partiel bien fondé sur un ensemble S* $(\mathrm{S}, \prec)$

$$\frac{\mathrm{F} \wedge (\mathrm{c} \in \mathrm{S}) \Rightarrow (\mathrm{H}_\mathrm{c} \rightsquigarrow (\mathrm{G} \vee \exists\, \mathrm{d} \in \mathrm{S} : (\mathrm{c} \succ \mathrm{d}) \wedge \mathrm{H}_\mathrm{d}))}{\mathrm{F} \Rightarrow ((\exists \mathrm{c} \in \mathrm{S} : \mathrm{H}_\mathrm{c}) \rightsquigarrow \mathrm{G})}$$

Règles de base de TLA

TLA1. $\dfrac{\mathrm{P} \wedge (\mathrm{f}' = \mathrm{f}) \Rightarrow \mathrm{P}'}{\Box \mathrm{P} \equiv \mathrm{P} \wedge [\mathrm{P} \Rightarrow \mathrm{P}']_\mathrm{f}}$ TLA2. $\dfrac{\mathrm{P} \wedge [\mathrm{A}]_\mathrm{f} \Rightarrow \mathrm{Q} \wedge [\mathrm{B}]_\mathrm{g}}{\Box \mathrm{P} \wedge \Box[\mathrm{A}]_\mathrm{f} \Rightarrow \Box \mathrm{Q} \wedge \Box[\mathrm{B}]_\mathrm{g}}$

Règles supplémentaires

INV1. $\dfrac{\mathrm{I} \wedge [\mathrm{N}]_\mathrm{f} \Rightarrow \mathrm{I}'}{\mathrm{I} \wedge \Box[\mathrm{N}]_\mathrm{f} \Rightarrow \Box \mathrm{I}}$ INV2. $\vdash \Box \mathrm{I} \Rightarrow (\Box[\mathrm{N}]_\mathrm{f} \equiv \Box[\mathrm{N} \wedge \mathrm{I} \wedge \mathrm{I}']_\mathrm{f})$

WF1. $\dfrac{\begin{array}{l}\mathrm{P} \wedge [\mathrm{N}]_\mathrm{f} \Rightarrow (\mathrm{P}' \vee \mathrm{Q}') \\ \mathrm{P} \wedge \langle \mathrm{N} \wedge \mathrm{A} \rangle_\mathrm{f} \Rightarrow \mathrm{Q}' \\ \mathrm{P} \Rightarrow \mathrm{Enabled}\langle \mathrm{A} \rangle_\mathrm{f}\end{array}}{\Box[\mathrm{N}]_\mathrm{f} \wedge \mathrm{WF}_\mathrm{fA} \Rightarrow (\mathrm{P} \rightsquigarrow \mathrm{Q})}$ WF2. $\dfrac{\begin{array}{l}\langle \mathrm{N} \wedge \mathrm{B} \rangle_\mathrm{f} \Rightarrow \langle \mathrm{M} \rangle_\mathrm{g} \\ \mathrm{P} \wedge \mathrm{P}' \wedge \langle \mathrm{N} \wedge \mathrm{A} \rangle_\mathrm{f} \wedge \overline{\mathrm{Enabled}}\langle \mathrm{M} \rangle_\mathrm{g} \Rightarrow \mathrm{B} \\ \mathrm{P} \wedge \overline{\mathrm{Enabled}}\langle \mathrm{M} \rangle_\mathrm{g} \Rightarrow \mathrm{Enabled}\langle \mathrm{A} \rangle_\mathrm{f} \\ \Box[\mathrm{N} \wedge \neg \mathrm{B}]_\mathrm{f} \wedge \mathrm{WF}_\mathrm{fA} \wedge \Box \mathrm{F} \\ \wedge \Diamond\Box\overline{\mathrm{E}}\mathrm{nabled}\langle \mathrm{M} \rangle_\mathrm{g} \Rightarrow \Diamond\Box \mathrm{P}\end{array}}{\Box[\mathrm{N}]_\mathrm{f} \wedge \mathrm{WF}_\mathrm{fA} \wedge \Box \mathrm{F} \Rightarrow \overline{\mathrm{WF}_\mathrm{gM}}}$

SF1. $\dfrac{\begin{array}{l}\mathrm{P} \wedge [\mathrm{N}]_\mathrm{f} \Rightarrow (\mathrm{P}' \vee \mathrm{Q}') \\ \mathrm{P} \wedge \langle \mathrm{N} \wedge \mathrm{A} \rangle_\mathrm{f} \Rightarrow \mathrm{Q}' \\ \Box \mathrm{P} \wedge \Box[\mathrm{N}]_\mathrm{f} \wedge \Box \mathrm{F} \Rightarrow \Diamond \mathrm{Enabled}\langle \mathrm{A} \rangle_\mathrm{f}\end{array}}{\Box[\mathrm{N}]_\mathrm{f} \wedge \mathrm{SF}_\mathrm{fA} \wedge \Box \mathrm{F} \Rightarrow (\mathrm{P} \rightsquigarrow \mathrm{Q})}$ SF2. $\dfrac{\begin{array}{l}\langle \mathrm{N} \wedge \mathrm{B} \rangle_\mathrm{f} \Rightarrow \langle \mathrm{M} \rangle_\mathrm{g} \\ \mathrm{P} \wedge \mathrm{P}' \wedge \langle \mathrm{N} \wedge \mathrm{A} \rangle_\mathrm{f} \Rightarrow \mathrm{B} \\ \mathrm{P} \wedge \overline{\mathrm{Enabled}}\langle \mathrm{M} \rangle_\mathrm{g} \Rightarrow \mathrm{Enabled}\langle \mathrm{A} \rangle_\mathrm{f} \\ \Box[\mathrm{N} \wedge \neg \mathrm{B}]_\mathrm{f} \wedge \mathrm{SF}_\mathrm{fA} \wedge \Box \mathrm{F} \\ \wedge \Box\Diamond\overline{\mathrm{Enabled}}\langle \mathrm{M} \rangle_\mathrm{g} \Rightarrow \Diamond\Box \mathrm{P}\end{array}}{\Box[\mathrm{N}]_\mathrm{f} \wedge \mathrm{SF}_\mathrm{fA} \wedge \Box \mathrm{F} \Rightarrow \overline{\mathrm{SF}_\mathrm{gM}}}$

Où :

$\mathrm{F}, \mathrm{G}, \mathrm{H}_\mathrm{c}, \mathrm{H}_\mathrm{d}$ sont des formules TLA — $\mathrm{A}, \mathrm{B}, \mathrm{N}, \mathrm{M}$ sont des actions

$\mathrm{P}, \mathrm{Q}, \mathrm{I}$ sont des prédicats — f, g sont des fonctions d'état

FIG. 3.3 – Règles de preuve de TLA (extrait de Lamport).

un *prédicat* (**predicate**). Il est égal à true ou false dans un état donné. La seconde définition, A, est une *action* (**action**). Elle contient une référence à une variable primée x′. Cela signifie une référence au prochain état du calcul. L'action A est vraie dans un état donné, si x > y dans cet état et si la valeur de x dans l'état suivant est la valeur de x courante à laquelle nous soustrayons la valeur courante de y. La formule temporelle (**temporal**) est une formule vraie pour certaines traces. Nous pouvons voir cette formule comme quelque chose qui sélectionne certaines traces parmi un ensemble plus grand de traces. Les traces qui satisfont *Spec* sont par définition les bonnes traces vis-à-vis de la spécification. Outre ces catégories, il y a aussi des fonctions d'état, des fonctions de transition, des théorèmes, etc., comme nous l'avons vu dans les sections précédentes.

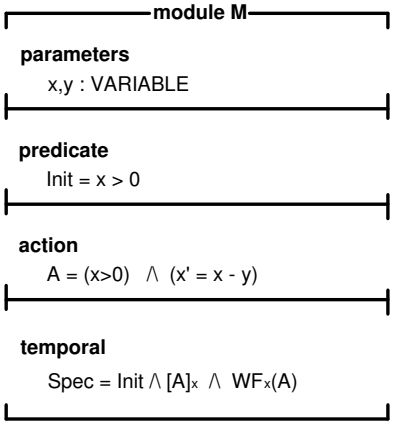


FIG. 3.4 – Exemple TLA$^+$

3.3.2 Syntaxe

Nous présentons par la suite de manière plus détaillée la syntaxe de TLA$^+$. Nous exposons tout d'abord la structure de module, puis la représentation des formules.

* Structure de module

Un module représente la plus petite unité de structure complète dans TLA$^+$. Cette structure est constituée des parties suivantes :

- une déclaration de paramètres ;
- des définitions d'opérateurs, dont une partie est exportée (visible depuis un module qui inclurait ce module) ;
- des hypothèses ;
- des théorèmes

Nous n'allons pas définir exhaustivement toute la structure d'un module, mais seulement les aspects les plus utiles.

Paramètres. Le mot-clé **parameter** (ou **parameters** s'il y en a plusieurs) donne une liste de paramètres. Il y a trois sortes de paramètres : les paramètres booléens (BOOLEAN), les constantes (CONSTANT) et les variables (VARIABLE). Les booléens représentent des formules constantes. Les constantes sont les variables rigides de la logique temporelle. Les variables sont les variables flexibles de la logique temporelle.

Définitions. Le mot-clé **definition** (ou **definitions**) indique des déclarations d'opérateurs. Un opérateur est une fonction associant des expressions primitives (c'est-à-dire employant des opérateurs déjà définis ou des primitives TLA$^+$) à des expressions. Le mot-clé **definition** peut être remplacé par des indications plus précises, telles qu'**action**. Aucun identificateur ne peut être utilisé comme une variable liée s'il a déjà une signification dans le module. Ceci interdit en particulier la quantification d'une variable dans une expression où celle-ci est déjà quantifiée. Les définitions ne peuvent donc pas être récursives.

Hypothèses et théorèmes. Les hypothèses (**assumptions**) et les théorèmes (**theorems**) sont définis comme les opérateurs mais sans arguments. Un module est dit valide si les hypothèses impliquent les théorèmes. Les hypothèses doivent être des formules constantes et ne peuvent donc pas contenir des primitives non constantes telles que $\Box, \exists, '$ ou *Enabled*. Enfin, les hypothèses et les théorèmes ne peuvent pas être utilisés ailleurs dans un module.

Import et Include. Une clause **import** importe toutes les déclarations de paramètres, les définitions exportées, les hypothèses et les théorèmes du module importé. Une déclaration **import** est utilisée pour scinder une spécification en plusieurs composantes. Une clause **include** importe uniquement des définitions.
Export. La clause **export** définit la visibilité des opérateurs définis dans un module. Si aucune clause **export** n'est présente, seules les déclarations d'opérateurs figurant dans le module seront exportées. Les définitions de modules inclus ou importés ne seront pas exportées.

*** Représentation de formules**

Pour accroître la lisibilité des formules de grande taille, Lamport a proposé un format de représentation des formules dans [Lamport, 1994b]. L'un des aspects de ce format est l'emploi des connecteurs $\wedge$ et $\vee$ sous forme préfixée. Plus précisément, dans TLA$^+$, il existe deux sortes de listes d'items : les listes conjonctives et disjonctives. Nous avons choisi d'ajouter une nouvelle sorte de liste, la liste compositionnelle. Ainsi,

$\wedge$A $\quad$ $\vee$A $\quad$ $\bullet$A
$\wedge$B $\quad$ $\vee$B $\quad$ $\bullet$B
$\wedge$C $\quad$ $\vee$C $\quad$ $\bullet$C

sont les équivalents préfixés de $A \wedge B \wedge C$ et $A \vee B \vee C$ et $A \bullet B \bullet C$. '$\bullet$' est la composition des relations et $A \bullet B \bullet C$ doit être compris comme l'"exécution" de A en premier, puis de B, enfin de C, le tout en un pas.

3.3.3 Composition dans TLA$^+$

La composition en TLA$^+$ correspond à la conjonction. On exprime celle-ci dans un nouveau module en chargeant les spécifications précédentes avec *Include*. Considérons par exemple les

deux modules M_1 et M_2 de la figure suivante :

MODULE M_1

Parameters
x, y **: VARIABLE**
Predicate
Init1 $\triangleq x > 0$
Action
A $\triangleq (x > y) \wedge (x' = x - y)$
Temporal
Spec1 $\triangleq$ Init1 $\wedge \Box[A]_{<x>} \wedge WF_{<x>}(A)$

MODULE M_2

Parameters
x, y **: VARIABLE**
Predicate
Init2 $\triangleq y > 0$
Action
B $\triangleq (x < y) \wedge (y' = y - x)$
Temporal
Spec2 $\triangleq$ Init2 $\wedge \Box[B]_{<y>} \wedge WF_{<y>}(B)$

La composition de ces deux modules est leur conjonction et peut se faire comme suit :

MODULE M_1_and_M_2

Parameters
x, y **: VARIABLE**
Include M_1
Include M_2
Temporal
Spec $\triangleq M_1$.Spec1 $\wedge M_2$.Spec2

Théorème de composition d'Abadi et Lamport
On distingue classiquement deux approches dans la construction modulaire des systèmes : l'approche *top-down* et l'approche *bottom-up*. Ces approches correspondent à deux formes différentes de raisonnement [Abadi and Lamport, 1995].

Approche *top-down*. Dans l'approche *top-down*, on opère par décomposition de spécifications TLA. Le contexte d'un sous-composant étant connu, on peut donc raisonner sur un sous-composant en incluant les propriétés nécessaires de son environnement. Cette approche peut être illustrée dans le cadre de la preuve de programme. Par exemple, la preuve de l'assertion de *true* $\{x := 2; x := x-1\}x \geq 0$ se décompose en la preuve des deux assertions $x-1 \geq 0\{x := x-1\}x \geq 0$ et *true* $\{x := 2\}x - 1 \geq 0$. La décomposition est introduite par l'instruction de séquencement et la seconde assertion a été calculée en fonction du contexte (la postcondition attendue est la précondition de la première assertion). Le principe de la preuve par décomposition est illustré par le théorème de Abadi et Lamport [Abadi and Lamport, 1995]. Ce théorème permet d'effectuer des preuves sur des spécifications composées à partir de preuves sur les composantes de spécifications. Le raffinement s'exprime à l'aide de l'implication logique et la composition s'exprime par la conjonction des spécifications. Schématiquement, M_1 et M_2 sont deux formules TLA décrivant deux systèmes, R_1 et R_2 sont leur raffinement et E_1 et E_2 sont les propriétés de contexte respectives de M_1 et M_2 (propriétés de composition). Le théorème de décomposition permettant de prouver les raffinements en utilisant des hypothèses sur la composition est le suivant :

- Composants indépendants

$$\frac{\begin{array}{c} R_1 \Rightarrow M_1 \\ R_2 \Rightarrow M_2 \end{array}}{R_1 \wedge R_2 \Rightarrow M_1 \wedge M_2}$$

- Composants dépendants

$$\frac{\begin{array}{c} M_1 \wedge M_2 \Rightarrow E_1 \wedge E_2 \\ R_1 \wedge E_1 \Rightarrow M_1 \\ R_2 \wedge E_2 \Rightarrow M_2 \end{array}}{R_1 \wedge R_2 \Rightarrow M_1 \wedge M_2}$$

Par exemple, pour construire un contrôleur on utilise généralement des propriétés sur le fonctionnement de l'environnement. Le théorème de décomposition permet de prouver les raffinements en utilisant des hypothèses sur la composition.

Approche *bottom-up*. Dans l'approche *bottom-up*, les composants sont spécifiés indépendamment et ils peuvent être réutilisables dans différents contextes. Il est donc possible de raisonner sur ces composants, hors de leur contexte d'utilisation. Un composant réutilisable peut néanmoins supposer certaines propriétés sur le contexte dans lequel il sera utilisé (son environnement). Il est alors spécifié sous la forme de spécifications *rely/guarantee* [Jones, l981]. La condition *rely* décrit les hypothèses de comportement de l'environnement et la condition *guarantee* est le contrat respecté par le comportement du composant (composant ouvert). Les opérateurs de composition permettent d'assembler de tels composants sous certaines conditions. Lorsque la composition est possible, le principe de compositionnalité doit permettre de plonger les raisonnements locaux dans le composant global. Ce principe peut demander une *règle d'adaptation* [Zwiers, 1989]. Pour reprendre l'exemple introduit dans le cas d'une approche par décomposition, l'approche par composition correspond à la preuve de programmes en présence de procédures ou de fonctions. Supposons que nous ayons montré de manière indépendante les deux formules *true* $\{x := 2\}x = 2$ et $x-1 \geq 0\{x := x-1\}x \geq 0$ (ces deux affectations correspondent au corps de deux procédures). Pour composer ces deux preuves nous devons utiliser la *règle d'adaptation* $x = 2 \Rightarrow x - 1 \geq 0$, qui permet de faire le lien entre la postcondition de la première assertion et la précondition de la seconde. Le théorème de composition de Abadi et Lamport s'exprime par la déduction :

$$\frac{E \wedge S_1 \wedge S_2 \Rightarrow E_1 \wedge E_2 \wedge S}{(E_1 \rightarrow S_1) \wedge (E_2 \rightarrow S_2) \Rightarrow (E \rightarrow S)}$$

Où la notation $E \rightarrow S$ permet de spécifier un composant ouvert (une spécification sous forme *rely/guarantee*. Cette spécification décrit le fait que le composant établit la propriété S, si l'environnement satisfait la propriété E. En effet, l'implication $E \Rightarrow S$ est valide dès que l'environnement ne vérifie pas les propriétés attendues. Comme dans le cas précédent, cette déduction n'est correcte sous cette forme que pour les propriétés de sûreté, et sous certaines hypothèses de partitionnement des variables.

3.3.4 Outil support de TLA$^+$: Model checker TLC

TLC est un outil développé par Yung Yan, Jean-Charles Grégoire et Leslie Lamport [Lamport and Yu, 2003] ; il comprend des composants pour l'analyse syntaxique, le formatage, la simulation et l'analyse exhaustive. TLC met en œuvre les techniques de model checking et permet de valider les spécifications écrites. On peut utiliser TLC pour réaliser des tâches liées à la validation de spécifications temporelles TLA$^+$.

3.4 Les diagrammes de prédicats

Un diagramme de prédicats [Cansell *et al.*, 2001a; 2001b] est une abstraction d'un système modélisé par un système à états transitions en particulier d'un système modélisé en TLA$^+$. Il sert à donner une vue abstraite et simple d'un système plus complexe. L'avantage principal du diagramme de prédicats est d'être un graphe fini de prédicats et on peut donc étudier ce graphe considéré comme un système fini avec des techniques de model checking. En outre, on peut développer des diagrammes de prédicats en utilisant une relation de raffinement dont l'objectif est de produire in fine un système conforme à cette abstraction. Les diagrammes de prédicats sont utilisés pour la modélisation et la vérification des propriétés de sûreté et de vivacité sur des systèmes infinis.

3.4.1 Définition

Un diagramme de prédicats [Cansell *et al.*, 2001a] est un graphe fini où les nœuds sont étiquetés par des prédicats booléens et les arcs par des noms d'action, et facultativement des annotations qui permettent à certaines expressions de décroître. Un nœud représente l'ensemble d'états satisfaisant les formules contenues au niveau du nœud. Les arcs représentent des transitions d'état possibles. Une condition d'équité (équité faible ou équité forte) peut être associée à chaque nom d'action. Les transitions peuvent être étiquetées par des annotations de la forme $(\mathrm{t}, \prec)$ ou $(\mathrm{t}, \preceq)$ assurant que le terme t décroît, ou n'augmente pas, au sens de la relation d'ordre bien fondée $\prec$. Un arc (n, m) est étiqueté par une action a si l'action engendre une transition à partir d'un état représenté par n vers un état représenté par m.

Soient P, A et O des ensembles finis de symboles respectivement de prédicats, des actions et des relations d'ordre $\prec$:

- $\overline{\mathrm{P}}$ dénote l'ensemble de prédicats formés à partir de P (les prédicats dans P et leurs négations),
- $\mathrm{O}^{=}$ dénote l'ensemble de symboles de relations d'ordre contenant $\prec$ et $\preceq$ pour chaque $\prec \in \mathrm{O}$.

Formellement, la définition des diagrammes de prédicat [Cansell *et al.*, 2001a] est relative à des ensembles finis P et A contenant les prédicats d'état et les noms d'actions ; par la suite, on utilise $\tau \notin \mathrm{A}$ pour dénoter une action de bégaiement.

Définition 1. Un diagramme de prédicat [Cansell *et al.*, 2001a] $\mathrm{G} = (\mathrm{N}, \mathrm{I}, \delta, \zeta, \mathrm{o})$ sur P et A est composé de :

- un ensemble fini $\mathrm{N} \subseteq 2^{\overline{\mathrm{P}}}$ de nœuds,
- un ensemble fini $\mathrm{I} \subseteq \mathrm{N}$ de nœuds initiaux,
- une famille $(\delta_{\mathrm{a}})_{\mathrm{a} \in \mathrm{A}}$ de relations $\delta_{\mathrm{a}} \subseteq \mathrm{N} \times \mathrm{N}$; on denote par δ l'union des relations δ_{a}, et par $\delta_{=}$ la fermeture reflexive de cette union,
- une fonction $\zeta : \{\mathrm{NF}, \mathrm{WF}, \mathrm{SF}\}$ qui associe une condition d'équité à chaque nom d'action (sans équité, équité faible, équité forte), et
- un arc étiqueté o qui associe des ensembles finis de pairs $\{(\mathrm{t}_1, \prec_1), ..., (\mathrm{t}_{\mathrm{k}}, \prec_{\mathrm{k}})\}$ de termes t_{i} et de symboles $\prec_{\mathrm{i}} \in \mathrm{O}^{=}$ avec les arcs $(\mathrm{n}, \mathrm{m}) \in \delta$.

On dit qu'une action $\mathrm{a} \in \mathrm{A}$ *peut être exécutée* à partir d'un nœud $\mathrm{n} \in \mathrm{N}$ ssi $(\mathrm{n}, \mathrm{m}) \in \delta_{\mathrm{a}}$ est vrai pour quelques $\mathrm{m} \in \mathrm{N}$, on dénote par $\mathrm{En}(\mathrm{a}) \subseteq \mathrm{N}$ l'ensemble de nœuds où a peut être exécutée.

On définit par la suite les traces à travers un diagramme comme des comportements qui correspondent à des exécutions possibles.

Définition 2. Soit $G = (N, I, \delta, \zeta, o)$ un diagramme de prédicat sur P et A. Une exécution de G est une ω-sequence $\rho = (s_0, n_0, a_0)(s_1, n_1, a_1)....$ de triplets où s_i est un état, $n_i \in N$ est un nœud, et $a_i \in A \cup \{\tau\}$ est une action telle que les conditions suivantes sont satisfaites [Cansell *et al.*, 2001a] :

- $n_0 \in I$ est un nœud initial,
- $s_i \models n_i$ est satisfaite pour tout $i \in \mathbb{N}$,
- pour tout $i \in \mathbb{N}$, soit $a_i = \tau$ et $n_i = n_i + 1$, où $a_i \in A$ et $(n_i, n_{i+1}) \in \delta_{a_i}$,
- si $a_i \in A$ et $(t, \prec) \in o(n_i, n_{i+1})$, alors $(s_i, s_{i+1}) \models t' \prec t$,
- si $a_i = \tau$ alors $(s_i, s_{i+1}) \models t' \preceq t$ est satisfaite pour chaque $m \in N$ et $(t, \prec) \in o(n_i, m)$,
- pour chaque action $a \in A$ telle que $\zeta(a) = WF$ il y a une infinité de $i \in \mathbb{N}$ telle que $a_i = a$ ou a ne peut pas être prise à partir de n_i, et
- pour chaque action $a \in A$ telle que $\zeta(a) = SF$, soit $a_i = a$ est vrai pour une infinité de $i \in \mathbb{N}$ ou bien il y a plusieurs $i \in \mathbb{N}$ telle que a peut être exécutée à partir de n_i.

L'ensemble tr(G) de traces de G représente tous les comportements $\sigma = s_0 s_1...$ tel qu'il existe une exécution $\rho = (s_0, n_0, a_0)(s_1, n_1, a_1)...$ de G basée sur les états de σ.

En plus des transitions qui sont représentées explicitement par des arcs du diagramme, il y a les transitions de bégaiement qui restent au niveau du nœud source. Si un nœud n_i a un arc sortant avec une annotation d'ordre $(t, \prec)$, on interdit aux transitions de bégaiement d'augmenter la valeur de t. Les conditions d'équité sont utilisées pour empêcher les occurrences infinies des transitions de bégaiement.

Considérons par exemple, le diagramme de prédicats présenté figure 3.5 ([Cansell *et al.*, 2001a]). Il est formé par quatre nœuds. Les deux nœuds supérieurs sont des nœuds initiaux. En effet, quand le nombre entier n est différent de zéro, il peut être pair ou impair. Le nœud successeur de *"Node0"* est le nœud *"Node2"* par l'occurrence de l'action Eat_0 qui met c_0 à *"e"* sans changer c_1 ou n. En particulier, n est toujours positif et pair. A partir du nœud *"Node2"*, seulement l'action Thk_0 est possible et mène de nouveau à un état où les deux processus sont dans leurs états de *"t"*. Cependant, n pourrait être pair ou impair comme représenté par les deux transitions abstraites du diagramme. La justification des transitions restantes est semblable.

Les traces à travers le diagramme sont des ω-séquences d'états qui satisfont les étiquettes des nœuds et elles sont reliées par les arcs du diagramme ou par les transitions de bégaiement. En outre, les annotations d'ordre excluent les comportements qui forment un cycle pour toujours entre les deux nœuds gauches (*"Node0"* et *"Node 2"*) parce que la valeur assignée à la variable n doit diminuer infiniment souvent, mais n'augmente jamais. Supposons une équité faible à Next, ce qui permet que tous les états du diagramme doivent être visités infiniment souvent durant chaque exécution.

3.4.2 Vérification de systèmes en utilisant les diagrammes de prédicats

Dans cette section, nous décrivons l'utilisation des diagrammes de prédicats pour la vérification des systèmes réactifs. Dans une logique temporelle linéaire telle que TLA, l'inclusion de traces est la relation d'implémentation appropriée. Ainsi, une spécification Spec implémente une

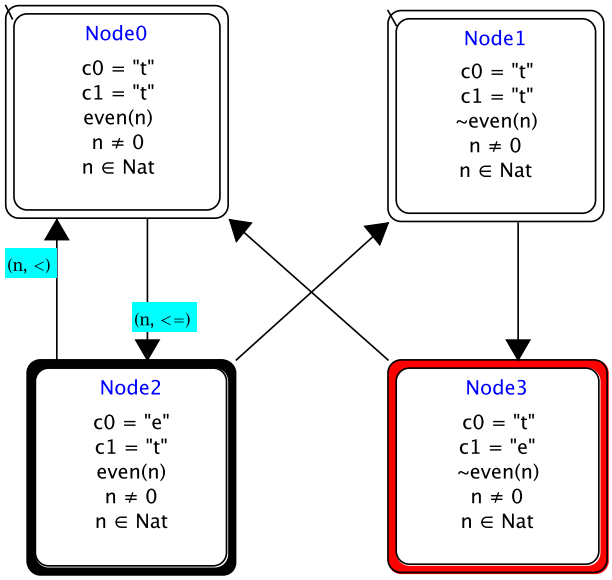


FIG. 3.5 – Diagramme de prédicats de l'exemple du "dining mathematicians"

$$
\begin{array}{l}
\text{Init} \equiv \text{n} \in \text{Nat} \wedge \text{n} \neq 0 \wedge \text{c}_0 = \text{"t"} \wedge \text{c}_1 = \text{"t"} \\
\text{Eat}_0 \equiv \text{c}_0 = \text{"t"} \wedge \text{even(n)} \wedge \text{c}_0' = \text{"e"} \wedge \text{c}_1' = \text{c}_1 \wedge \text{n}' = \text{n} \\
\text{Thk}_0 \equiv \text{c}_0 = \text{"e"} \wedge \text{c}_0' = \text{"t"} \wedge \text{n}' = \text{n div } 2 \wedge \text{c}_1' = \text{c}_1 \\
\text{Eat}_1 \equiv \text{c}_1 = \text{"t"} \wedge \neg\text{even(n)} \wedge \text{c}_1' = \text{"e"} \wedge \text{c}_0' = \text{c}_0 \wedge \text{n}' = \text{n} \\
\text{Thk}_1 \equiv \text{c}_1 = \text{"e"} \wedge \text{c}_1' = \text{"t"} \wedge \text{n}' = 3 * \text{n} + 1 \wedge \text{c}_0' = \text{c}_0 \\
\text{Next} \equiv \text{Eat}_0 \vee \text{Thk}_0 \vee \text{Eat}_0 \vee \text{Thk}_1 \\
\text{v} \equiv \langle \text{c}_0, \text{c}_1, \text{n} \rangle \\
\text{DM} \equiv \text{Init} \wedge \Box[\text{Next}]_\text{v} \wedge \text{WF}_\text{v}(\text{Next})
\end{array}
$$

FIG. 3.6 – Spécification TLA$^+$ de l'exemple "dining mathematicians".

propriété ou une spécification F si et seulement si l'implication Spec $\Rightarrow$ F est valide. Les diagrammes de prédicats peuvent être utilisés pour raffiner cette implication sous deux conditions : premièrement, tous les comportements permis par Spec doivent être des traces du diagramme (le diagramme est une abstraction correcte de Spec), et deuxièmement, chaque trace du diagramme doit satisfaire F. Pour montrer la correction de l'abstraction, on considère les étiquettes des nœuds et des transitions comme des prédicats sur l'espace d'états concret de Spec, et on réduit l'inclusion de traces en un ensemble d'obligations de preuve concernant les états et les transitions. D'un autre côté, pour montrer que le diagramme implique une propriété, on regarde tous les labels comme des variables booléennes. Le diagramme de prédicats peut être vu comme un système de transitions étiqueté fini dont les propriétés temporelles sont vérifiées par *model checking*.

Relation entre specifications et diagrammes de prédicats. Un diagramme de prédicats [Cansell *et al.*, 2001a] G est conforme à une spécification Spec si chaque comportement qui satisfait Spec est une trace du diagramme. En général, prouver la conformance nécessite de raisonner sur les comportements.

Model checking des diagrammes de prédicats. Puisque les diagrammes de prédicat sont des diagrammes de transitions finis, leurs exécutions peuvent être encodées dans le langage d'entrée des models checkers standards tel que SPIN. Les prédicats dans P sont représentés par des variables booléennes, qui sont modifiées selon l'étiquette du nœud courant. On ajoute des variables $b_{(t,\prec)}$, pour chaque terme t et chaque relation $\prec \in O$ telle que $(t, \prec)$ apparaît dans une annotation d'ordre $o(n, m)$. Ces variables sont mises à 2 si la dernière transition prise est étiquetée par $(t, \prec)$, à 1 si elle est étiquetée par $(t, \preceq)$ ou si c'est une transition de bégaiement, et à 0 autrement.

Les conditions d'équité associées au diagramme sont exprimées facilement par des assertions LTL pour Spin. Par la définition 2, on suppose que l'action a est permise quand le nœud courant actif possède un arc sortant dans δ_a. Pour capturer l'effet des annotations d'ordre, on ajoute des formules de type :

$$\Box\Diamond b_{(t,\prec)} = 2 \Rightarrow \Box\Diamond b_{(t,\prec)} = 0$$

Comme des assertions additionnelles pour chaque variable $b_{(t,\prec)}$ introduite. Ces assertions garantissent que les transitions qui font diminuer t en respectant $\prec$ ne peuvent être prises infiniment souvent sauf si quelques transitions prises augmentent la valeur de t infiniment souvent.

Les propriétés F dont les formules atomiques sont contenues dans l'ensemble P de prédicats peuvent être établies par model checking du système de transitions résultant à partir de l'encodage. Si la vérification réussit, alors chaque trace du diagramme satisfait F et par transitivité d'inclusion de trace et il s'ensuit que F est satisfaite par chaque spécification qui conforme le diagramme. Dans l'autre cas, des contre-exemples sont produits par le model checker, parce que des détails sont perdus dans l'abstraction. Cependant, ces contre-exemples sont utiles dans le but de corriger l'abstraction, par exemple en ajoutant des annotations d'ordre. Évidemment, la taille des diagrammes qui peuvent être analysés est surtout limitée par le nombre des conditions d'équité et des annotations d'ordre du diagramme.

Dans l'exemple du "Dining mathematicians" présenté figure 3.5, les propriétés suivantes peuvent être toutes vérifiées à partir du diagramme de prédicats.

(Pos)	$\Box(n \in Nat \wedge n \neq 0)$	(Excl)	$\Box\neg(c_0 = \text{"e"} \wedge c_1 = \text{"e"})$
(Live0)	$\Box\Diamond(c_0 = \text{"e"})$	(Live1)	$\Box\Diamond(c_1 = \text{"e"})$

Les premières propriétés sont des invariants qui assurent que n est un nombre naturel positif et que l'exclusion mutuelle est assurée. Les deux autres propriétés sont des propriétés de vivacité. Comme il est indiqué à la définition 2, la vérification de la propriété (Live1) s'appuie sur les annotations d'ordre qui interdisent le cycle entre les nœuds "*Node0*" et "*Node2*".

3.4.3 Raffinement des diagrammes de prédicats

Les diagrammes de prédicats utilisent la technique du développement top-down. Commençant par une abstraction qui représente le comportement global du système, des détails sont ajoutés petit à petit jusqu'à obtenir une spécification du système suffisamment concrète. Techniquement, une relation de raffinement [Cansell *et al.*, 2001a] entre les diagrammes est définie. Comme dans une méthodologie générale, on a besoin d'une définition en termes des "*obligations de preuve locales*". En d'autres termes, la structure globale du diagramme abstrait va être préservée dans

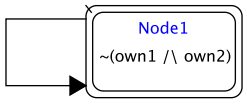


FIG. 3.7 – Exclusion mutuelle : diagramme initial.

le diagramme raffiné.

Raffinement structurel. On suppose initialement que le diagramme raffiné G^1 interprète tous les prédicats et les actions apparaissant dans le diagramme abstrait G^2. En particulier, l'ensemble des variables d'état qui apparaissent dans G^1 est un sur-ensemble de celles qui apparaissent dans G^2.

Définition 3. Soient deux diagrammes de prédicats $G^1 = (N^1, I^1, \delta^1, \zeta^1, o^1)$ sur P^1 et A^1 et $G^2 = (N^2, I^2, \delta^2, \zeta^2, o^2)$ sur P^2 et A^2 où $P^1 \supseteq P^2$ et $A^1 \supseteq A^2$, et soit $f : N^1 \to N^2$. On dit que G^1 raffine d'une manière structurelle G^2 selon f si et seulement si (ssi) les conditions suivantes sont satisfaites :

1. $f(I^1) \subseteq I^2$
2. $\models n \Rightarrow f(n)$ est satisfaite pour chaque nœud $n \in N^1$.
3. Pour chaque $n, m \in N^1$ et tout $a \in A^1$ telle que $(n, m) \in \delta^1_a$:
   (a) si $a \in A^2$ alors $(f(n), f(m)) \in \delta^2_a$, et
   (b) si $a \in A^1 \backslash A^2$ alors $(f(n), f(m)) \in \delta^2_=$.
4. Pour chaque $n, m \in N^1$, tout $a \in A^1$ telle que $(n, m) \in \delta^1_a(n, m)$, tous les termes t et les relations $\prec \in O^=$:
   (a) $(t, \prec) \in o^1(n, m)$ quand $(t, \prec) \in o^2(f(n), f(m))$,
   (b) $(t, \preceq) \in o^1(n, m)$ quand $f(n) = f(m)$ et $(t, \prec) \in o^2(f(n), m')$ pour quelques $m' \in N^2$.
5. Pour chaque exécution $\rho^1 = (s_0, n_0, a_0)(s_1, n_1, a_1)...$ de G^1 et chaque action $a \in A^2$ telle que $\zeta^2(a) = \text{WF}$, on a soit $a_i = a$ ou $f(n_i) \notin \text{dom}(\delta^2_a)$ est satisfaite pour plusieurs $i \in \mathbb{N}$.
6. Pour chaque exécution $\rho^1 = (s_0, n_0, a_0)(s_1, n_1, a_1...$ de G^1 et chaque action $a \in A^2$ telle que $\zeta^2(a) = \text{SF}$, soit $a_i = a$ pour une infinité de $i \in \mathbb{N}$ ou $f(n_i) \in \text{dom}(\delta^2_a)$ pour plusieurs $i \in \mathbb{N}$.

La définition du raffinement structurel [Cansell *et al.*, 2001a] est telle que les conditions (1)-(4) sont purement structurelles ou peuvent être vérifiées sans utiliser un raisonnement temporel. D'un autre côté, les conditions (5) et (6) sont basées sur les exécutions du diagramme et peuvent être établies par model checking utilisant l'encodage décrit dans la section précédente. Notons en particulier qu'on n'exige pas des conditions d'équité de haut niveau pour être syntaxiquement préservées dans le diagramme de raffinement.

Les conditions (1)-(4) permettent de garantir que les transitions entre les deux diagrammes doivent être reliées. Les labels des nœuds de G^1 impliquent celles des nœuds correspondants de G^2. Les nœuds initiaux de G^1 sont reliés aux nœuds initiaux de G^2 (condition 1). Les transitions dans G^1 doivent correspondre à des transitions dans G^2 (avec bégaiement possible) (condition 3) et les annotations d'ordre présentes dans G^2 doivent être préservées dans G^1 (condition 4).

D'un autre côté, on exige que les transitions de G^1 préservent n'importe quelles annotations d'ordre supposées pour les transitions correspondantes de G^2, et les transitions de G^1 correspondent à des étapes de bégaiement en G^2 qui n'augmentent pas les termes pour lesquels les annotations d'ordre sont présentes dans G^1. Cette condition de préservation stricte peut être relaxée en permettant une transition du diagramme abstrait avec une contrainte d'ordre associée $(t, \prec)$ pour être simulée par une séquence de transitions dans le diagramme raffiné telle que chaque transition garantit $t' \preceq t$ et au moins une transition garantit $t' \prec t$.

Le raffinement structurel garantit l'inclusion de traces [Cansell *et al.*, 2001a]. En particulier, toutes les propriétés de G^2 restent valide pour G^1.

Exemple : Le protocole d'exclusion mutuelle

Nous allons maintenant argumenter que la notion de raffinement structurel est flexible en exécutant une séquence de raffinements qui conduit aux protocoles d'exclusion mutuelle de deux processus différents, un problème standard de référence dans la modélisation des systèmes réactifs. La contrainte est de garantir que les deux processus ne peuvent jamais accéder simultanément à la ressource partagée. Le plus simple diagramme exprimant cette contrainte est donné figure 3.7. Il n'indique pas comment l'exclusion mutuelle est réalisée, ni s'il satisfait des propriétés de vivacité. Une première étape vers l'implémentation est représentée par le diagramme de la figure 3.8 et dans les diagrammes suivants, on utilise une convention de numérotation pour indiquer la fonction de raffinement : un nœud x.y du diagramme raffiné est lié au nœud x du diagramme abstrait. Il est simple de voir que le deuxième diagramme est un raffinement structurel du premier parce que chaque étiquette d'un nœud implique $\neg(own_1 \wedge own_2)$ et chaque transition est permise par la boucle sur le seul nœud du diagramme initial.

En continuant le développement, on joute des propriétés de vivacité pour les processus qui ont demandé une ressource. On ajoute ainsi les prédicats Req_1 et Req_2 qui indiquent quel processus est satisfait et on divise les nœuds en conséquence (Figure 3.9).

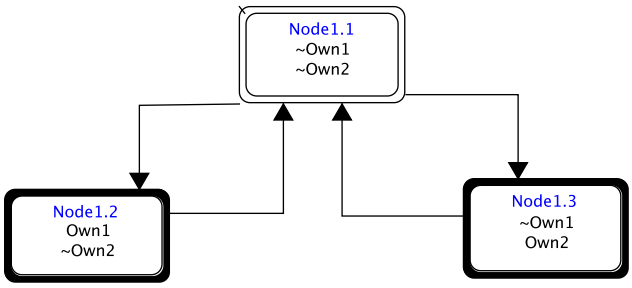


FIG. 3.8 – Exclusion mutuelle : premier raffinement.

On a aussi introduit les actions $Take_1$, $Take_1$, $Drop_1$, $Drop_2$ qui modélisent l'acquisition et la libération de la ressource par les processus. Les conditions de la définition 3 sont aussi vérifiées d'une manière simple, en particulier les transitions telles que, l'un à partir du nœud 1.1.1 vers le nœud 1.1.2 où le processus 1 demande la ressource sont couverts par les transitions de bégaiement dans le diagramme précèdent. On a besoin de l'équité forte pour les actions $Take_i$, et de l'équité faible pour les actions $Drop_i$ pour garantir que les conflits sont résolus par équité et qu'aucun processus ne monopolise la ressource. La technique de model checking établit que le

diagramme satisfait les propriétés de vivacité :

$$\text{Req}_1 \rightsquigarrow \text{own}_1 \qquad \text{Req}_2 \rightsquigarrow \text{own}_2$$

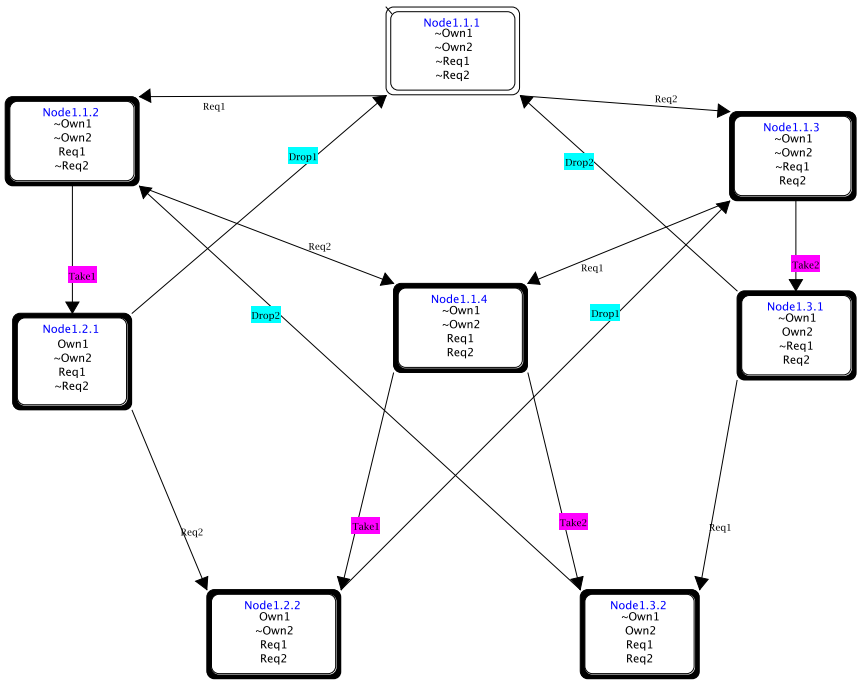


FIG. 3.9 – Exclusion mutuelle : Ajout des requètes.

On a maintenant obtenu un diagramme qui représente l'implémentation de sémaphores de l'exclusion mutuelle : le sémaphore est pris dans ses états où quelques processus possèdent la ressource. On a besoin seulement de diviser le nœud 1.1.4 en deux sous nœuds comme c'est indiqué figure 3.9 ; les nœuds 1.1.4.1 et 1.1.4.2 associent des propriétés aux différents processus. Parce qu'il n'y a pas de conflit entre les actions Take_1 et Take_2, la contrainte d'équité forte pour ces actions est remplacée par l'équité faible. Il est facile d'établir les conditions (1)-(4) pour prouver que le diagramme de la figure 3.9 est structurellement raffiné par celui de la figure 3.10, parce que le dernier diagramme permet moins de transitions que le premier. Prouver le raffinement des conditions d'équité originales est plus subtil parce que les nœuds 1.1.4.1 et 1.1.4.2 sont les deux reliés au nœud abstrait 1.1.4, où les actions Take_1 et Take_2 sont permises. Cependant, l'alternance de priorités assure que les conflits sont résolus avec équité, et en fait, le model checker confirme que la condition (6) de la définition (3) est satisfaite. Le diagramme de prédicats présenté par la figure 3.10 hérite de toutes les propriétés de sûreté et de vivacité obtenues pour le diagramme précédent.

3.4.4 Outil support des diagrammes de prédicats DIXIT

DIXIT [Fejoz *et al.*, 2005] est un outil qui supporte l'utilisation des diagrammes de prédicats pour la vérification des systèmes à états infinis avec la particularité de se focaliser sur la preuve des

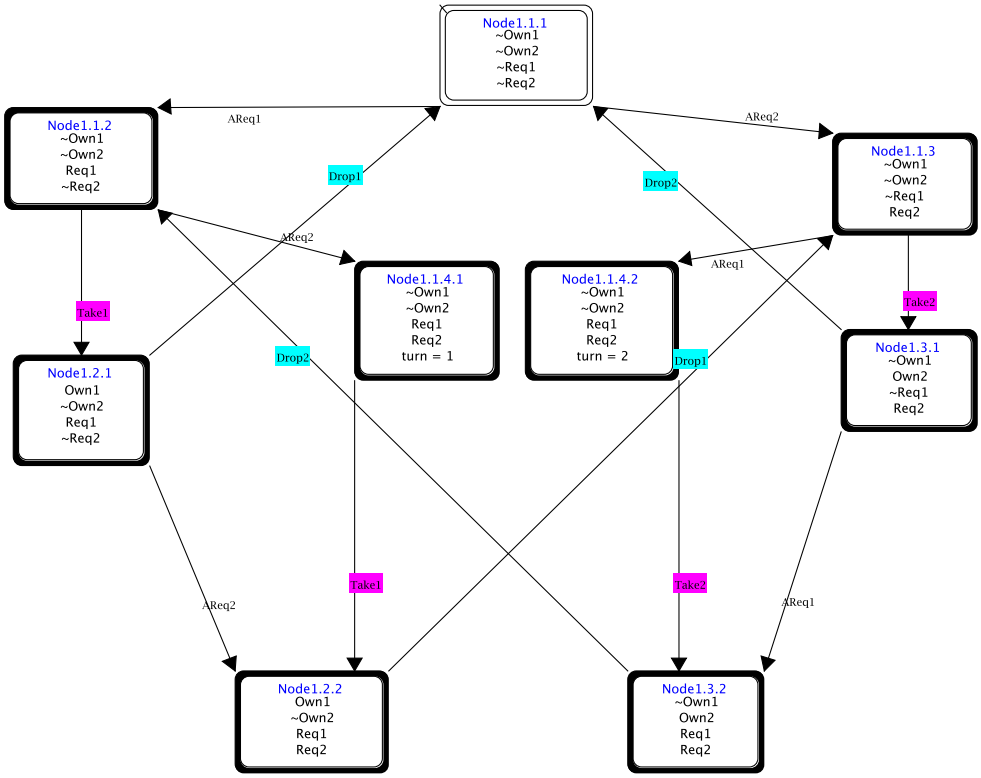


FIG. 3.10 – Exclusion mutuelle : implementation.

propriétés de vivacité. La présente version permet la vérification des modules TLA$^+$. Il permet à un utilisateur de dessiner des diagrammes de prédicats et de vérifier les propriétés de correction en utilisant la technique de model checking. En plus, il permet de vérifier qu'un diagramme de prédicat est un raffinement correct d'un autre. Les obligations de preuve pour établir le raffinement indique que le diagramme raffiné reflète la structure des transitions du diagramme abstrait. D'un autre côté, les conditions d'équité du diagramme peuvent être implémentées d'une manière flexible. Aussi, DIXIT peut générer des obligations de preuve permettant de déduire que le diagramme de prédicats est conforme à une spécification TLA$^+$.

3.5 Conclusion

Nous avons présenté tout au long de ce chapitre le langage de modélisation TLA$^+$ et les diagrammes de prédicats. TLA$^+$ est un langage qui permet la modélisation des systèmes réactifs et concurrents et se base sur un système de preuve des propriétés de sûreté et de vivacité. L'outil TLC met en oeuvre les techniques de model checking et permet de valider les spécifications écrites en TLA$^+$ sur des systèmes finis.

Un diagramme de prédicats est une abstraction d'un système modélisé en TLA$^+$. Il permet de prouver des propriétés de vivacité sur des systèmes infinis. Les diagrammes de prédicats supportent le raffinement et utilisent la technique du développement top-down. Des obligations de preuve sont générées pour montrer que la structure globale et les propriétés du diagramme

abstrait vont être préservées dans le diagramme raffiné. L'outil DIXIT supporte l'utilisation des diagrammes de prédicats pour la vérification des systèmes à états infinis avec la particularité de se focaliser sur la preuve des propriétés de vivacité.

Chapitre 4

Raffinement et composition dans le développement de systèmes complexes

4.1 Introduction

Dans ce chapitre, nous présentons les techniques de raffinement et de composition sur lesquels se basent nos contributions pour le développement de systèmes automatisés.

Le raffinement est une technique de développement qui permet de traiter des systèmes complexes en introduisant graduellement les éléments d'un cahier des charges. La vérification consiste à prouver que le modèle abstrait satisfait les exigences de la partie du cahier des charges qu'il prend en compte et, à chaque étape, le modèle raffiné conserve les propriétés du modèle qu'il raffine et satisfait les nouveaux éléments du cahier des charges introduits dans le modèle.

La composition ou plutôt la décomposition-composition, consiste à concevoir un modèle comme un assemblage de parties (moins complexes) conçues séparément. On applique ici le principe : diviser pour régner. L'opération de composition consiste à définir comment les parties communiquent et interagissent entre elles. La composition facilite la preuve, en effet, chaque partie est plus facile à prouver que l'ensemble et l'obligation de preuve de l'ensemble s'exprime en faisant l'hypothèse que les parties sont prouvées.

Enfin, nous précisons que ces deux techniques sont des paradigmes de spécification et de vérification compatibles pour s'assurer de la sécurité des logiciels. Dans la section 2, nous présentons la technique de raffinement utilisée pour le développement de systèmes complexes. Dans la section 3, nous présentons la technique de composition puis nous citons les travaux existants autour de cette technique.

4.2 Raffinement et vérification

Le raffinement est un processus de développement et de vérification de plus en plus répandue pour construire progressivement des programmes corrects : il consiste à dériver par étapes successives une spécification initiale en vérifiant que chaque transformation du programme préserve bien sa correction vis-à-vis de la spécification précédente. D'une manière générale, on distingue différents types de raffinement :

- Le raffinement des données consiste à remplacer les structures de données abstraites (ensembles, etc.) par des structures plus concrètes et plus proches de celles de l'implantation

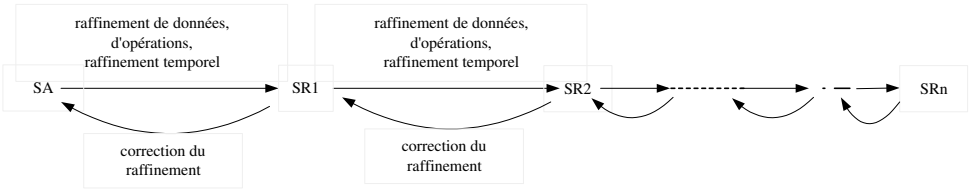


FIG. 4.1 – Raffinement : processus de conception et de vérification.

(tableaux, etc.),

- Le raffinement d'opérations, qui consiste à détailler les opérations utilisées tout en leur permettant de manipuler les données raffinées,
- Le raffinement temporel, qui consiste à réduire la granularité du temps. Alors qu'au niveau abstrait, on n'observait qu'une seule action atomique, en divisant les unités de temps, on observe de nouvelles actions qui détaillent le comportement de notre système.

Le raffinement est un processus de conception qui propose de passer d'une spécification abstraite SA à une spécification plus concrète SR en ajoutant des détails, mais c'est aussi un processus de vérification. En effet, il est nécessaire de s'assurer de la correction du raffinement pour affirmer qu'une spécification concrète donnée est bien une vue plus détaillée d'une spécification abstraite de départ. Le raffinement s'exprime et se vérifie de différentes manières. La méthode la plus courante est la recherche d'une relation de simulation de la spécification concrète par sa spécification abstraite, que nous détaillons la section suivante.

4.2.1 Simulation de la relation de raffinement

Les méthodes formelles utilisent généralement des relations de simulation ou des outils pour assurer la correction du raffinement. Les relations de simulation permettent de prouver le raffinement entre deux systèmes à l'aide de conditions suffisantes. Il existe deux principales relations : *forward* et *backward*. Toutes les méthodes formelles ne proposent pas les deux. Dans les cas de B et B événementiel, le raffinement est prouvé à l'aide d'obligations de preuve correspondant à une simulation *backward*, c'est-à-dire elle passe des post-conditions aux pré-conditions. La disponibilité d'outils a également son importance : une méthode n'utilisant qu'un seul type de relation avec un bon outil peut s'avérer dans la pratique plus efficace et plus utile qu'une méthode sans outil. Une autre technique consiste à vérifier le raffinement par model-checking, comme dans le cas de CSP.

Dans ce qui suit, nous présentons les principales formes de raffinement afin d'illustrer la diversité des définitions possibles du raffinement.

4.2.2 Exemples de raffinement

Dans cette section, nous présentons le raffinement dans plusieurs sémantiques. Trois formes principales de raffinement sont considérées dans cet état de l'art : le raffinement de séquences d'opérations (comme dans les machines B [Abrial, 1996b] ou les types de données abstraits Z [Spivey, 1992]), le raffinement opérationnel au niveau des transitions d'états (comme en B

événementiel [Abrial, 2000]) et le raffinement dénotationnel des traces, échecs et divergences (comme en CSP [Roscoe, 1998]).

La notion de raffinement a été introduite dans les années 1970 par Dijkstra [Dijkstra, 1976], puis formalisée par Back [Back, 1978; Back and v. Wright, 1998]. Plusieurs travaux ont ensuite développé cette notion de raffinement, en particulier Abadi et Lamport [Abadi and Lamport, 1988] à propos du raffinement TLA, Morris [Morris, 1987], Morgan [Morgan, 1994], Pnueli [Pnueli, 1992] à propos du raffinement LTL et Abrial [Abrial, 1996b] pour le raffinement de systèmes B.

*** Raffinement de Dijkstra**

Dijkstra [Dijkstra, 1976] est le premier qui a parlé de raffinement. Pour vérifier la correction des programmes, Dijkstra a défini le *langage des commandes gardées* et il a introduit la notion de *plus faible précondition*, notée *wp*, qui s'appuie sur la logique de Hoare. L'opérateur *wp* permet de calculer la *plus faible précondition* qui garantisse la terminaison d'un programme S et qui laisse le système dans un état qui satisfait la postcondition Q. Pour un programme S et une postcondition Q, *wp(S,Q)* est la *plus faible précondition* P telle que $P <S> Q$.

La correction et le raffinement sont maintenant exprimés dans cette sémantique (sémantique axiomatique). Un programme S de précondition P et de postcondition Q est alors correct si : $\mathrm{P} \Rightarrow \mathrm{wp(S, Q)}$. Dans la sémantique *wp*, le raffinement se traduit par :

$$\mathrm{S} \sqsubseteq \mathrm{S}' \Leftrightarrow (\forall \mathrm{Q}, \mathrm{wp(S, Q)} \Rightarrow \mathrm{wp(S', Q)})$$

A partir d'un programme S et d'une postcondition Q, l'opérateur *wp* permet de revenir sur les pré-conditions possibles (la sémantique *wp* est dite *backward*).

La sémantique du raffinement en terme de plus faible pré-condition a servi de base pour de nombreux raffinements, entre autres pour le raffinement des systèmes d'actions [Butler, 1996], le raffinement CSP [Hoare, 1985], le raffinement Z [Bow,] et le raffinement B [Abrial, 1996b].

*** Raffinement dans les systèmes d'actions**

Les systèmes d'actions de Back [Back and Kurki-Suonio, 1983; Back and von Wright, 1998] et de Butler [Butler, 1996] représentent une version étendue du *langage des commandes gardées* de Dijkstra, fondés sur le calcul des plus faibles préconditions *wp*.

Un système d'actions est un quadruplet $\mathrm{AS} = (\mathrm{E}, \mathrm{v}, \mathrm{A_i}, \mathrm{A})$ où :

- E est un ensemble d'étiquettes d'actions,
- v est l'ensemble de variables d'état,
- $\mathrm{A_i}$ est l'action d'initialisation et
- A est un ensemble d'actions gardées.

Une action $\mathrm{A_e} \in \mathrm{A}$ est de la forme **action** $\mathrm{A_e}$: - $\mathrm{grd(A_e)} \rightarrow \mathrm{subst(A_e)}$ et elle est activable si l'état courant du système satisfait la garde $\mathrm{grd(A_e)}$ de l'action.

Soient P et P′ deux prédicats. On écrira $\mathrm{P} \Rrightarrow \mathrm{P}'$, si tout état satisfaisant P satisfait également P′. Le raffinement d'actions est défini dans [Butler, 1996] de la manière suivante :
une action $\mathrm{R_e}$ raffine une action $\mathrm{A_e}$, notée $\mathrm{A_e} \preceq \mathrm{R_e}$, si pour tout prédicat P, $\mathrm{wp(subst(A_e), P)} \Rrightarrow \mathrm{wp(subst(R_e), P)}$. Cette expression du raffinement est similaire au raffinement de Dijkstra.

Dans un raffinement, les variables abstraites sont remplacées par des variables plus concrètes. Un invariant d'abstraction abs exprime le lien entre une variable d'état $\mathrm{v_A}$ d'un système d'actions abstrait $\mathrm{AS_A}$ et une variable d'état $\mathrm{v_R}$ d'un système d'actions plus concret $\mathrm{AS_R}$. Le raffinement d'actions s'exprime alors de la manière suivante :

Une action R_e raffine, sous invariant d'abstraction abs, une action A_e, notée $A_e \preceq_{abs} R_e$, si pour tout prédicat P indépendant de v_R, il existe v_A tel que :
$abs \wedge wp(subst(A_e), P) \Rrightarrow wp(subst(R_e), \exists v_A.(abs \wedge P))$.

Le raffinement d'un système d'actions complet est défini en termes de simulation sous invariant d'abstraction entre le système d'actions abstrait et son homologue raffiné. Une relation de simulation sous invariant d'abstraction abs est une relation entre deux systèmes d'actions AS_A et AS_R telle que, pour tout $e \in E$,

- $A_e \preceq_{abs} R_e$,
- $\exists v_A.(abs \wedge grd(A_e)) \Rrightarrow grd(R_e)$

La première condition assure que chaque action de AS_R raffine son homologue dans AS_A. La deuxième condition assure que AS_R n'introduit pas de nouveaux blocages.

Notons qu'on ne peut pas introduire de nouvelles actions lors du raffinement d'un système d'actions.

*** Raffinement dans CSP**

Le langage CSP (Communicating Sequential Process) [Hoare, 1985] permet de spécifier le comportement de systèmes grâce à une algèbre de processus. Trois modèles sémantiques [Roscoe, 1998] sont donnés :

- Le modèle des *traces* associe à chaque processus les séquences finies d'événements que le processus peut effectuer, noté *traces(P)*. Ce modèle permet de représenter les comportements possibles des processus sous forme de traces.
- Le modèle des *échecs stables* associe à chaque processus P des couples de la forme (σ, E), où σ est une trace finie admise par P et E est l'ensemble des événements que le processus ne peut pas exécuter après avoir exécuté les événements de σ. L'ensemble de ces couples est noté *failures(P)*. Ce modèle permet de caractériser les blocages de P. En effet, si E est égal à l'ensemble des événements exécutables par P, alors P se retrouve bloqué.
- le modèle des *échecs-divergences* associe à chaque processus P l'ensemble de ses échecs stables et l'ensemble de ses divergences. Un processus P n'est divergent que s'il se retrouve dans un état dans lequel les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences de P, noté *divergences(P)*, est l'ensemble des traces σ telles que le processus se retrouve dans un état divergent après avoir exécuté σ. Si le processus est déterministe, alors *divergences(P)* est vide.

La vérification du raffinement CSP consiste alors à calculer et à comparer les modèles sémantiques de deux processus. Il dépend donc du modèle considéré, et se définit en termes d'inclusion de modèles. Par exemple, dans le cas du modèle des échecs-divergences, si P et Q sont deux processus, alors Q raffine P si :

- $traces(Q) \subseteq traces(P)$, dans le cas du modèle des traces,
- $failures(Q) \subseteq failures(P)$, dans le cas du modèle des échecs stables
- $divergences(Q) \subseteq divergences(P)$, dans le cas du modèle des échecs-divergences,

Il existe toutefois des outils comme FDR [Broadfoot and Roscoe, 2000; Welch, 2002] qui permettent d'analyser les traces, les échecs et les divergences d'un processus. Ils sont cependant limités par la complexité des modèles des expressions de processus. Comme pour le model checking, l'analyse d'un processus avec FDR peut échouer à cause d'une explosion du nombre d'états.

*** Raffinement des systèmes de transitions**

Dans [Bellegarde *et al.*, 2000b], les auteurs ont exprimé le raffinement des systèmes d'événements B en termes de relation de simulation entre deux systèmes de transitions finis. Un système de transitions étiqueté (LTS) est un tuple de la forme $\langle S, S_0, E, \rightarrow \rangle$ où :

- S est un ensemble d'états,
- S_0 est l'ensemble des états initiaux du système ($S_0 \subseteq S$),
- E est un ensemble fini d'étiquettes de transitions,
- $T \subseteq S \times E \times S$ est une relation de transition, et

Dans le cadre de programmes utilisant des variables V, le LTS est augmenté d'une fonction l qui affecte à chaque état s du système des valeurs aux variables V. L'état s'exprime alors sous la forme d'une conjonction de propositions d'états correspondant aux différentes variables. Un chemin entre deux états s et s′ de S est une séquence finie de transitions e_1, ...,e_n de T telles qu'il existe des états intermédiaires s_1, ..., s_{n-1} de S vérifiant :

$$s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} s_3 \ldots\ldots \xrightarrow{e_{n-1}} s_{n-1} \xrightarrow{e_n} s'$$

Soient $TS_1 = \langle S_1, S_{01}, E_1, \rightarrow_1, l_1 \rangle$ et $TS_2 = \langle S_2, S_{02}, E_2, \rightarrow_2, l_2 \rangle$ deux systèmes de transitions, syntaxiquement le raffinement de systèmes de transitions finis se traduit par les conditions suivantes :

- le raffinement peut introduire de nouvelles actions ($E_1 \subseteq E_2$),
- le raffinement peut introduire de nouvelles variables, quant aux anciennes variables, elles sont renommées.
- un *prédicat de collage* $I_{1,2}$ exprime le lien entre les variables du système abstrait et celles du système raffiné. Pour faire le lien entre les états abstraits de S_1 et les états concrets de S_2, on utilise l'invariant de collage $I_{1,2}$ et une relation de collage $\rho : S_2 \times S_1$ telle que :
  $s_2\ \rho\ s_1 \Leftrightarrow ((l_2(s_2) \wedge I_{1,2}) \Rightarrow l_1(s_1))$

Pour vérifier que le système de transitions (LTS) ST_1 est raffiné par ST_2, il suffit de montrer que les conditions suivantes sont satisfaites. Elles permettent de prouver que les nouvelles transitions introduites par ST_2 ne contredisent pas le comportement de ST_1.

- Le raffinement strict des transitions consiste à vérifier que pour chaque transition de $\rightarrow_2$, il existe une transition correspondante dans le LTS abstrait :
  $(s_2\ \rho\ s_1 \wedge\ s_2 \xrightarrow{e}_2 s'_2) \Rightarrow (\exists s'_1.\ s_1 \xrightarrow{e}_1 s'1 \wedge\ s'_2\ \rho\ s'_1)$
- Le bégaiement de transition est l'introduction d'une nouvelle transition τ entre deux états concrets correspondant au même état abstrait :
  $(s_2\ \rho\ s_1 \wedge\ s_2 \xrightarrow{\tau}_2 s'_2) \Rightarrow (s'_2\ \rho\ s_1)$
- Un état qui n'a pas de transition avec un autre état est un état qui bloque dans le LTS. Il s'agit d'un blocage du système de transitions. Pour ne pas introduire de nouveaux blocages pendant le raffinement, on vérifie que chaque état qui bloque dans ST_2 (noté $\nrightarrow_2$) correspond à un état qui bloque dans ST_1 (noté $\nrightarrow_1$) :
  $(s_2\ \rho\ s_1 \wedge s_2 \nrightarrow_2) \Rightarrow (s_1 \nrightarrow_1)$
- Pour éviter que les nouvelles transitions τ ne prennent indéfiniment le contrôle, on vérifie la non τ-divergence. Un état s_1 de E_1 est collé à plusieurs états concrets.
  $s_2\ \rho\ s_1 \Rightarrow \neg\ (s_2 \xrightarrow{\tau}_2 s'_2 \xrightarrow{\tau}_2 s''_2 \xrightarrow{\tau}_2 \ldots \xrightarrow{\tau}_2 \ldots)$
- Le LTS concret doit préserver le non-déterminisme externe de T_1 :
  $(s_1 \xrightarrow{e} s'_1 \wedge\ s_2\ \rho\ s_1) \Rightarrow (\exists s'_2, s''_2, s''_1.(s'_2\ \rho\ s_1 \wedge\ s'_2 \xrightarrow{e} s''_2 \wedge\ s_1 \xrightarrow{e} s''_1 \wedge s''_2\ \rho\ s''_1))$

- Enfin, pour tout état initial concret, il existe un état initial abstrait qui lui est collé : $\forall s_2 \in S_{02} \exists s_1 \in S_{01}.s_2 \rho s_1$

*** Raffinement LTL**

Pnueli [Pnueli, 1992] a introduit la logique temporelle linéaire (LTL) pour exprimer la sémantique temporelle d'un système de transitions par une formule de Logique Temporelle Linéaire (LTL). Le système ainsi que les propriétés sont vérifiés dans le même formalisme.

Le raffinement s'exprime alors très simplement : soient ϕ_A et ϕ_R deux formules LTL, une spécification ϕ_R raffine une spécification ϕ_A si toutes les exécutions qui satisfont ϕ_R satisfont également ϕ_A, autrement dit, quand $\phi_R \Rightarrow \phi_A$. L'avantage principal du raffinement LTL est la préservation des propriétés LTL par transitivité de l'implication. Si une spécification abstraite ϕ_A satisfait une propriété ψ (c.à.d. si $\phi_A \Rightarrow \psi$ et si ϕ_A est raffiné par ϕ_R, alors ϕ_R satisfait aussi ψ). La plupart du temps, il est par contre difficile de vérifier l'implication $\phi_R \Rightarrow \phi_A$.

Dans ce qui suit, nous étudions la deuxième technique de développement de systèmes automatisés : la composition.

4.3 Composition

La composition est une technique de développement horizontal qui permet de maîtriser la complexité des systèmes de grande taille. Après avoir identifié et modélisé des sous-systèmes ayant une cohérence interne, un des problèmes essentiels est de définir comment composer ces composants pour obtenir le système global. Plus généralement, on peut aussi être amenés à utiliser des composants standards et à les combiner pour obtenir le système souhaité. Dans ce cas, l'effort essentiel porte sur la définition de l'interaction entre les composants. Pour ce faire, il convient de définir des contraintes aux transitions du produit des deux systèmes de transition. On utilise le plus souvent le terme de synchronisation pour définir de telles contraintes. Le problème posé par la composition réside dans la préservation des propriétés des composants dans le système complet.

Dans cette section, nous présentons tout d'abord les compositions classiques des systèmes de transitions. Par la suite, nous présentons les compositions qui ont été proposées dans le cadre de la méthode B.

4.3.1 Compositions classiques des systèmes de transitions

Pour modéliser un système de transitions complexe, il convient d'identifier ses sous-systèmes, de les modéliser comme des systèmes de transitions indépendants, puis de définir les transitions entre ces sous-systèmes, autrement dit de les synchroniser.

Il existe de nombreuses manières de définir la synchronisation de diagrammes de transitions. Nous allons présenter ici brièvement les principales méthodes.

Exemple. La figure 3 présente deux systèmes de transition étiquetés (STE) qui serviront à illustrer les différentes définitions que nous allons présenter dans la suite de cette section.

- La figure 5.1(a) illustre le STE qui modélise le comportement d'un compteur modulo 2,
- La figure 5.1(b) illustre le STE qui modélise le comportement d'un compteur modulo 3.

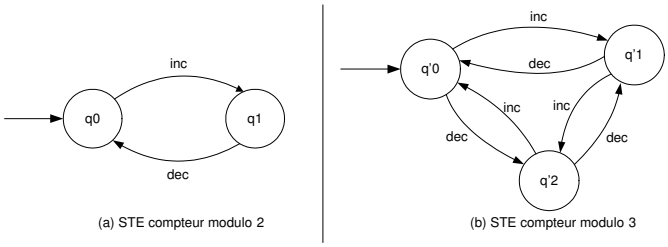


FIG. 4.2 – STEs des compteurs modulo 2 et modulo 3

*** Composition libre des systèmes de transitions (produit cartésien)**

Le cas le plus simple se présente lorsque le système peut être décomposé en sous-systèmes n'intéragissant pas entre eux. Le système de transitions résultant est alors le produit cartésien des systèmes de transitions représentant les composantes.

Notation. On introduit une nouvelle étiquette de transition '-' correspondant à l'action neutre qui consiste à ne rien modifier. On notera $E_1 \otimes E_2 \triangleq ((E_1 \cup \{-\}) \times (E_2 \cup \{-\})) \setminus \{(-,-)\}$. De même, on notera (q_1, q_2) un état de $Q_1 \times Q_2$ et $(q_1, q_2) \overset{(e_1,e_2)}{\longrightarrow} (q'_1, q'_2)$ une transition de $(Q_1 \times Q_2) \times (E_1 \otimes E_2) \times (Q_1 \times Q_2)$.

Définition.

Soient $S_1 = \langle Q_1, Q_{01}, E_1, T_1, l_1 \rangle$ et $S_2 = \langle Q_2, Q_{02}, E_2, T_2, l_2 \rangle$ deux systèmes de transitions. Le produit cartésien de S_1 et S_2, noté $S_1 \times S_2$ est défini par le cinq-uplet $S = \langle Q, Q_0, E, T, l \rangle$ comme suit :

- $Q = Q_1 \times Q_2$,
- $Q_0 = Q_{01} \times Q_{02}$,
- $E = E_1 \otimes E_2$,
- $T = Q \times E \times Q$ est définie ainsi :
  - $(q_1, q_2) \overset{(e_1,-)}{\longrightarrow} (q'_1, q_2) \in T$ si $q_1 \overset{e_1}{\rightarrow} q'_1 \in T_1$
  - $(q_1, q_2) \overset{(-,e_2)}{\longrightarrow} (q_1, q'_2) \in T$ si $q_2 \overset{e_2}{\rightarrow} q'_2 \in T_2$
  - $(q_1, q_2) \overset{(e_1,e_2)}{\longrightarrow} (q'_1, q'_2) \in T$ si $q_1 \overset{e_1}{\rightarrow} q'_1 \in T_1 \wedge\ q_2 \overset{e_2}{\rightarrow} q'_2 \in T_2$
- $l((q_1, q_2)) = l_1(q_1) \cup l_2(q_2)$.

Exemple. La figure 4.3 montre le produit cartésien d'un compteur modulo 2 et d'un compteur modulo 3. Ce système de transition contient 2*3= 6 états. Il présente tous les comportements possibles issus de ces deux compteurs, aussi bien des comportements au niveau des composants : incrémenter ou décrémenter le premier compteur ((inc,-), (dec,-)), incrémenter ou décrémenter le deuxième compteur ((-,inc),(-,dec)) ; que des comportements synchronisés : incrémenter/incrémenter (inc,inc), incrémenter/décrémenter (inc,dec), décrémenter/décrémenter (dec,dec) ou décrémenter/incrémenter (dec,inc). Chaque état est constitué de deux composantes distinctes qui sont en fait des états des compteurs individuels et les transitions entre les états du système sont obtenues à partir des compteurs individuels.

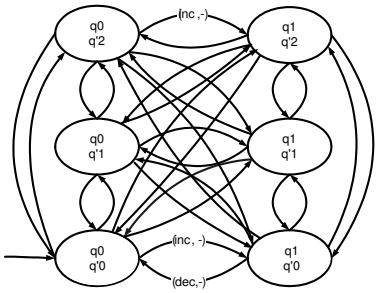


FIG. 4.3 – Produit cartésien d'un compteur modulo 2 et d'un compteur modulo 3

*** Produit synchronisé de systèmes de transitions**

Le produit cartésien autorise tous les comportements possibles. Pour restreindre ces comportements, Arnold et Nivat ont introduit dans [Arnold, 1992b; 1992a] la notion de produit synchronisé. Afin de décrire plus précisément les comportements souhaités, on explicite les transitions autorisées par un ensemble de synchronisations.

Définition : Ensemble de synchronisations
Soient E_1 et E_2 les ensembles d'étiquettes de deux systèmes de transitions S_1 et S_2. Un ensemble de synchronisations Syn est un sous-ensemble de $E_1 \otimes E_2$. Il indique parmi les éléments de $E_1 \otimes E_2$, les transitions qui participent à des comportements souhaités. Le produit synchronisé ne contient que les transitions qui figurent dans l'ensemble des transitions autorisées, aussi appelées transitions synchronisées.

Définition : Produit synchronisé
Soient $S_1 = \langle Q_1, Q_{01}, E_1, T_1, l_1 \rangle$ et $S_2 = \langle Q_2, Q_{02}, E_2, T_2, l_2 \rangle$ deux systèmes de transitions et Syn un ensemble de synchronisations, le produit synchronisé de S_1 et S_2, noté $S_1 \times_{Syn} S_2$ est défini par le cinq-uplet $S = \langle Q, Q_0, E, T, l \rangle$ comme suit :

- $Q \subseteq Q_1 \times Q_2$,
- $Q_0 \subseteq Q_{01} \times Q_{02}$,
- $E = Syn$,
- $T = Q \times E \times Q$ est définie ainsi :
  - $(q_1, q_2) \xrightarrow{(e_1,-)} (q'_1, q_2) \in T$ si $q_1 \xrightarrow{e_1} q'_1 \in T_1$ et $(e_1, -) \in Syn$
  - $(q_1, q_2) \xrightarrow{(-,e_2)} (q_1, q'_2) \in T$ si $q_2 \xrightarrow{e_2} q'_2 \in T_2$ et $(-, e_2) \in Syn$
  - $(q_1, q_2) \xrightarrow{(e_1,e_2)} (q'_1, q'_2) \in T$ si $q_1 \xrightarrow{e_1} q'_1 \in T_1 \wedge q_2 \xrightarrow{e_2} q'_2 \in T_2$ et $(e_1, e_2) \in Syn$
- $l((q_1, q_2)) = l_1(q_1) \cup l_2(q_2)$.

Par exemple, si l'on veut coupler fortement les deux compteurs modulo 2 et modulo 3, on définit comme ensemble de synchronisations :

$$Syn = \{(inc, inc), (dec, dec)\}$$

Le produit synchronisé est la donnée des diagrammes de transitions des composantes et de l'ensemble des synchronisations. Une exécution du produit synchronisé est une exécution du produit cartésien réduite aux transitions qui ont pour étiquette un élément de Syn.

*** Compositions définies à partir du produit synchronisé**

Plusieurs compositions peuvent être vues comme des instances particulières du produit synchronisé : le produit asynchrone, le produit synchrone, la composition basée sur des étiquettes de transitions communes ou encore des systèmes communicants (synchronisation par échange de messages).

- Produit asynchrone de deux systèmes de transitions

Le produit asynchrone est un produit synchronisé dans lequel seuls les comportements individuels des composants sont autorisés.

Définition. Soient $S_1 = \langle Q_1, Q_{01}, E_1, T_1, l_1 \rangle$ et $S_2 = \langle Q_2, Q_{02}, E_2, T_2, l_2 \rangle$ deux systèmes de transitions, l'ensemble de synchronisations Syn pour le produit asynchrone est défini ainsi :

$$\text{Syn} = (E_1 \times \{-\}) \cup (\{-\} \times E_2)$$

Le produit asynchrone de S_1 et S_2 est égal à $S_1 \times_{\text{Syn}} S_2$.

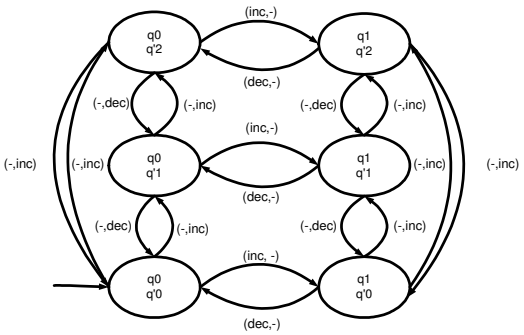


FIG. 4.4 – Produit asynchrone d'un compteur modulo 2 et d'un compteur modulo 3

Exemple : Le produit asynchrone d'un compteur modulo 2 avec un compteur modulo 3 est le produit synchronisé dont l'ensemble de synchronisations *Syn* est formé uniquement des transitions des deux composants ($\text{Syn} = \{(\text{inc}, -), (\text{dec}, -),$
$(-, \text{inc}), (-, \text{dec})\}$). Le résultat est illustré par la figure 4.4.

- Produit synchrone de deux systèmes de transitions

Le produit synchrone est aussi un produit synchronisé dans lequel uniquement les comportements synchronisés entre tous les composantes sont autorisés.

Définition. Soient $S_1 = \langle Q_1, Q_{01}, E_1, T_1, l_1 \rangle$ et $S_2 = \langle Q_2, Q_{02}, E_2, T_2, l_2 \rangle$ deux systèmes de transitions étiquetés, l'ensemble de synchronisations Syn pour le produit synchrone est défini ainsi : $\text{Syn} = E_1 \times E_2$.
Le produit synchrone de S_1 et S_2 est égal à $S_1 \times_{\text{Syn}} S_2$.

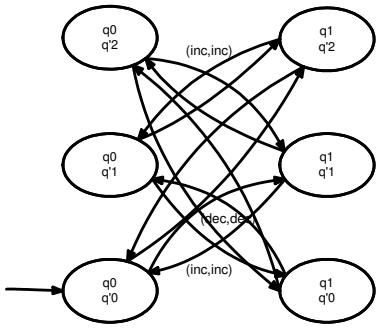


FIG. 4.5 – Produit synchrone d'un compteur modulo 2 et d'un compteur modulo 3

Exemple : Le produit synchrone d'un compteur modulo 2 avec un compteur modulo 3 est le produit synchronisé dont l'ensemble de synchronisations *Syn* est formé uniquement de toutes les transitions synchronisées des deux composants (Syn = $\{(\text{inc}, \text{inc}), (\text{dec}, \text{dec}), (\text{dec}, \text{inc}), (\text{inc}, \text{dec})\}$). Le résultat est illustré par la figure 4.5.

- Systèmes de transitions à étiquettes communes

Un cas particulier du produit synchronisé consiste à considérer deux composants $\text{S}_1 = \langle \text{Q}_1, \text{Q}_{01}, \text{E}_1, \text{T}_1, l_1 \rangle$ et $\text{S}_2 = \langle \text{Q}_2, \text{Q}_{02}, \text{E}_2, \text{T}_2, l_2 \rangle$ tels que $\text{E}_1 \cap \text{E}_2 \neq \varnothing$, puis à réaliser la synchronisation des transitions en utilisant ces étiquettes communes. Les transitions d'étiquettes communes auront lieu simultanément, les autres transitions auront lieu indépendamment les unes des autres. L'ensemble de synchronisations Syn, dans ce cas, est défini ainsi :

$$\begin{gathered}\text{Syn} = \{(\text{e}, \text{e}) | \text{e} \in \text{E}_1 \wedge \text{e} \in \text{E}_2\} \cup \\ \{(\text{e}_1, -) | \text{e}_1 \in \text{E}_1 \wedge \text{e}_1 \notin \text{E}_2\} \cup \\ \{(-, \text{e}_2) | \text{e}_2 \notin \text{E}_1 \wedge \text{e}_2 \in \text{E}_2\}.\end{gathered}$$

Synchronisation par échange de messages (systèmes de transitions communicants)

Un cas particulier du produit synchronisé est celui de la synchronisation par envoi/réception de messages. Le premier composant envoie, par exemple, un message m qui doit être reçu par le second composant. Parmi les étiquettes des transitions, on distingue celles correspondant à l'émission d'un message m, notée !m et celles correspondant à la réception de ce même message, notée ?m. Dans le produit synchronisé, on n'autorisera que les transitions où toute émission est accompagnée de la réception correspondante ou vice-versa : on parlera de synchronisation par échange de messages. Les autres transitions (celles ne correspondant ni à l'envoi, ni à la réception d'un message) ne sont pas synchronisées.

Définition. Soient $\text{S}_1 = \langle \text{Q}_1, \text{Q}_{01}, \text{E}_1, \text{T}_1, l_1 \rangle$ et $\text{S}_2 = \langle \text{Q}_2, \text{Q}_{02}, \text{E}_2, \text{T}_2, l_2 \rangle$ deux systèmes de transitions, l'ensemble de synchronisations Syn pour deux systèmes communicants par envoie de messages est défini ainsi :

- $\text{Syn} = \{(e_1, e_2) | e_1 \in E_1 \wedge e_2 \in E_2 \wedge ((e_1 = !m \wedge e_2 = ?m) \vee (e_1 = ?m \wedge e_2 = !m))\} \cup$
$\{(e_1, -) | e_1 \in E_1 \wedge e_1 \neq !m \wedge e_1 \neq ?m\} \cup$
$\{(-, e_2) | e_2 \in E_2 \wedge e_2 \neq !m \wedge e_2 \neq ?m\}$

Le système complet résultant de la synchronisation par échange de messages entre S_1 et S_2 est égal à $S_1 \times_{\text{Syn}} S_2$.

- Synchronisation par variables partagées

Il existe un autre moyen de faire communiquer entre elles les différentes composantes d'un système, en leur faisant partager un certain nombre de variables. Une même variable peut être utilisée et partagée par plusieurs composants. Un système ouvert *(open system)* est un système dans lequel certaines des variables peuvent être modifiées par un autre composant. On parle alors de composition par variables partagées. Un composant à variables partagées est classiquement appelé *module* [Kupferman and Vardi, 2000]. Dans un module M, l'ensemble des variables est partitionné en trois ensembles : l'ensemble des variables privées, l'ensemble des variables d'interface et l'ensemble des variables externes. Les variables privées ne peuvent ni être lues, ni être modifiées par un autre module. Les variables d'interface peuvent uniquement être lues par les autres modules et sont modifiées que par le module M. Les variables externes peuvent uniquement être lues par M, mais qui sont modifiées par d'autres modules.

L'ensemble des variables contrôlées par M est formé par les ensembles des variables privées et des variables d'interface. L'ensemble des variables observables de M depuis un autre module est formé par les ensembles des variables d'interface et des variables externes de M.

Dans ce qui suit, nous présentons les compositions réalisées autour de la méthode B.

4.3.2 Mécanisme de composition en B classique

La structure de base en B classique est la machine abstraite qu'on transforme (raffine) pour obtenir une implémentation. Une machine abstraite modélise un logiciel qui présente une interface d'accès sous forme d'opérations. Pour faciliter la modélisation de logiciels complexes, B fournit un certain nombre de clauses pour composer différentes machines.

Nous présentons dans ce qui suit, les différents types de relations entre machines abstraites selon trois niveaux :

- Un premier niveau d'inclusion correspond à une fonction de partage permettant uniquement de travailler sur les variables et les constantes des machines incluses. Les clauses **USES** et **SEES** permettent de désigner les machines extérieures sui seront utilisées en partageant des données entre plusieurs composants formels. La clause **SEES** ne permet d'appeler que des opérations de consultation (ne modifiant aucune variable). De plus, les invariants ne peuvent pas porter sur les variables vues. La clause **USES** ne permet pas d'appeler les opérations.

  Les clauses définies dans ce niveau de structuration permettent de réutiliser des ensembles et des variables dans la définition de propriétés, soit dans l'INVARIANT pour les variables issues de la clause **SEES**, soit dans l'INVARIANT et dans les préconditions pour les variables issues de la clause **USES**. Par contre, ces deux mécanismes de structuration ne permettent pas d'utiliser le résultat des opérations des machines inclues.

Les obligations de preuve pour une machine contenant une clause **SEES** sont celles relatives à la préservation de l'invariant, par l'initialisation et les opérations. Pour les appels d'opérations, la vérification des préconditions est prise en compte par ces obligations de preuve

- L'utilisation de la clause **INCLUDES**, correspondant à un certain niveau d'inclusion, permet de regrouper et d'utiliser les clauses SETS, CONSTANTS, PROPERTIES, INVARIANT et INITIALISATION de la machine inclue. En d'autres termes, l'inclusion d'une machine permet d'hériter des ensembles, les constantes et les variables d'une machine définis dans la machine incluse. Néanmoins les variables ainsi importées ne peuvent être réutilisées que dans les parties de substitution de la machine, c'est-à-dire dans l'INVARIANT et les préconditions. La modification des variables incluses ne peut être réalisée que par exécution de l'opération dans leur machine de définition, donc dans la machine inclue. Pour modifier ces variables, il est donc nécessaire de réaliser des appels à opérations. Pour inclure une opération, il est nécessaire de la promouvoir en utilisant la clause **PROMOTES**.

- Le dernier niveau d'inclusion par la clause **EXTENDS** représente la liste des machines qui seront totalement incorporées, c'est-à-dire que les constantes, les variables et les opérations, sont réutilisables dans la machine incluante. Cette opération correspond aux clauses **INCLUDES** et **PROMOTES** définies précédemment, et permet la réutilisation des résultats des opérations réalisées dans une autre machine.

Dans la composition de machines B, on ne s'intéresse pas seulement à la composition de spécifications, mais aussi à la composition des preuves d'invariant et des preuves de raffinement. Le langage B offre différents principes d'assemblage. Ces clauses d'assemblage vont devoir permettre, en plus de composer du texte, de composer les preuves. Pour ce faire, Marie-Laure Potet [Potet, 2002] a fait une étude approfondie de diverses notions de composition, elle s'est intéressée plus précisément aux obligations de preuve dans un contexte de développement s'appuyant sur la décomposition "horizontale" et le raffinement. Elle a présenté, à travers des exemples, les différentes clauses d'assemblage de la méthode B et la manière de les utiliser. Elle a proposé des formes de structuration pour développer certaines formes d'architecture, tout en respectant les restrictions imposées. Néanmoins, ils restent difficiles à mettre en œuvre, en raison des restrictions nécessaires qui leur sont associées. En particulier, l'architecture doit être conçue de manière à respecter ces contraintes. Celles-ci étant en grande partie liées aux restrictions sur les invariants, il s'ensuit que les architectures dépendant fortement des propriétés à valider.

En effet, la méthode B offre un bon niveau de compositionalité que ce soit au niveau de la conception que le processus de validation par la preuve. La restriction la plus pénalisante semble être le principe un écrivain/plusieurs lecteurs, qui impose de construire des architectures dans les quelles les modifications attachées à un jeu de variables sont localisées dans un composant.

4.3.3 Travaux existants autour de la composition en B événementiel

Certains travaux se sont intéressés à la composition dans le cadre de B, afin de pouvoir spécifier une machine ou un système d'événements B sous la forme de machines ou de systèmes d'événements composants. Dans cette section, nous nous limitons à l'analyse des travaux de Abrial, Julliand et Treharne partageant le même cadre d'étude de notre travail.

*** Composition proposée par Abrial**

Abrial [Metayer *et al.*, 2005; Rodin Partners, 2006] a proposé la décomposition-composition combinée au raffinement comme outil pour traiter des problèmes complexes. Il spécifie un système d'événements B, puis réalise sa décomposition en plusieurs sous-systèmes, les raffine et prouve indépendamment chacun des sous-systèmes obtenus et déduit la correction du système complet en faisant l'hypothèse de la correction des sous-systèmes.

L'approche de décomposition d'Abrial est proche du principe de décomposition proposé par Lamport [Abadi and Lamport, 1995] (présenté chapitre 3) qui permet de raffiner un sous-composant en tenant compte des propriétés des autres sous composants (constituant l'environnement). En effet, les spécifications doivent être exprimées sous une forme *rely-garantee*, afin de préciser les interférences possibles entre les composants et leur environnement.

Décomposition. La décomposition consiste à former plusieurs modèles à partir des éléments d'un seul, elle induit une partition des variables du système initial, certaines variables étant partagées entre certains sous-systèmes. Toutefois la communication entre les deux modèles va se faire à l'aide de variables partagées (modifiées dans un module et consultées dans l'autre). Les variables externes "consultées" sont déclarées *variables externes* (external variables dans le projet RODIN [Rodin Partners, 2006]). Les variables externes "consultées" nous amènent à introduire dans les sous-systèmes des événements externes qui simulent la façon dont les variables partagées sont modifiées dans les autres sous-systèmes. Ces événements dépendent seulement des variables externes. En d'autres termes, leurs gardes et substitutions généralisées n'utilisent pas des variables internes. Les événements externes sont raffinés aussi en événements externes. Lors du raffinement des sous systèmes, les variables partagées doivent être raffinées de façon cohérente.

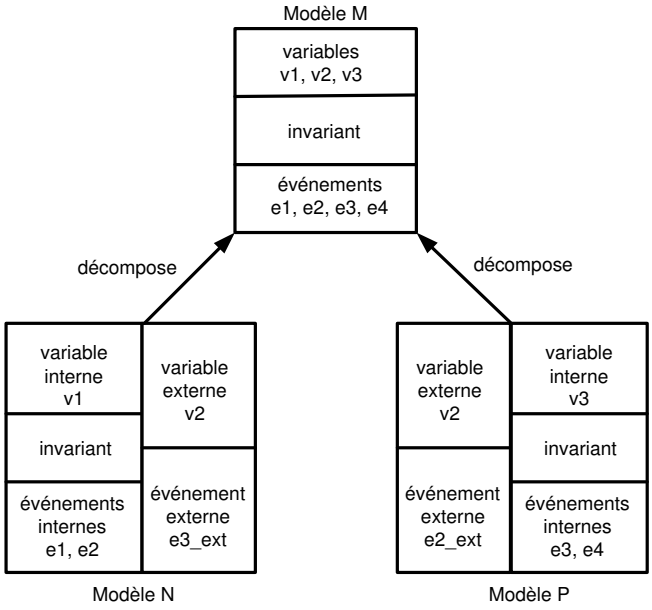


FIG. 4.6 – Décomposition.

La figure 4.6, montre un exemple de décomposition d'un modèle M en deux sous modèles N

et P. Cette décomposition est réalisée en partitionnant les événements de M en deux groupes : ceux de N et ceux de P. De même, les variables sont partagées en deux groupes : ceux de N et ceux de p. La variable v_2 est externe et les événements e_3_ext dans N et e_2_ext dans P sont externes.

Raffinement et recomposition. La recomposition des sous-systèmes consiste à faire la conjonction des invariants des sous-systèmes, l'union des variables et à supprimer les événements externes. Les sous-systèmes ayant été prouvés, la preuve de raffinement du modèle recomposé s'effectue en s'acquittant d'une obligation qui porte sur les événements externes.

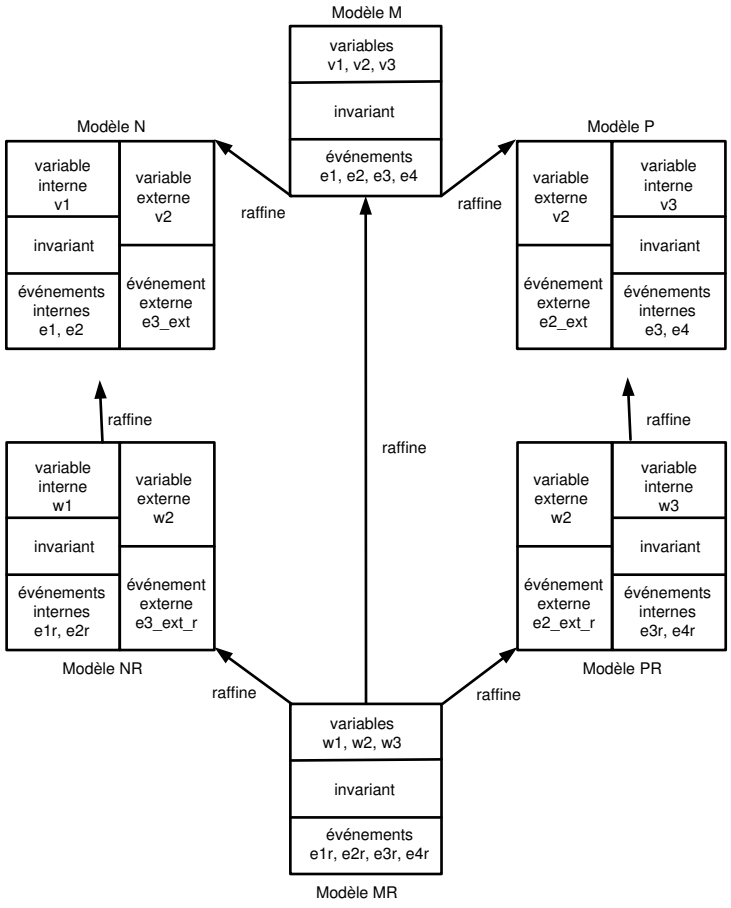


FIG. 4.7 – Recomposition.

Dans la figure 4.7, le modèle N est raffiné en un modèle NR avec variables internes w_1 et variables externes w_2. Le modèle P est raffiné en un modèle PR avec variables internes w_3 et variables externes w_2. Toutefois, il y a des contraintes qui doivent être prises en compte : les variables partagées v_2 doivent être raffinées de la même manière. Ceci est réalisé en ayant le même invariant de collage dans les deux modèles. Par la suite, il est possible de recomposer les modèles NR et PR pour former le modèle MR en faisant la conjonction des invariants des deux modèles et en supprimant les événements externes. Pour prouver que le modèle recomposé MR est un raffinement du modèle original M, il suffit de prouver que les événements externes e_3_ext dans N et e_2_ext dans P sont raffinés en événements e_3 et e_2 dans M.

*** Composition proposée par Bellegarde et Julliand**

Dans leur travaux [Bellegarde *et al.*, 2002], les auteurs ont étudié la composition des systèmes autonomes fermés (systèmes d'événements B) et le raffinement associé. Ils ont donné une interprétation de la composition en termes de composition de systèmes de transitions étiquetées. Ils ont étudié de manière compositionnelle le raffinement de ces systèmes et la préservation des propriétés par le raffinement.

Ils ont proposé de spécifier des composants réactifs par des systèmes d'événements B et leur interaction comme suit :

- un couple de noms d'événements indique la synchronisation entre deux événements appartenant à deux composants différents.
- La synchronisation entre les composants est réalisée en contraignant l'activation des événements d'un composant par un prédicat sur les variables d'état de l'autre composant.

La composition (parallèle synchronisée) correspond à un produit de systèmes de transitions auquel s'appliquent des contraintes qui portent sur les transitions. Ces contraintes sont exprimées par un ensemble de conditions associées à des événements.

En effet, les auteurs ont proposé une extension du produit synchronisé d'Arnold et Nivat : les systèmes à composants synchronisés. Les différents composants sont donnés par des systèmes de transitions étiquetés et les interactions entre composants sont, quant-à-elles, exprimées par un ensemble de synchronisations. Ils ont contraints l'activation des transitions synchronisées par des conditions d'activations.

Les auteurs définissent une composition qui, à partir de deux systèmes à événements B et d'informations complémentaires, permet de construire un nouveau système. Les variables du nouveau système sont formées de l'union des variables des composants après renommage si nécessaire, l'invariant est la conjonction de ceux des composants, l'initialisation est la composition parallèle des substitutions d'initialisation. Les événements du nouveau système sont, soit des événements d'un des composants avec éventuellement une garde renforcée, soit un événement défini comme la composition parallèle d'un événement d'un composant et d'un événement de l'autre composant, cette composition étant éventuellement gardée. Les informations qui permettent de guider la composition constituent la spécification de synchronisation, c'est une liste dont les éléments sont, soit des noms ou des couples de noms d'événements auxquels sont éventuellement associés un prédicat qui définit le renforcement de la garde.

Exemple de spécification de synchronisation :
DS = { E_{A1} when $V_{A1} = vide$, E_{A2} when $V_{A2} = stopped$, (E_{A3}, E_{B3}) when $V_{A4} = on \wedge V_{B4} = off$, E_{A5}, E_{B5},... }

Ces travaux ont été, pendant un certain temps pour nous, une source d'inspiration. Nous avons été intéressés par l'expression de contraintes associées à l'opération de composition, mais nous avons pris conscience que notre problème était différent car il se pose à un niveau de décomposition plus fin qui consiste, pour une machine, à séparer ce qui est contrôle de ce qui est contrôlé. Il s'ensuit une dissymétrie entre ces deux composants et l'expression des contraintes de composition n'a pas tout à fait la même finalité. Une autre différence réside dans le fait que la composition n'est pas parallèle : le contrôleur et le contrôlé évoluant dans une boucle de contrôle.

*** Composition proposée par Treharne et Schneider**

Avec CSP || B, H. Treharne et S-A. Schneider [Schneider and Treharne, 2002] proposent une modélisation qui allie l'algèbre de processus CSP [Butler, 2000] et le B classique pour contrôler

les interactions entre plusieurs machines B. L'algèbre de processus *CSP* est un langage de haut niveau pour décrire les différents processus d'un système concurrent, ainsi que leurs interactions sous forme d'échanges de messages. Les processus peuvent être synchronisés et la communication s'effectue par l'intermédiaire de canaux de communication. Dans [Schneider and Treharne, 2002] un sous-ensemble du langage *CSP* est utilisé pour décrire des contrôleurs pour chacune des machines B.

A chaque machine MA_i du modèle, est associé un contrôleur CSP C_i. La combinaison d'une machine B et du contrôleur correspondant notée $MA_i||C_i$ forme une machine contrôlée. Le système automatisé est constitué de machines contrôlées qui communiquent au travers des contrôleurs *CSP*. Ce modèle qui sépare le contrôle (CSP) des actions (les opérations de B) convient bien au type de problème qui nous intéresse.

Nous avons préféré néanmoins utiliser un formalisme unique B événementiel pour modéliser les deux composants en étant bien conscient de la différence de nature qui existe entre contrôleur et contrôlé.

4.3.4 Conclusion

Le raffinement et la composition sont deux techniques importantes pour le développement de systèmes complexes et peuvent être des paradigmes de spécification et de vérification pour s'assurer de la correction des systèmes. Le raffinement est un processus de développement et de vérification vertical allant de l'abstrait vers le concret en introduisant les détails d'un cahier des charges. La composition est un processus de développement horizontal qui permet la réutilisation, elle facilite les preuves de propriétés. La composition et le raffinement correspondent à deux moyens pour construire des modèles et prouver des propriétés d'une spécification.

Nous avons introduit dans ce chapitre, des opérateurs de composition classiques : le produit cartésien, puis le produit synchronisé. La majorité des compositions pouvaient être vues comme des extensions du produit synchronisé, qu'il s'agisse du produit synchrone, du produit asynchrone, des compositions par étiquettes communes ou des synchronisations par échange de messages et de la composition par variables partagées.

La décomposition est le processus qui consiste à exhiber les différentes parties d'un système et leur assemblage. L'aspect méthodologique est de savoir comment décomposer au vue des mécanismes offerts par l'approche et des objectifs visés. Il faut, par exemple, étudier les classes de propriétés et les classes de systèmes pour lesquels il est possible de proposer des décompositions efficaces qui permettent de maîtriser la complexité, en particulier vis-à-vis des outils.

Dans les travaux que nous avons présentés, les composants des systèmes sont de même nature ; de ce fait, nous ne pouvons pas utiliser leurs approches. Dans les systèmes automatisés, les composants (contrôleur et contrôlé) ne sont pas de nature symétrique ; il s'ensuit que l'opération de composition a un caractère particulier.

Enfin, notons que nous ne nous intéressons pas, dans notre travail, à l'intérêt de la composition pour la preuve mais plutôt à la vérification de l'interaction du système à contrôler avec le système de contrôle avant sa mise en oeuvre et à faire la preuve sur le système automatisé.

Chapitre 5

B événementiel et les propriétés de vivacité

5.1 Introduction

Dans ce chapitre, nous nous intéressons au développement formel de systèmes de contrôle commande qui sont une classe de systèmes réactifs. Les propriétés qui nous intéressent sont les propriétés d'invariance et de vivacité.

Notre travail s'appuie principalement sur la méthode de développement B événementiel et sur les outils associés de l'atelier B pour spécifier, développer et valider des systèmes. Comme B ne permet de développer que des systèmes devant satisfaire des propriétés d'invariance, nous avons eu recours au langage de modélisation TLA$^+$ pour la vérification des propriétés de vivacité. Ce langage de modélisation permet un développement par raffinement, et se base sur la notion d'actions qui s'apparente à celle d'événements ce qui la rend compatible avec B (les deux langages se basent sur la notion de systèmes de transition).

Nous proposons dans ce chapitre, une approche qui combine le B et TLA$^+$ pour exprimer et vérifier des propriétés d'invariance et de vivacité. Nous profitons ainsi de l'environnement de spécification de B et de l'outil de preuve associé pour vérifier des propriétés d'invariance, et de TLA$^+$ et son environnement pour vérifier les propriétés de vivacité qui ne peuvent être facilement exprimées et vérifiées en B. Nous proposons dans un premier temps de définir une extension du B événementiel permettant d'exprimer des propriétés de vivacité. Nous donnons ensuite une nouvelle définition sémantique de B événementiel pour exprimer la sémantique de l'extension. Par la suite, nous proposons des règles de preuve pour la vérification de telles propriétés dans l'axiomatique de B. Dans un second temps, nous présentons un prototype que nous avons développé, appelé B2TLA$^+$, permettant la transformation d'un modèle B en un module TLA$^+$. Enfin, nous proposons de vérifier ces propriétés en utilisant le prouveur de théorèmes Isabelle/TLA [Merz, 1999] ou le model checker TLC, après avoir transformé le modèle B temporel en un module TLA$^+$.

Nous présentons dans la section 2 l'approche proposée utilisant conjointement le B événementiel et le langage TLA$^+$. Dans la section 3, nous présentons le prototype (B2TLA$^+$) que nous avons développé pour transformer un modèle B étendu en un module TLA$^+$. Dans la section 4, nous utilisons le prouveur de théorèmes Isabelle/TLA pour la vérification des propriétés de vivacité et nous illustrons notre méthode sur le cas d'un système simple modélisant un time_out. Dans la section 5, nous utilisons le model checker TLC pour la vérification des propriétés de

vivacité sur des systèmes finis et nous illustrons notre méthode sur le système d'un tri de paquets postaux. Dans la section 6, nous utilisons les diagrammes de prédicat, qui sont un formalisme pour la vérification des systèmes à états infinis en se focalisant sur la preuve des propriétés de vivacité puisque la technique de model checking est utilisée pour la vérification des propriétés sur des systèmes finis.

Nous appellerons par la suite, B temporel l'extension de B événementiel tandis que B désignera toute référence à B événementiel. Le modèle muni d'une extension et à chaque fois qu'on fait référence à B, il s'agit du B événementiel.

5.2 Comparaison du B événementiel et du langage TLA$^+$

Dans ce qui suit, nous donnons les différences entre des spécifications en B événementiel et en TLA$^+$.

- Une spécification TLA$^+$ est une formule temporelle, alors que le B ne permet pas d'exprimer des formules temporelles.
- A la différence de B, TLA$^+$ n'est pas typé. La correction de type des variables d'une spécification TLA$^+$ *Spec* est une propriété d'invariance affirmant que tout état atteint pendant toute exécution satisfaisant *Spec*, une variable d'état est un élément d'un ensemble approprié (son type). Nous trouvons le type erreurs en vérifiant cette propriété d'invariance. Le fait qu'il est non typé, TLA$^+$ est plus expressif que B.
- TLA$^+$ et B manipulent les ensembles et les fonctions mais ils utilisent différents opérateurs pour les décrire. Les relations sont des constructions B qu'on ne retrouve pas dans TLA$^+$. Une spécification B utilise des tuples et des fonctions qui sont des relations particulières et on peut y distinguer les fonctions, les injections, les surjections et les bijections (qui peuvent être totales ou partielles) tandis qu'une spécification TLA$^+$ ne dispose que des enregistrements et des fonctions totales.
- TLA$^+$ peut être utilisé pour spécifier des propriétés de sûreté et de vivacité. B ne spécifie que des propriétés de sûreté.

Comparaison qualitative

Dans le tableau ci-dessous, nous donnons une comparaison qualitative de ces deux méthodes. Nous avons choisi un ensemble d'attributs qui décrit les deux méthodes de spécification. Nous donnons la définition de ces attributs, puis nous évaluons ces attributs pour les deux méthodes. Les résultats sont décrits dans le tableau de la figure 5.1.

Attributs d'évaluation des deux méthodes
Modèle sous-jacent : Cet attribut caractérise comment la notation de la spécification décrit le comportment du système. Les deux méthodes utilisent le modèle *transition d'état* où la spécification décrit une relation de transition sur un ensemble d'états. Les transitions sont étiquetées par des événements.

Concurrence : Une notation permet la modélisation de la concurrence si le système peut être décrit en termes de processus qui communiquent.

La concurrence en B correspond à l'exécution simultanée (substitution parallèle) de deux substitutions tandis qu'en TLA$^+$ elle correspond à la composition parallèle en utilisant la conjonction logique $\wedge$.

non-déterminisme : Cet attribut détermine si la notation permet le non-déterminisme.

En B, le non-déterminisme est réalisé par la substitution généralisée choix non borné alors qu'en TLA$^+$, il est réalisé par l'action qui commence par un $\exists$.

Preuve : Cet attribut détermine si les propriétés sur les spécifications peuvent être vérifiées en utilisant un système de preuve.

Model checking : Cet attribut détermine si les propriétés sur les spécifications peuvent être vérifiées en énumérant les états du système.

Le tableau fournit une description de chaque méthode en respectant les attributs définis précédemment.

Méthode	Modèle sous-jacent	Concurrence	Non-determinisme	Preuve	Model checking Propriétés d'invariance	Model checking Propriétés de vivacité
B événementiel	transition d'état	oui	oui	oui, outil (Click_n_Prove)	oui, outil (ProB)	non
TLA+	transition d'état	oui	oui	oui (Isabelle/TLA)	oui, outil (TLC)	oui, outil (TLC)

FIG. 5.1 – Comparaison du B événementiel et du langage TLA$^+$

5.3 Approche proposée

Dans cette section, nous proposons une approche pour la vérification des propriétés de vivacité à partir d'un modèle B temporel. La sémantique de B est basée sur la notion de transformateurs de prédicats et son extension temporelle requiert une redéfinition d'une sémantique fondée sur les traces d'exécution. Pour ce faire, nous proposons :

- L'extension du modèle B qui présente :
  - un cadre syntaxique des modèles B engendré par l'ajout des clauses décrivant des propriétés temporelles (d'équité et de fatalité),
  - un cadre sémantique basé sur les traces d'exécution,
  - des règles de preuve pour la vérification des propriétés temporelles dans l'axiomatique de B,
- La transformation du modèle B temporel en un module TLA$^+$ qui comporte :
  - la définition des règles de transformation d'un modèle B temporel en un module TLA$^+$,
  - le développement du prototype B2TLA$^+$ permettant la traduction d'un modèle B temporel en un module TLA$^+$,

  - la vérification des propriétés de vivacité en utilisant un outil associé à TLA$^+$ (prouveur de théorèmes Isabelle/TLA ou le model checker TLC),
- L'utilisation des diagrammes de prédicat et l'outil associé DIXIT pour la vérification des propriétés de vivacité sur des systèmes à états infinis puisque la technique de model checking est utilisée pour des systèmes à états finis.

5.3.1 Présentation de l'approche

Dans cette section, nous proposons d'intégrer l'aspect temporel dans le B événementiel sans changer sa notation [Méry, 1986; 1987]. Notre approche intéresse des utilisateurs familiers de B et veulent toutefois exprimer et vérifier des propriétés de vivacité. Nous donnons la possibilité à l'utilisateur d'exprimer ces propriétés de vivacité en restant dans le formalisme de B. Pour la vérification de ce type de propriétés, nous donnons des règles de preuve dans l'axiomatique de B. Néanmoins, nous préférons nous placer dans un environnement de vérification pour ce type de propriétés. C'est pourquoi, nous nous sommes intéressés à TLA$^+$ à cause de sa compatibilité avec B. Nous utilisons TLA$^+$ pour la vérification des propriétés de vivacité parce que les utilisateurs sont des connaisseurs de B.

Nous proposons ainsi de regrouper le B événementiel et le langage TLA$^+$ dans une approche qui permet de spécifier et de vérifier des propriétés de sûreté, de fatalité et d'équité. B et TLA$^+$ ont des syntaxes différentes mais ils sont fondés sur la notion des systèmes de transition et le raffinement ce qui font d'elles deux méthodes compatibles. Un module TLA$^+$ et un modèle B comportent des éléments compatibles à l'exception des aspects liés aux propriétés sur les traces qui ne peuvent pas être prises en compte dans B c'est la raison pour laquelle nous proposons d'étendre le modèle B et de réexprimer la sémantique de B en terme de traces qui est le concept sous-jacent à la sémantique de TLA$^+$. Ainsi, nous pourrons transformer le modèle B temporel en un module TLA$^+$ pour la vérification de ce type de propriétés et utiliser le prouveur Isabelle/TLA ou le model checker TLC.

La méthode que nous proposons (Figure 5.2) consiste à :

1. Modéliser le système avec B en ne considérant que les propriétés de sûreté et d'invariance,
2. Vérifier les propriétés d'invariance par Clic_n_prove,
3. Ajouter dans le modèle B des propriétés de vivacité. La clause FAIRNESS spécifie les propriétés d'équité et la clause EVENTUALITY spécifie les propriétés de fatalité. Le modèle obtenu est un modèle B temporel. La syntaxe de ce modèle est donnée par la figure 5.1,
4. Transformer le modèle B temporel obtenu en un module TLA$^+$ en utilisant un prototype (B2TLA$^+$) que nous avons développé,
5. Vérifier des propriétés de vivacité en utilisant le prouveur de théorèmes Isabelle/TLA, ou le model checker TLC,
6. Utiliser les diagrammes de prédicats pour la vérification des propriétés de vivacité sur des systèmes infinis et aussi pour la vérification du raffinement des conditions d'équité,
7. Raffiner le modèle et itérer à partir de l'étape 2 tant que toutes les propriétés ne sont pas satisfaites. Le raffinement est donné par la figure 5.3.

Notons que nous avons des utilisateurs qui veulent rester dans le cadre de B et exprimer néanmoins des propriétés de vivacité. La transformation en un module TLA$^+$ est réalisée automatiquement par le prototype B2TLA$^+$. La vérification des propriétés de vivacité est alors réalisée en utilisant soit le prouveur Isabelle/TLA, soit le model checker TLC.

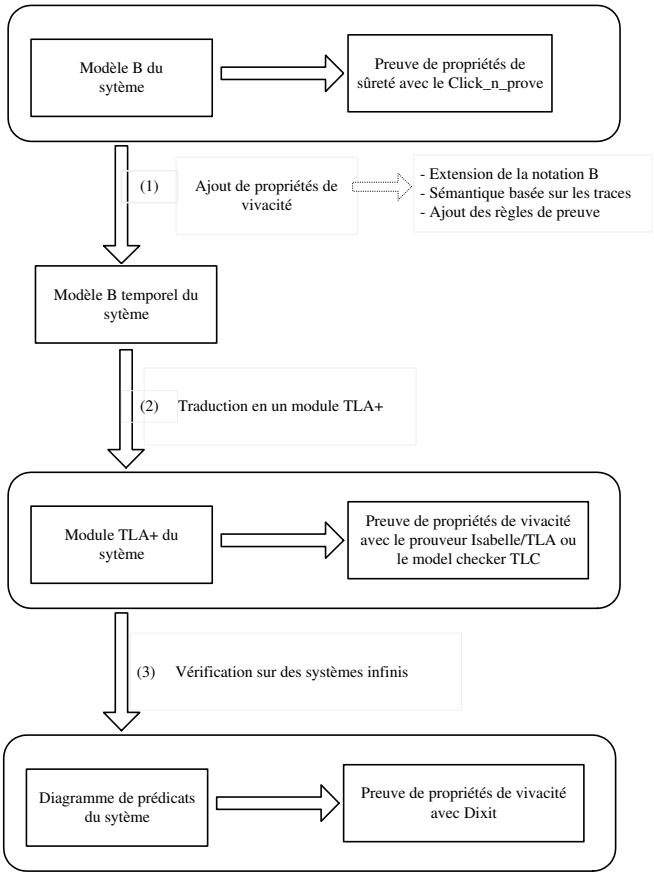


FIG. 5.2 – Traduction d'un modèle B vers un module TLA$^+$.

Dans ce qui suit, nous nous focalisons sur les points (1) et (2) de la figure 5.3. Le premier point concerne l'extension de la notation B par des propriétés de vivacité. Le deuxième point concerne la traduction du modèle B temporel en un module TLA$^+$. Le premier point traite d'une part, la syntaxique et la sémantique de l'extension pour exprimer des propriétés de vivacité et d'autre part, l'ajout des règles de preuve dans l'axiomatique de B pour la vérification de ce type de propriétés. Pour le deuxième point, nous définissons des règles de transformation d'un modèle B temporel vers un module TLA$^+$ et nous présentons le prototype B2TLA$^+$ que nous avons développé.

5.3.2 Syntaxe et sémantique de l'extension

Dans cette section, nous définissons une extension de la syntaxe du B événementiel et nous en définissons la sémantique en terme de traces d'exécution. L'extension temporelle porte essentiellement sur l'ajout de l'opérateur temporel *"Leads to"* inspiré de celui de la logique TLA. D'autre part, la preuve des propriétés de vivacité nécessite des hypothèses d'équité, faible et forte, représentées respectivement par *WF(e)* et *SF(e)*, où e est un événement B. Dans ce qui suit, nous donnons la syntaxe de l'extension, puis nous donnons sa sémantique basée sur les traces de la même manière qu'en TLA$^+$.

MODEL ⟨*nom*⟩
SETS ⟨*ensembles*⟩
CONSTANTS ⟨*constantes*⟩
VARIABLES ⟨*variables*⟩
INVARIANT ⟨*invariants*⟩
INITIALISATION ⟨*initialization de variables*⟩
EVENTS ⟨*événements*⟩
FAIRNESS ⟨***propriétés d'équité***⟩
EVENTUALITY ⟨***propriétés de fatalité***⟩
END

TAB. 5.1 – Extension temporelle de B.

*** Cadre syntaxique**

Avant de définir la syntaxe des formules qui étendent l'expressivité de B, nous commençons par quelques préliminaires.

- Variables rigides et d'états. L'état d'un système se compose d'un ensemble dénombrable des variables flexibles ou d'états *(V)*. Soit X l'ensemble dénombrable de variables rigides. Ces variables ne sont pas modifiées par des transitions du programme et gardent la valeur initiale durant toute l'exécution (ce sont des constantes logiques). Un état est une valuation des variables flexibles.

- Termes et états. un terme t est défini récursivement comme suit :
$t ::= c \mid x \mid f(\mathrm{t}_1,...,\mathrm{t}_n)$ où c est une constante, x est une variable $(x \in [V \cup X])$, $\mathrm{t}_1,...,\mathrm{t}_n$ sont des termes et f est un symbole de fonction d'arité n.

- Propositions atomiques. Une proposition atomique ap est une formule de la forme :
$ap ::= p(\mathrm{t}_1,...,\mathrm{t}_n)$ où p est un symbole de prédicat d'arité n et $\mathrm{t}_1,...,\mathrm{t}_n$ sont des termes.

- Prédicats d'états. Soit $\mathrm{SP}_{V \cup X}$, l'ensemble des prédicats d'états. Un prédicat d'état sp est une proposition définie par la grammaire suivante sp ::= ap | ¬sp | sp∨sp | sp∧sp | sp ⇒ sp | sp ⇔ sp | ∃x sp | ∀x sp.

- Propriétés de sûreté. Les propriétés de sûreté sont des formules de la forme :
F ::= □sp | □(sp ⇒ □sp) , où sp est un prédicat d'état.

Nous présentons dans ce qui suit notre extension concernant les formules de transition et de vivacité que nous ajoutons dans un modèle B.

- Formules de Transition. Une formule de transition décrit les transitions d'états. Une formule de transition ac est une formule de la forme :
ac ::= GS(e) | [e]sp | ⟨e⟩sp
où sp est un prédicat d'état, e est un événement et GS(e) est la substitution généralisée de l'événement e. [e]sp est la notation ensembliste de la plus faible précondition et ⟨e⟩sp est la plus faible précondition conjuguée.

- Propriétés de vivacité. Les propriétés de vivacité (fatalité et équité) sont des formules définies comme suit :

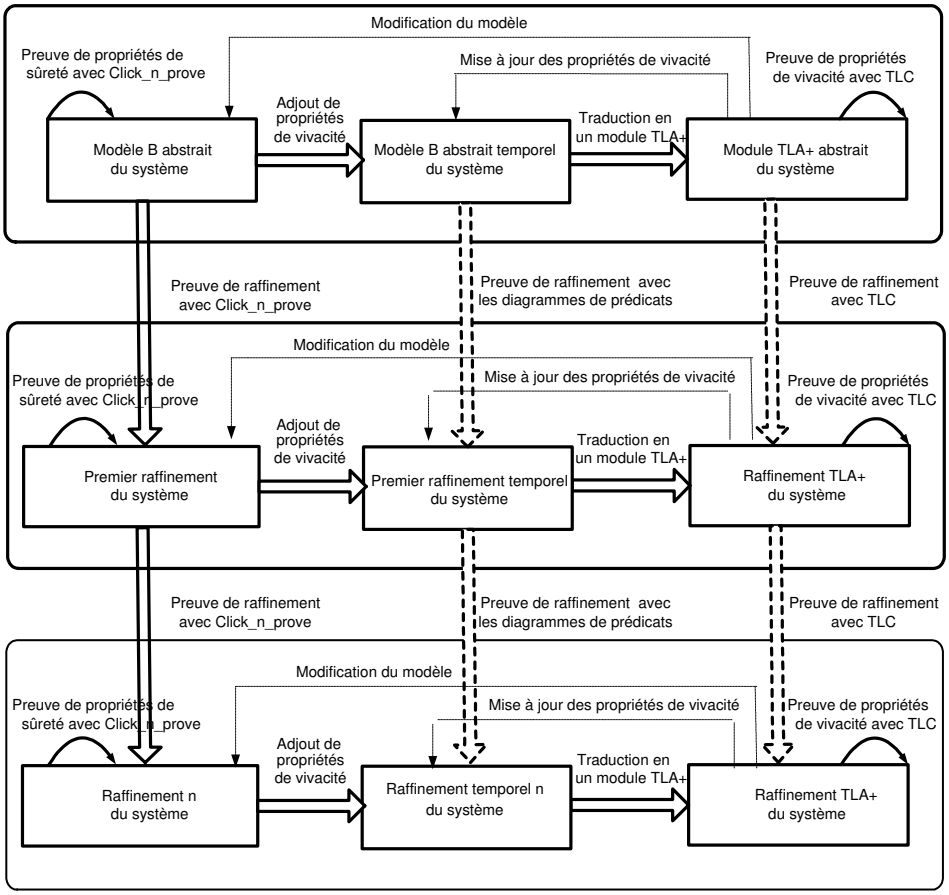


FIG. 5.3 – Raffinement pour le développement des systèmes automatisés.

- Les propriétés de fatalité sont exprimées par des formules de la forme :

  F $\rightsquigarrow$ G (F *leads to* G) définies par $\Box(F \Rightarrow \Diamond G)$ et signifient que chaque F sera suivi par G, où F et G sont des formules de la forme : F ::= sp | $\Diamond$F | $\Box$F | WF(e)|SF(e). où sp est un prédicat d'état, WF(e) et SF(e) sont respectivement les équités faible et forte de l'événement e.

  Ces propriétés sont contenues dans la clause EVENTUALITY.

- Les propriétés d'équité sont exprimées par des formules de la forme :
  - Pour l'équité faible d'un événement e :
    WF(e) est définie par $\Diamond\Box grd(e) \Rightarrow \Box\Diamond GS(e)$ et signifie que si un événement est continûment autorisé, alors il est infiniment souvent activé.
  - Pour l'équité forte d'un événement e :
    SF(e) est définie par $\Box\Diamond grd(e) \Rightarrow \Box\Diamond GS(e)$ et signifie que si un événement est autorisé infiniment souvent, alors il est infiniment souvent activé.

  où e est un événement, grd(e) est la garde de l'événement e (prédicat d'état) et GS(e) est la substitution généralisée (formule de transition).

  Ces propriétés sont contenues dans la clause FAIRNESS.

Comme en TLA$^+$, nous considérons trois types de propriétés :

- *Les propriétés d'états* qui dénotent les propriétés sur les états du système S (interprétées sur les états), que nous appelons les prédicats d'états,
- *Les propriétés relationnelles* qui dénotent les relations entre les paires d'états, que nous appelons les formules de transition,
- *Les propriétés temporelles* qui dénotent des propriétés définies sur les traces et utilisent des propriétés d'états, des propriétés relationnelles et des opérateurs temporels ($\Box$, $\Diamond$, $\rightsquigarrow$, ...).

*** Cadre sémantique**

L'extension de B pour exprimer des propriétés temporelles nécessite une redéfinition de la sémantique associée au langage. La sémantique "classique" de B traite principalement des substitutions qui réalisent des transitions d'états. Elle s'appuie sur la notion de transformateurs de prédicats. L'extension temporelle de B nécessite une sémantique adaptée s'appuyant sur les traces d'exécution, comme c'est le cas pour TLA. Pour être compatible avec TLA nous redéfinissons la sémantique des substitutions.

Dans notre extension, l'ensemble d'événements sont vus comme une relation entre les variables primées et non primées et nous utilisons ce point comme base pour définir la sémantique de l'extension de B.

Un système S est modélisé par un ensemble d'événements déclenchant des actions, quand leurs gardes sont vraies. Un événement e est défini par une garde dénotée grd(e) et par une relation sur un ensemble de variables flexibles (V) notée par GS(e) (relation indiquant la transformation des variables).

Nous interprétons toutes les propriétés en terme de traces d'états. Pour ce faire, nous introduisons un certain nombre d'éléments pour nous permettre de décrire la sémantique :

- V est l'ensemble de variables d'états (flexibles) du système S ; v est une variable d'état, v' désigne cette la variable v après l'exécution d'un événement. Primed_Var(S) = $\{v' \mid v \in V\}$ et Unprimed_Var(S) = $\{v \mid v \in V\}$.

- Val est l'ensemble des valeurs des variables du système S.

- States(S) est l'ensemble des états du système S et un état s_i de S est une fonction de V dans Val :
$$s_i : V \rightarrow Val.$$
- Une interprétation des prédicats d'états sur les états du système S est définie comme suit :
$s_i, \xi \models sp$ signifie que le prédicat d'état sp est satisfait à l'état s_i,
où ξ est une valuation des variables rigides de S.
- Init(S) désigne les valeurs initiales des variables flexibles du système S.

- Events(S) désigne l'ensemble des événements du système S. Un événement e est défini comme suit :
$$e \triangleq (grd(e), GS(e))$$
Où :

  - grd(e), est la garde associée à l'événement e. C'est un prédicat d'état qui s'évalue à vrai dans un état courant s_i ssi il est possible de faire un pas d'exécution à partir de cet état.

  - GS(e) est la substitution généralisée associée à l'événement e, correspondant à une transition à partir de l'état courant qui définit une modification des variables d'état.

- $BA_e(x, x')$ est le prédicat avant-après associé à l'événement e (où x correspondent aux valeurs courantes des variables d'état et x' les valeurs après obtenues après l'exécution de e). C'est une formule de la logique du premier ordre construite à partir des constantes ainsi que des variables primées et non primées.

- $\xrightarrow{e}$ est une relation sur l'ensemble des états du système S traduisant un pas d'exécution.

- Next(S) est une formule construite sur les variables primées et non primées du système S correspondant à la relation sur States(S). Next est de la forme suivante :

$$\text{Next} \triangleq R(e_1)(x, x') \vee ... \vee R(e_n)(x, x').$$

  Où $R(e_i)(x, x')$ est une relation TLA correspondant à une des trois formes des événements B.

  - $P(x, x')$
  - $G(x) \wedge P(x, x'))$
  - $\exists t.(G(t, x) \wedge P(x, x', t))$

  Où :

  - $P(x, x')$ est le prédicat avant après correspondant à l'événement simple,
  - $G(x) \wedge P(x, x')$ est le prédicat avant après correspondant à l'événement gardé,
  - $\exists t.(G(t, x) \wedge P(x, x', t))$ est le prédicat avant après correspondant à l'événement indéterministe.

  Nous avons : $(s_i, s_{i+1}), \xi \models \text{Next(S)} \triangleq s_i \rightarrow s_{i+1}$. La validité de Next(S) est définie sur les paires d'états et la propriété définie sur les paires d'états est nommée relation.
  Les formules $BA_e(x, x')$ et Next expriment des relations entre les variables primées et non primées ; de ce fait il y a une équivalence sémantique entre $BA_e(x, x')$ et Next, ce qui implique que $BA_e(x, x')$ est interprété par la formule Next en TLA.

- Invariants(S) est un ensemble de propriétés sur States(S) invariant pour S. $\varphi \in$ Invariants(S), si

  1. $\text{Init(S)} \Rightarrow \varphi$
  2. $\forall\, s_0, s_i \in \text{States(S)} \;:\; (s_0, \xi \models \text{Init(S)} \wedge (s_0 \rightarrow^* s_i)) \Rightarrow s_i, \xi \models \varphi$

- Traces(S) est l'ensemble des traces générées à partir de Init(S) utilisant $\rightarrow$. Une trace est une suite d'états notée par $\sigma = s_0 s_1 ..... s_i .....$ et satisfait les contraintes suivantes :

  1. $s_0, \xi \models \text{Init(S)}$ (l'état initial s_0 satisfait la condition initiale),
  2. $\forall i \in \mathbb{N} : (s_i \rightarrow s_{i+1}) \vee (s_i = s_{i+1})$, chaque deux états successifs (s_i, s_{i+1}) satisfait le prédicat avant-après $BA_e(x, x')$ pour des événements e et des variables x, ou bien que toutes les valeurs des variables restent inchangées (étapes de bégaiement)

Soit $\sigma \in$ Traces(S), une propriété φ sur les séquences d'états du système S est une propriété d'état, ou une propriété relationnelle ou une propriété temporelle ; la sémantique sur les traces unifie la sémantique sur les états et pairs d'états comme suit :

1. une propriété d'état φ est une propriété de trace telle que : $\sigma, \xi \models \varphi$, si $s_0, \xi \models \varphi$.
2. une propriété relationnelle φ est aussi une propriété de trace en étendant la sémantique sur les pairs d'états vers une sémantique sur les traces comme suit : $\sigma, \xi \models \varphi$, si $(s_0, s_1), \xi \models \varphi$.

Les propriétés temporelles contiennent les prédicats d'états, les propriétés relationnelles et la combinaison temporelle de ces propriétés. Notre extension est la même qu'en TLA$^+$ et un système S est spécifié par l'expression temporelle suivante :

$$\begin{aligned} \text{Specification(S)} \triangleq\ & \wedge\ \text{Init(S)} \quad \text{définit les conditions initiales,} \\ & \wedge\ \Box[\text{Next(S)}]_{<\text{unprimed\_var(S)}>} \quad \text{définit la construction des traces} \\ & \wedge\ \text{WF}_{\text{unprimed\_var(S)}}(\text{S}) \quad \text{définit la contrainte d'équité faible} \\ & \wedge\ \text{SF}_{\text{unprimed\_var(S)}}(\text{S}) \quad \text{définit la contrainte d'équité forte} \end{aligned}$$

Où $\text{WF}_{\text{unprimed\_var(S)}}(\text{S})$ définit la condition d'équité faible sur le système S et $\text{SF}_{\text{unprimed\_var(S)}}(\text{S})$ définit la condition d'équité forte sur le système S.
$\text{WF}_{\text{unprimed\_var(S)}}(\text{S})$ et $\text{SF}_{\text{unprimed\_var(S)}}(\text{S})$ sont définis comme suit :

$\text{WF}_{\text{unprimed\_var(S)}}(\text{S}) \triangleq \bigwedge_{\text{E} \in \text{WF\_Events(S)}} \text{WF}_{\text{unprimed\_var(S)}}(\text{E})$ et

$\text{SF}_{\text{unprimed\_var(S)}}(\text{S}) \triangleq \bigwedge_{\text{E} \in \text{SF\_Events(S)}} \text{SF}_{\text{unprimed\_var(S)}}(\text{E})$

où WF_Events(S) est l'ensemble des événements sous hypothèses d'équité faible et SF_Events(S) est l'ensemble des événements sous hypothèses d'équité forte. $\text{WF}_{\text{unprimed\_var(S)}}(\text{E})$ dénote que l'événement E est exécuté sous hypothèses d'équité faible et $\text{SF}_{\text{unprimed\_var(S)}}(\text{E})$ dénote que l'événement E est exécuté sous hypothèses d'équité forte. En effet, à chaque événement est associée une condition d'équité qui peut être faible, forte ou indéfinie.

Dans TLA$^+$, pour une action e, la condition de permission Enabled(e) est définie par la quantification existentielle sur les occurrences primées des variables d'état ; ainsi, le prédicat d'état Enabled(e) est vrai pour les états qui ont un état successeur relié par une occurrence de l'événement e. $\text{Enabled(e)} \triangleq \exists \text{x}' : \text{BA}_\text{e}(\text{x}, \text{x}')$.

La garde grd(e) dans B est interprétée par la condition Enabled(e) en TLA$^+$.

Nous pouvons résumer la sémantique de l'extension de la notation B sur les traces par les équivalences suivantes en TLA$^+$:

$$\begin{aligned} \text{grd(e)} &\triangleq \text{Enabled(e)} \\ \text{BA}_\text{e} &\triangleq \text{Next} \end{aligned}$$

*** Interprétation des formules**

Soient V l'ensemble des variables flexibles et X l'ensemble des variables rigides du système S. Soient $\sigma = s_0 s_1 ...$ une séquence d'états et ξ une valuation des variables rigides de S. Soient $[|\text{x}|]^{\xi}_{\text{s}_\text{i}}$ la valeur de la variable x à l'état s_i et $[|\text{f}(\text{t}_1, ..., \text{t}_\text{n})|]^{\xi}_{\text{s}_\text{i}}$ l'interprétation du terme $\text{f}(\text{t}_1, ..., \text{t}_\text{n}$ à l'état s_i pour une valuation ξ.

Termes

$$[|x|]^{\xi}_{s_i} = \begin{cases} \xi(x) & x \in X; \\ s_i(x) & x \in V. \end{cases}$$

$$[|f(t_1, ..., t_n)|]^{\xi}_{s_i} = [|f|]([|t_1|]^{\xi}_{s_i}, ..., [|t_n|]^{\xi}_{s_i})$$

Dans ce qui suit, nous dénotons par $s_i, \xi \models sp$ la satisfaction du prédicat d'état sp à l'état s_i du système de transition et par $\sigma, \xi \models F$ la satisfaction de la formule temporelle F sur une trace $\sigma \in$ Traces(S).

Propositions atomiques

$s_i, \xi \models ap$ ssi ap est vrai à l'état s_i

Prédicats d'état

$s_i, \xi \models sp$ ssi sp est satisfait à l'état s_i

Formules booléennes

$s_i, \xi \models \neg sp$ ssi il n'est pas vrai que $s_i, \xi \models sp$
$s_i, \xi \models sp_1 \wedge sp_2$ ssi $s_i, \xi \models sp_1$ et $s_i, \xi \models sp_2$
$s_i, \xi \models sp_1 \vee sp_2$ ssi $s_i, \xi \models sp_1$ ou $s_i, \xi \models sp_2$
$s_i, \xi \models sp_1 \Rightarrow sp_2$ ssi $s_i, \xi \models \neg sp_1$ ou $(s_i, \xi \models sp_1 \ \wedge \ s_i, \xi \models sp_2)$
$s_i, \xi \models sp_1 \Leftrightarrow sp_2$ ssi $s_i, \xi \models sp_1 \Rightarrow sp_2$ et $s_i, \xi \models sp_2 \Rightarrow sp_1$
$s_i, \xi \models (\exists x)\ sp$ ssi $(\exists x) \in V :\ s_i, \xi \models sp$
$s_i, \xi \models (\forall x)\ sp$ ssi $(\forall x) \in V :\ s_i, \xi \models sp$

Invariant

Un prédicat I est un invariant ssi $I \in SP_{V \cup X} \wedge \forall s.(s \in S \Rightarrow s, \xi \models I)$

Formules de transition

$(s, s'), \xi \models GS(e)$ ssi $s \xrightarrow{e} s'$

$s, \xi \models [e]sp'$ ssi pour toute exécution de l'événement e, si $s \xrightarrow{e} s'$ alors l'état $s', \xi \models sp'$
[e]sp est la plus faible précondition wp(e, sp). Cette plus faible
condition assure que le prédicat d'état sp est vrai après exécution de e.

$s_i, \xi \models \langle e \rangle sp'$ ssi il existe une exécution de l'événement e, tel que si $s \xrightarrow{e} s'$ alors l'état $s', \xi \models sp'$
$\langle e \rangle$sp est la plus faible précondition conjuguée $\neg$wp(e, $\neg$sp) qui donne la possibilité d'atteindre le prédicat d'état sp après exécution de e.

Formules temporelles

Nous interprétons une formule temporelle comme une assertion sur les traces d'exécution. Dans les définitions ci-dessous, $\sigma|_i, \xi \models F$ signifie que la formule F est satisfaite sur le suffixe de σ qui commence à partir de i.

$\sigma, \xi \models \Box F$ ssi $\sigma|_i, \xi \models F$ pour tout $i \in \mathbb{N}$
La formule $\Box F$ (always F) affirme que F est vraie à tout instant du comportement σ.

$\sigma, \xi \models \Diamond F$ ssi $\sigma|_i, \xi \models F$ pour un certain $i \in \mathbb{N}$ $(\Diamond F \equiv \neg\Box\neg F)$
La formule $\Diamond$F(eventually F) affirme que F est vraie pour quelques suffixes de la trace σ.

$\sigma, \xi \models \Box\Diamond F$ ssi pour tout $i \in \mathbb{N}$ il existe $j \geq i$ tel que $\sigma|_j, \xi \models F$
La formule $\Box\Diamond$F affirme que F est vraie indéfiniment sur σ.

$\sigma, \xi \models \Diamond\Box F$ ssi il existe $j \in \mathbb{N}$ tel que pour tout $i \geq j$, $\sigma|_i, \xi \models F$.
La formule $\Diamond\Box$F affirme que F est vraie infiniment souvent sur σ.

Propriété Leads-to :
Cette formule affirme que chaque suffixe qui satisfait la propriété temporelle F est suivi par des suffixes qui satisfont la propriété temporelle G.

$\sigma, \xi \models F \rightsquigarrow G$ ssi pour tout $i \in \mathbb{N}$, si $\sigma|_i, \xi \models F$ alors $\sigma|_j, \xi \models G$ pour quelques $j \geq i$

$F \rightsquigarrow G \equiv \Box(F \Rightarrow \Diamond G)$.

Propriété de vivacité sous hypothèse d'équité faible :
Un comportement satisfait une hypothèse d'équité faible pour un événement e si et seulement si le fait que e est indéfiniment autorisé entraîne que GS(e) est infiniment souvent exécutée. Ce qui s'exprime par la formule : $WF(e) = \Diamond\Box$ grd(e) $\Rightarrow \Box\Diamond$GS(e).

$\sigma, \xi \models WF(e)$ ssi il existe $j \in \mathbb{N}$ tel que pour tout $i \geq j$, $\sigma|_i, \xi \models$ grd(e) alors pour tout $n \in \mathbb{N}$, il existe $m \in \mathbb{N}$ tel que pour tout $k \geq n + m$, $(s_i, s_k), \xi \models GS(e)$

Propriété de vivacité sous hypothèse d'équité forte :
Un comportement satisfait une hypothèse d'équité forte pour un événement e si et seulement si le fait que e est infiniment souvent autorisé entraîne que GS(e) est infiniment souvent exécutée. Ce qui s'exprime par la formule : $SF(e) = \Box\Diamond$ grd(e) $\Rightarrow \Box\Diamond$GS(e).

$\sigma, \xi \models SF(e)$ ssi pour tout $i \in \mathbb{N}$, il existe $j \in \mathbb{N}$ tel que pour tout $l \geq i + j$, $\sigma|_l, \xi \models$ grd(e) alors pour tout $n \in \mathbb{N}$, il existe $m \in \mathbb{N}$ tel que pour tout $k \geq n+m$, $(s_i, s_k), \xi \models GS(e)$.

5.3.3 Règles de preuves des propriétés de vivacité

Dans cette section, nous introduisons les règles notées WF et SF pour la vérification des propriétés de vivacité élémentaires sous condition d'équité et WFO pour vérifier les propriétés de vivacité complexes utilisant la notion de relation bien formée.

Nous avons réalisé ce travail en nous inspirant de règles analogues conçues pour TLA$^+$ et notre intention était de les traduire dans un prouveur comme Isabelle. Nous n'avons pas abandonné ce projet mais nous avons trouvé plus facile de transformer B temporel en un module TLA$^+$ et d'utiliser l'environnement de TLA$^+$ pour conduire les preuves des propriétés temporelles. Dans les perspectives, nous envisageons de faire le codage de B en Isabelle (prouveur de théorèmes général) pour pouvoir utiliser ces règles de preuve..

Règle sous hypothèse d'équité faible :
Soit S un système d'événements B et WF(S) l'ensemble des événements exécutés sous hypothèse d'équité faible du système S. La règle à appliquer pour prouver une propriété de *"Leads-to"* sous hypothèse d'équité faible est :

$$\text{WF.}\ \frac{\begin{array}{l} I \wedge P \wedge \neg Q \Rightarrow [e](P \vee Q) \text{ pour tout événement e de S} \\ \text{il existe un événement } e \in S \text{ telque :} \\ \quad I \wedge P \wedge \neg Q \Rightarrow \langle e \rangle \text{ true} \wedge [e]Q \\ \quad e \in WF(S) \end{array}}{S \models P \rightsquigarrow Q}$$

Dans cette règle, P et Q sont des prédicats d'état, I est l'invariant du système S. Par la première prémisse, n'importe quel successeur d'un état satisfaisant P doit satisfaire P ou Q, ainsi P est vrai tant que Q ne l'est pas. Par la deuxième prémisse, il existe un successeur d'un état satisfaisant P doit satisfaire Q et s'assure qu'en tous ces états, l'événement e est permis ($\langle e \rangle$ true signifie la condition de faisabilité de l'événement e), et la condition d'équité faible assurer que fatalement l'événement e se produira, à moins que Q ne soit devenu vrai avant. Enfin, la troisième prémisse s'assure que e est un événement pour lequel l'équité faible est assurée.

Preuve de propriétés de vivacité sous hypothèse d'équité faible
Supposons que $\sigma = s_0, s_1, ...s_i, ...$ est une trace satisfaisant $\Box I \wedge WF(e)$, et que P est satisfait à l'état s_i. Nous devons prouver que Q est satisfait dans quelques états s_j avec $j \geq i$. Soit s_0 l'état initial et s_i satisfait P. Supposons qu'aucun état successeur ne satisfait Q, ainsi tous les états successeurs doivent satisfaire P. Par la seconde premise, il existe un successeur d'un état satisfaisant P dans lequel un événement e est permis et son exécution mène toujours dans un état satisfaisant Q (contradiction). Ainsi, à partir d'un état s_i satisfaisant P, nous pouvons atteindre un état $s_j (j \geq i)$ satisfaisant Q par l'exécution d'un événement e sous hypothèse d'équité faible.

Règle sous hypothèse d'équité forte :
Soit S un système d'événements B et SF(S) l'ensemble des événements d'équité forte. Similaire à la règle précédente, la règle suivante est utilisée pour prouver une formule de *"Leads-to"* sous hypothèse d'équité forte.

$$\text{SF.}\ \frac{\begin{array}{l} I \wedge P \wedge \neg Q \Rightarrow [e](P \vee Q) \text{ pour tout événement e de S} \\ \text{il existe un événement } e \in S \text{ telque :} \\ \quad I \wedge P \wedge \neg Q \Rightarrow [e]Q \\ \quad S \models \Box(I \wedge P \wedge \neg Q) \Rightarrow \Diamond \text{grd } (e) \\ \quad e \in SF(S) \end{array}}{S \models P \rightsquigarrow Q}$$

Dans cette règle, P et Q sont des prédicats d'état, I est l'invariant du système S, e est un événement d'équité forte. Nous supposons que σ est un comportement satisfaisant $\Box I \wedge SF(e)$ et

P est satisfait à l'état s_i.

Nous devons prouver que Q est satisfait par quelques états s_j avec $j \geq i$. La première prémisse impose que tout successeur d'un état satisfaisant P doit satisfaire P ou Q. La deuxième prémisse impose qu'il existe un événement $e \in S$ où son exécution à partir d'un état satisfaisant p évolue vers un état satisfaisant Q. La troisième premise assure qu'en tous ces états, l'événement e est permis, et ainsi la condition d'équité forte assure que par la suite e se produira, à moins que Q ne soit devenu vrai avant. Enfin, la dernière prémisse assure que e est un événement avec une équité forte.

Utilisation de la règle WFO **:**
La règle WFO est utilisée pour vérifier les propriétés de vivacité en utilisant les relations bien formées. $(S, \prec)$ est une relation binaire telle qu'il ne va pas exister une chaîne descendante infinie $x_1 \prec x_2, ...$ d'éléments $x_i \in S$. F et G sont des formules temporelles.

$$\text{WFO.}\frac{\begin{array}{c}(S, \prec) \text{ est une relation bien } \textit{formée} \text{ sur un ensemble S} \\ \forall x \in S : F(x) \rightsquigarrow G \vee (\exists y \in S : (y \prec x) \wedge F(y))\end{array}}{(\exists x \in S : F(x)) \rightsquigarrow G \qquad (x \text{ n}'\text{est pas libre dans G})}$$

Dans cette règle, x et y sont des variables rigides telles que x n'apparaît pas dans G et y n'apparaît pas dans F. La deuxième hypothèse de la règle est elle-même une formule temporelle qui exige que chaque occurrence de F, pour n'importe quelle valeur $x \in S$, soit suivie d'une occurrence de G, ou par un certain F, pour une valeur de y plus petite. Puisque la première hypothèse s'assure qu'il ne peut pas y avoir une chaîne descendante infinie des valeurs dans S, fatalement G doit être vrai.

Autres règles de vérification :

$$\frac{P \rightsquigarrow Q \qquad Q \rightsquigarrow R}{P \rightsquigarrow R}\ (\text{trans}) \qquad \frac{P \rightsquigarrow Q \qquad R \rightsquigarrow Q}{P \vee R \rightsquigarrow Q}\ (\text{disj})$$

$$\frac{P \Rightarrow Q}{P \rightsquigarrow Q}\ (\text{dedu}) \qquad \frac{P \rightsquigarrow Q}{(\exists x : P(x)) \rightsquigarrow (\exists x : Q(x))}\ (\text{exists})$$

Ces règles sont utilisées pour combiner des formules élémentaires de Leadsto.

5.4 Transformation d'un modèle B temporel en un module TLA$^+$

Dans cette section, nous donnons dans un premier temps, les règles de transformation d'un modèle B temporel en un module TLA$^+$. Dans un second temps, nous présentons le prototype B2TLA$^+$, que nous avons développé pour automatiser cette transformation.

5.4.1 Les règles de traduction d'un modèle B temporel en un module TLA$^+$

Dans cette partie, nous donnons les règles de traduction des données, des événements et des formules logiques d'un modèle B temporel vers un module TLA$^+$.

Modèle B	*Module TLA$^+$*
SYSTEM	MODULE
REFINEMENT	MODULE
REFINES	EXTENDS
CONSTANTS	CONSTANTS
VARIABLES	VARIABLES
PROPERTIES	ASSUME
INVARIANT	INVARIANT
INITIALISATION	INIT
ASSERTIONS	THEOREM
FAIRNESS	FAIRNESS
EVENTUALITY	LIVENESS

Tab. 5.2 – Noms des clauses en B et TLA$^+$.

*** Correspondance entre les noms de clauses de B et TLA$^+$**

La correspondance entre les clauses d'un modèle B temporel et d'un module TLA$^+$ est donnée par le tableau 5.2.

*** Traduction des données**

La modélisation des données en B et TLA$^+$ est basée sur les variables, ensembles et fonctions ; mais B manipule les relations alors que TLA$^+$ non.

B utilise une simplification de la théorie des ensembles classique. Les constructions basiques de la théorie des ensembles sont : produit cartésien ($\times$), ensemble des parties ($\mathbb{P}$) et la compréhension

Construction basique	*Définition*
Set $\times$ Set	dénotation du produit cartésien
$\mathbb{P}$(Set)	dénotation de l'ensemble des sous-ensembles d'un ensemble
({Variable | Prédicat })	définition d'un ensemble en compréhension

Tab. 5.3 – Constructions basiques de la théorie des ensembles en B.

Construction basique	*Définition*
$\{x \in S : P\}$	l'ensemble des éléments x de S satisfaisant la propriété P
$\{e : x \in S\}$	l'ensemble des expressions de la forme e, pour chaque x appartenant à S

TAB. 5.4 – Constructions basiques de la théorie des ensembles en TLA^+.

($\{ \mid \}$). Le tableau 5.3 exprime les constructions basiques de la théorie des ensembles en B.

En TLA^+, les constructions de base de la théorie des ensembles sont basées sur les formes simples $\{x \in S : P\}$ et $\{e : x \in S\}$, et les formes générales sont définies en fonction de celles ci. Le tableau 5.4 exprime les constructions basiques de la théorie des ensembles en TLA^+.

En TLA^+, les deux opérateurs très utilisés de la théorie des ensembles sont UNION et SUBSET, définis comme suit :

- UNION S est l'union de tous les éléments de S et,
- SUBSET S est l'ensemble de tous les sous ensembles de S. $T \in$ SUBSET S ssi $T \subseteq S$ (c'est l'ensemble des parties de $\mathbb{P}(S)$ ou 2^S).

B et TLA^+ sont basés sur la théorie des ensembles de Zermelo-Fraenkel (ZF). Ceci nous permet d'établir une correspondance entre les constructions de B et TLA^+. Dans le tableau 5.5, nous donnons les règles de traduction des données d'un modèle B en un module TLA^+.

TLA^+ ne manipule que les fonctions totales alors que B manipule les fonctions totales et partielles. Les fonctions partielles en B sont transformées en fonctions totales de la manière suivante :

Soit $F_i : S \nrightarrow T$, cette fonction se transforme en $[S \rightarrow T \cup \mathit{Undef}]$ où *Undef*, désigne l'image des éléments de S où la fonction est indéfinie. Nous aurons :

$F = F_u \cup F_i$ où $F_u : (S - \mathrm{dom}(F_i)) \lhd F$ ($\lhd$ est un symbole de restriction sur le domaine de F)
$F_u = (\lambda x.x \in S - \mathrm{dom}(F_i) \mid \mathrm{Undef})$

Dans le tableau 5.6, nous donnons la correspondance entre les fonctions en B et TLA^+.

*** Traduction des événements**

Soient $e_1, ..., e_n$ les événements d'un modèle B. Un événement e_i transforme l'état courant du système représenté par l'ensemble des variables noté x vers un état caractérisé par les variables x'. Un événement e_i est donc une relation entre x et x' au sens de la logique TLA qui se note par $R(e_i)(x, x')$.

Nous pouvons définir pour chaque modèle B une spécification TLA^+ où :

$$\mathrm{Next} \triangleq R(e_1)(x, x') \vee ... \vee R(e_n)(x, x').$$

Où $R(e_i)(x, x')$ est une relation TLA correspondant à une des trois formes des événements B :

- $P(x, x')$ pour un événement simple,

Source B	*Génération TLA$^+$*	*Définition*
c_1, c_2 dans CONSTANTS	c_1, c_2 dans CONSTANTS	Constantes du modèle
v_1, v_2 dans VARIABLES	v_1, v_2 dans VARIABLES	Variables du modèle
E_1 dans SETS $E_2 = \{x_1, .., x_n\}$ dans SETS	E_1 dans CONSTANTS $x_1, .., x_n$ dans CONSTANTS et $E_2 = \{x_1, .., x_n\}$ est une définition qui apparaît après la clause ASSUME	Ensembles du modèle
$\{e_1, ..., e_n\}$	$\{e_1, ..., e_n\}$	Ensemble formé par les éléments e_i
$\{x \mid x \in S \wedge P\}$	$\{x \in S : P\}$	Ensemble des éléments x dans S satisfaisant P
$\{e \mid x \in S\}$	$\{e : x \in S\}$	Ensemble des éléments e tel que $x \in S$
$\mathbb{P}(S)$	SUBSET S	Ensemble des sous-ensembles de S

TAB. 5.5 – Constantes, Variables et Ensembles en B et TLA$^+$.

- $G(x) \wedge P(x, x')$ pour un événement gardé,
- $\exists t.(G(t, x) \wedge P(x, x', t))$ pour un événement indéterministe.

Les événements d'un modèle B événementiel correspondent à des actions dans un module TLA$^+$. Dans le tableau 5.7, nous donnons les relations TLA$^+$ équivalentes associées aux trois types d'événements.

*** Traduction des formules logiques**

- *Les formules propositionnelles du premier ordre.* B et TLA$^+$ se basent sur la logique des prédicats du premier ordre où un prédicat est défini à partir des propositions atomiques, des opérateurs $(\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow)$ et des quantificateurs $(\forall, \exists)$.
  Les formules logiques en B ont une syntaxe très proche de celle de TLA$^+$. Le tableau 5.8, qui définit la traduction des prédicats B en TLA$^+$ montre les légères différences.
- *Les formules temporelles.* La traduction des propriétés de vivacité est évidente parce que l'extension est inspirée de la syntaxe de TLA$^+$. Dans le tableau 5.9, nous donnons les règles de traduction des propriétés de vivacité d'un modèle B temporel en un module TLA$^+$ (où e est un événement, F et G sont des propriétés de vivacité).

Source B	*Génération TLA$^+$*	*Définition*
f(e)	f[e]	Application de la fonction f
Dom(f)	Domain f	Domaine de la fonction f
$(\lambda x.x \in S \mid e)$ Lambda expression	$[x \in S \mapsto e]$	Fonction f tel que $f[x] = e$ pour $x \in S$
$S \rightarrow T, S \twoheadrightarrow T,$ $S \rightarrowtail T, S \rightarrowtail\!\!\!\!\twoheadrightarrow T$	$[S \rightarrow T]$	Ensemble de fonctions f avec $f[x] \in T$ pour $x \in S$
$F_i : S \nrightarrow T$ nous avons $F = F_u \cup F_i$ où $F_u : (S - dom(F_i)) \lhd F$ et $F_u = (\lambda x.x \in S - dom(F_i)$ $\mid$ Undef)	$[S \rightarrow T \cup Undef]$	F_i se transforme en une fonction totale de S vers le codomaine $(T \cup Undef)$
f[S]	$Image(f, S) \equiv \{f[x] : x \in S\}$	Fonction f tel que $x \in S$
$f(x) := y$	$f' = [f \text{ EXCEPT!}[x] = y]$	définit une fonction égale à f sauf au point x où elle est égale à $f[x] = y$

TAB. 5.6 – Correspondance des fonctions en B et TLA$^+$.

Evénement B	*Action TLA$^+$*
$e \equiv$ BEGIN x : $P(x_0, x)$ END	$e \equiv \wedge P(x, x')$ $\wedge$ UNCHANGED $<$ unprimed_var(e) $>$
$e \equiv$ SELECT G(x) THEN $x : Q(x_0, x)$ END	$e \equiv \wedge G(x)$ $\wedge Q(x, x')$ $\wedge$ UNCHANGED $<$ unprimed_var(e) $>$
$e \equiv$ ANY t WHERE $G(t, x)$ THEN x : $R(x_0, x, t)$ END	$e \equiv \exists t : \wedge G(t, x) \wedge R(x, x', t)$ $\wedge$ UNCHANGED $<$ unprimed_var(e) $>$

TAB. 5.7 – Translation des événements B en relations TLA.

Formule B	*Formule TLA$^+$*
P	P
¬P	¬P
P ∧ Q	P ∧ Q
P ∨ Q	P ∨ Q
P ⇒ Q	P ⇒ Q
P ⇔ Q	P ⇔ Q
!x.(P ⇒ Q) !(x, y).(P ⇒ Q)	∀x : (P ⇒ Q) ∀(x, y) : (P ⇒ Q)
∃x.(P ∧ Q)	∃x : (P ∧ Q)

TAB. 5.8 – Traduction des prédicats de B en TLA$^+$.

Source B	*Génération TLA$^+$*	*Définition*
WF(e)	WF(e)	équité faible de l'événement e
SF(e)	SF(e)	équité forte de l'événement e
F ⇝ G	F ⇝ G	F leads to G
□F	□F	F est toujours vrai
◊F	◊F	F est éventuellement vrai

TAB. 5.9 – Les propriétés de vivacité en B et TLA$^+$.

Nous trouvons à l'annexe A, toutes les primitives de traduction d'un modèle B en un module TLA$^+$.

5.4.2 Le Prototype B2TLA$^+$

Le prototype B2TLA$^+$ effectue la traduction d'un modèle B temporel en un module TLA$^+$. Cette transformation est basée sur la technique de la traduction dirigée par la syntaxe qui consiste à associer des actions sémantiques à la grammaire de B étendue par les clauses d'équité et de fatalité pour obtenir un module TLA$^+$ équivalent. La figure 5.4 décrit l'architecture du système. Le prototype est implémenté en utilisant l'outil Flex-Yac. Nous présentons à l'annexe A, l'interface du prototype.

La figure 5.4 décrit la méthode à suivre par l'utilisateur. Il spécifie tout d'abord un modèle B qu'il valide avec Click_n_prove. Ensuite, il ajoute les propriétés de vivacité qu'il voulait satisfaire sur le modèle. Il introduit les hypothèses d'équité (équité faible, équité forte, pas d'équité) pour chaque événement ainsi que les propriétés de fatalité à vérifier. Ces propriétés sont ajoutées dans

un modèle B afin d'obtenir un modèle B temporel. Par la suite ce modèle est traduit vers son équivalent TLA$^+$ pour la vérification des propriétés de vivacité.

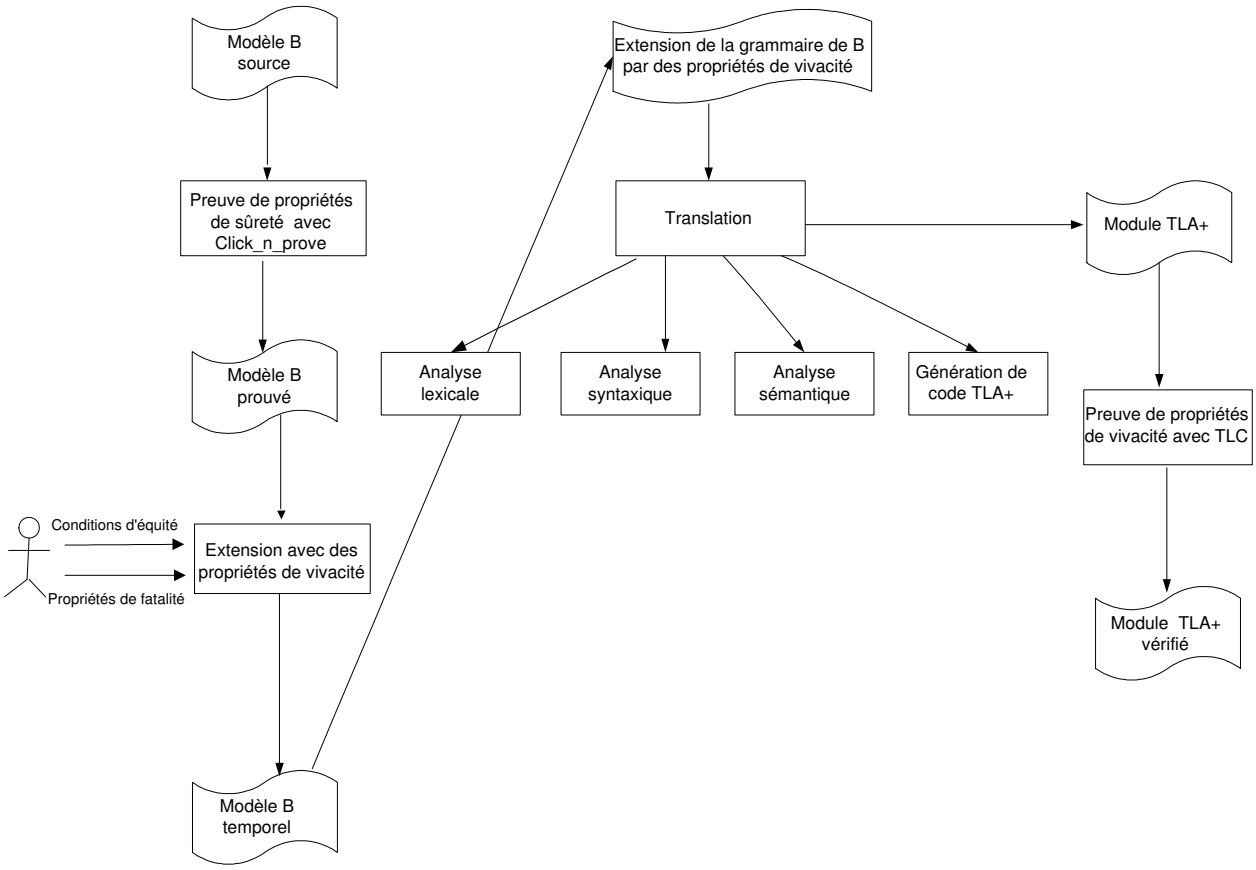


FIG. 5.4 – Méthode proposée utilisant le système de prototype B2TLA$^+$.

Dans ce quit suit, nous utilisons deux outils pour la vérification des propriétés de vivacité : le prouveur de théorèmes Isabelle/TLA et le model checker TLC.

Pour être tout à fait rigoureux, il faudrait montrer que la sémantique d'un modèle B temporel est conservé par la transformation de B2TLA$^+$. Nous n'avons pas fait ce travail qui risquait d'être long et fastidieux mais nous envisageons de prouver la correction des règles de traduction dans les travaux futurs.

5.5 Validation d'un module TLA$^+$ à l'aide du prouveur de théorèmes Isabelle/TLA

Dans cette section, nous utilisons comme outil de preuve, le prouveur de théorème Isabelle/TLA (encodage de TLA dans la logique d'ordre supérieur de l'assistant de preuve générique Isabelle) [Merz, 1999]. Ce prouveur repose principalement sur les logiques du premier ordre et d'ordre supérieur ainsi que le calcul des séquents et le lambda calcul. La syntaxe logique et les règles d'inférence sont spécifiées déclarativement, permettant la construction de la preuve des

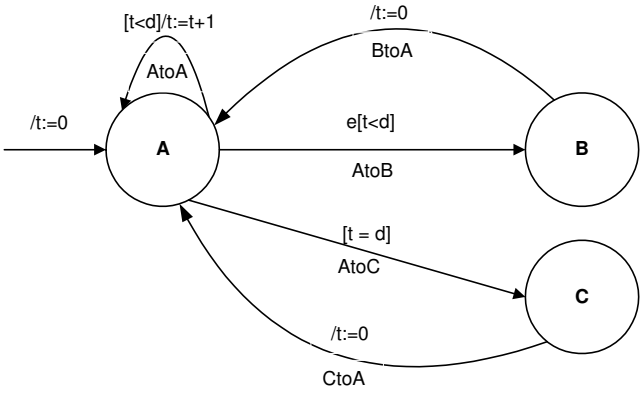


FIG. 5.5 – Statechart représentant le time-out.

spécifications en une étape. Le prouveur Isabelle est très utilisé ; il permet une automatisation de la preuve de théorèmes. Ce prouveur offre des tactiques qui sont des commandes utilisées par l'utilisateur pour guider le système dans une compilation automatique des objets de preuve.

Nous illustrons dans ce qui suit, la méthode proposée sur l'exemple simple d'un time-out générique.

5.5.1 Description informelle de l'exemple

Le timeout est un système réactif qui peut être dans un des trois états *A,B* ou *C*. Une fois que le système est à l'état *A*, si un signal de l'environnement apparaît dans un intervalle de *d* unités de temps alors le système passe à l'état *B*. Si rien ne s'est passé à l'issue de la période *d*, le système passe à l'état *C*. Pour avoir un comportement infini, les états B et C rebouclent sur l'état A avec une remise à 0 du temps. Le système est représenté par un statechart à la figure 5.5.

5.5.2 Illustration de la méthode proposée

Etape 1 : Modèle B du système

Nous appelons *Etats* l'ensemble des trois états du système *A, B, C* ; *d* est une constante qui désigne la durée maximale à passer à l'état *A*. Les variables *etat_sys, t, e* désignent respectivement l'état du système, le compte du temps et l'occurrence d'un événement *e*.

L'événement *AtoA* exprime les conditions sous lesquelles le système reste à l'état *A* (pas de réception de l'événement *e* et la valeur de *t* n'a pas atteint *d* unités de temps). L'événement *AtoB* exprime la condition de transit de l'état *A* vers *B* sous l'occurrence de l'événement *e*. L'événement *AtoC* exprime la condition de transit de l'état *A* vers *C*. L'événement *Change* permet de changer d'une manière aléatoire l'occurrence de l'événement *e*. L'événement *BtoA* reboucle l'état B sur l'état A et l'événement *CtoA* reboucle l'état C vers l'état A.

MODEL time_out
SETS $Etats = \{A, B, C\}$;
CONSANTS d
PROPERTIES $d = 10$
VARIABLES $etat\_sys$, t, e
INVARIANT
$etat\_sys \in Etats \wedge t \in \text{NATURAL} \wedge e \in \text{BOOL} \wedge t \leq d$
INITIALISATION
$etat\_sys := A \parallel t := 0 \parallel e := FALSE$
EVENTS
AtoA $\triangleq$ **SELECT** $etat\_sys = A \wedge e = \text{FLASE} \wedge t < d$ **THEN** $etat\_sys := A \parallel t := t+1$ **END** ;
AtoB $\triangleq$ **SELECT** $etat\_sys = A \wedge e = \text{TRUE} \wedge t < d$ **THEN** $etat\_sys := B$ **END** ;
AtoC $\triangleq$ **SELECT** $etat\_sys = A \wedge t = d$ **THEN** $etat\_sys := C$ **END** ;
BtoA $\triangleq$ **SELECT** $etat\_sys = B$ **THEN** $etat\_sys := A \parallel t := 0$ **END** ;
CtoA $\triangleq$ **SELECT** $etat\_sys = C$ **THEN** $etat\_sys := A \parallel t := 0$ **END** ;
Change $\triangleq$ **BEGIN** $e :: \text{BOOL}$ **END**
END

FIG. 5.6 – Modèle B du time_out.

Etape 2 : Preuves

Le modèle B a été prouvé par le prouveur et toutes les obligations de preuve générées ont été vérifiées d'une manière automatique et interactive.

Etape 3 : Modèle B temporel

Nous exprimons dans ce modèle des propriétés de vivacité. Nous commençons par donner des hypothèses d'équité sur l'exécution de certains événements. Nous donnons des hypothèses d'équité forte sur l'exécution de l'événement *AtoB* parce qu'il est activé infiniment souvent et une équité faible à l'événement *AtoC* pour ne pas exécuter indéfiniment les événements *AtoA* et *change*. Ces hypothèses sont ajoutées au niveau de la clause FAIRNESS sous la forme : SF(AtoB) $\wedge$ WF(AtoC).

Notre modèle doit vérifier la propriété suivante : le système atteindra fatalement l'état B ou l'état C ($etat\_sys = A \rightsquigarrow etat\_sys = B \vee etat\_sys = C$). Cette propriété est définie au niveau de la clause EVENTUALITY.

Le modèle B temporel est présenté à la figure 5.7.

Etape 4 : Traduction du modèle B temporel en un module TLA$^+$

Pour pouvoir vérifier les propriétés d'équité et de fatalité, nous transformons le modèle B temporel en un module TLA$^+$. Ce module décrit les variables, les prédicats et les actions de la spécification. La transformation d'un modèle B temporel en un module TLA$^+$ est réalisée par l'outil B2TLA$^+$. Le module TLA$^+$ obtenu est présenté à la figure 5.8.

MODEL time_out
SETS $Etat\_sys = \{A, B, C\}$;
CONSANTS d
PROPERTIES $d = 10$
VARIABLES $etat\_sys$, t, e
INVARIANT
$etat\_sys \in Etats \wedge t \in$ NATURAL $\wedge\ e \in$ BOOL$\wedge\ t \leq d$
INITIALISATION
$etat\_sys := A \ ||\ t := 0 \ ||\ e := FALSE$
EVENTS
AtoA $\triangleq$ **SELECT** $etat\_sys = A \wedge e =$ FLASE $\wedge$ t<d **THEN** $etat\_sys := A \ ||\ t := t+1$ **END** ;
AtoB $\triangleq$ **SELECT** $etat\_sys = A \wedge e =$ TRUE $\wedge$ t<d **THEN** $etat\_sys := B$ **END** ;
AtoC $\triangleq$ **SELECT** $etat\_sys = A \wedge t = d$ **THEN** $etat\_sys := C$ **END** ;
Change $\triangleq$ **BEGIN** e : : BOOL **END** ;
BtoA $\triangleq$ **SELECT** $etat\_sys = B$ **THEN** $etat\_sys := A \ ||\ t := 0$ **END** ;
CtoA $\triangleq$ **SELECT** $etat\_sys = C$ **THEN** $etat\_sys := A \ ||\ t := 0$ **END** ;
FAIRNESS
SF(AtoB) $\wedge$ WF(AtoC)
EVENTUALITY
$etat\_sys = A \leadsto etat\_sys = B \vee etat\_sys = C$
END

FIG. 5.7 – Modèle B temporel du time_out.

Etape 5 : Vérification des propriétés de vivacité par le prouveur de théorèmes Isabelle/TLA

Avant d'utiliser le prouveur Isabelle/TLA pour la vérification des propriétés d'équité et de fatalité, nous transformons le module TLA$^+$ en un module accepté par Isabelle utilisant le codage de TLA$^+$ en Isabelle. Ce module est présenté à la figure 5.9. Les propriétés de fatalité sont prouvées par le prouveur Isabelle/TLA.

Le modèle de la figure 5.9 utilise les notions de function d'état (*stfun*), prédicat d'état (*stpred*), d'actions (*action*) et de formules temporelles (*temporal*). En effet, les variables non booléennes sont considérées comme des fonctions d'état (*stfun*), les variables booléennes et l'initialisation (*Init*) sont des prédicats d'état (*stpred*). Pour les actions, la valeur de la variable avant l'exécution de la substitution est dénotée par $ *variable* et la valeur de la variable après l'exécution de la substitution est dénotée par *variable* $.

Le time-out est un exemple simple mais dont la preuve des propriétés de vivacité en Isabelle/TLA est complexe à réaliser par un non expert. Il nécessite un investissement et requiert une bonne maîtrise de l'outil et une connaissance des tactiques utilisées. En plus, nous devons fournir une étape de transformation supplémentaire du module TLA$^+$ en un module compréhensible par le codage de TLA$^+$ en Isabelle. Néanmoins, ce n'est pas applicable par un utilisateur non expert. La vulgarisation de cet outil n'est pas envisageable, le model checking offre, en revanche une alternative plus facile à mettre en œuvre. Cette technique n'est pas aussi satisfaisante que la preuve mais cela a l'avantage d'être utilisé par un non expert des techniques de preuve et c'est un outil à aspect "press-bouton".

MODULE time_out

CONSTANTS *d, A, B, C,*
VARIABLES *etat_sys , t, e*
ASSUME $d \in NATURAL$
$Etat\_sys = \{A, B, C\}$
TypeInvariant = $\land\ t \in NATURAL \land etat\_sys \in Etat\_sys \land t \leq d \land e \in \{TRUE, FALSE\}$
Init = $\land\ etat\_sys = A \land t = 0 \land e = FALSE$
AtoA = $\land\ etat\_sys = A \land t < d \land e = FALSE \land t' = t + 1 \land$ UNCHANGED$_{<etat_sys,e>}$
AtoB = $\land\ etat\_sys = A \land t < d \land e = TRUE \land etat\_sys' = B \land$ UNCHANGED$_{<t,e>}$
AtoC = $\land\ etat\_sys = A \land t = d \land etat\_sys' = C \land$ UNCHANGED$_{<t,e>}$
BtoA = $\land\ etat\_sys = B \land etat\_sys' = A \land t' = 0 \land$ UNCHANGED$_{<e>}$
CtoA = $\land\ etat\_sys = C \land etat\_sys' = A \land t' = 0 \land$ UNCHANGED$_{<e>}$
change = $\land\ e' \in \{TRUE, FALSE\} \land$ UNCHANGED$_{<etat_sys,t>}$
Next = **AtoA** $\lor$ **AtoB** $\lor$ **AtoC** $\lor$ **change** $\lor$ **BtoA** $\lor$ **CtoA**
$trvars = <etat\_sys, t, e>$
FAIRNESS = $\mathrm{SF}_{\mathrm{trvars}}(\mathrm{AtoB}) \land \mathrm{WF}_{\mathrm{trvars}}(\mathrm{AtoC})$
Spec = **Init** $\land\Box[\mathrm{Next}]_{\mathrm{trvars}}\land$ **FAIRNESS**
Liveness = $\land\ etat\_sys{=}A \leadsto etat\_sys = B \lor etat\_sys = C$
THEOREM Spec $\Rightarrow \Box$ **TypeInvariant**
THEOREM Spec $\Rightarrow$ **Liveness**

FIG. 5.8 – Module TLA$^+$ du time_out.

time_out.thy

Timeout = TLA+
datatype *etat_sys = A | B | C*
constdefs *d : : nat "d==10"*
constdefs *etat_sys : : Etats stfun*
t : : nat stfun e : : stpred
rules *base "basevars (etat_sys, t, e)"*
constdefs *InitTimeout : : stpred*
"InitTimeout == PRED etat_sys = #A & t=#0 & e = #False"
AtoA : : action
"AtoA == ACT $ etat_sys = #A & $ t < #d & $ e = #False & etat_sys $= #A & t $ = $ t + # 1 & unchanged e "
AtoB : : action
"AtoB == ACT $ etat_sys = #A & $ t < #d & $ e = #TRUE & etat_sys $= #B & unchanged (t, e) "
AtoC : : action
"AtoC == ACT $ etat_sys = #A & $ t = #d & etat_sys $= # C & unchanged (t, e) "
BtoA : : action
"BtoA == ACT $ etat_sys = #B & etat_sys $= # A & t $ = $ # 1 & unchanged (e) "
CtoA : : action
"CtoA == ACT $ etat_sys = #C & etat_sys $= # A & t $ = $ # 1 & unchanged (e) "
change : : action
"change == ACT e $: : {# True, # FALSE} & unchanged (t, etat_sys) "
Next : : action
"Next == ACT $ (AtoA | AtoB | AtoC | change | BtoA | CtoA)"
Timeout : : temporal
"Timeout == TEMP (Init InitTimeout
& $\Box[\mathrm{Next}]_{(-\mathrm{etat\_sys,t,e})}$
& $\Box\mathrm{SF}(AtoB)_{-(etat\_sys,t,e)}$
& $\Box\mathrm{WF}(AtoC)_{-(etat\_sys,t,e)}$*)"*
Live : : temporal
"Live == TEMP (etat_sys = #A $\leadsto$ etat_sys = #B $\vee$ etat_sys = #C)"

FIG. 5.9 – Module TLA$^+$ du time_out.

5.6 Validation d'un module TLA$^+$ à l'aide du model checker TLC

La technique de model checking effectue la vérification d'un modèle du système par rapport aux propriétés qui sont attendues sur ce modèle. Cette technique repose sur des systèmes finis et décidables. Cette vérification est entièrement automatisée et consiste à explorer tous les cas possibles. Le résultat de cette analyse est soit la confirmation que chaque propriété est maintenue par le modèle, soit qu'elle ne l'est pas. Dans le dernier cas, et c'est un des principaux intérêts de cet outil, le model checker renvoie un contre exemple qui montre pourquoi la propriété n'est pas maintenue.

Nous illustrons dans ce qui suit notre approche sur l'exemple d'un tri de paquets et nous donnons trois niveaux de raffinement.

5.6.1 Description informelle de l'exemple

Une trieuse de paquets est formée d'aiguillages à une entrée et deux sorties reliées entre eux par des conduits. On peut représenter la trieuse (Figure 5.10) par un arbre binaire dont les aiguillages sont les nœuds, les conduits sont les arcs et les bacs représentent les feuilles. Dans la suite, pour des raisons de simplification de notation mais sans perdre en généralité, nous avons fixé la taille de la trieuse à 3 niveaux donc à 8 bacs de réception. Un paquet est introduit dans un magasin récepteur où il est retenu par une porte avant d'entrer dans un conduit qui débouche sur le premier aiguillage.

La trieuse est construite sur un plan incliné et le paquet, une fois sorti du magasin, glisse jusqu'à un des bacs de reception. Chaque paquet a une adresse correspondant à un des bacs d'arrivée. Un dispositif de lecture placé avant la porte du magasin permet de connaître cette adresse. Chaque aiguillage est muni d'un détecteur d'entrée et d'un détecteur sur chacune de ses sorties. Il existe aussi des détecteurs à la sortie des conduits qui débouchent sur les bacs. Les aiguillages devront alors être actionnés pendant que le paquet circule, toutefois, pour éviter de détériorer le paquet, on s'interdit d'actionner un aiguillage quand un paquet s'y trouve. Les détecteurs permettent de déterminer si un actionnement est possible.

5.6.2 Illustration de la méthode proposée

Dans le développement du système de tri, nous construisons un modèle du système automatisé englobant contrôleur et contrôlé.

*** Niveau abstrait**

- *Étape 1 : Modèle B abstrait du système.*

Le modèle abstrait du système se réduit à un seul événement.

- *Constantes du modèle.* Dans ce modèle, nous définissons cinq constantes :
  - *EPARCELS* est un ensemble abstrait qui modélise l'ensemble des paquets,
  - *NODES* est un ensemble abstrait qui modélise l'ensemble des nœuds,
  - *PARCELS* est un sous ensemble de *NODES* qui modélise les paquets entrés dans le système de tri,
  - *BASKETS* est un sous ensemble de *NODES* qui modélise les bacs de destination,

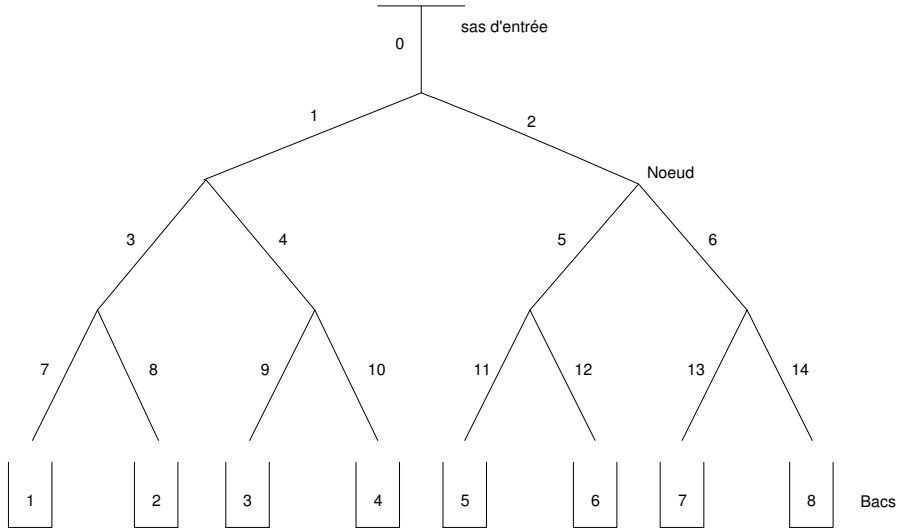


FIG. 5.10 – Tri de paquets postaux.

- *adr* est une fonction qui, à chaque paquet, associe une valeur dans l'ensemble *BASKETS*. Cette constante modélise l'adresse de destination.

CONSTANTS
PARCELS, adr, BASKETS
PROPERTIES
$PARCELS \subset EPARCELS \wedge\ PARCELS \neq \varnothing \wedge BASKETS \neq \varnothing \wedge$
$adr \in PARCELS \rightarrow BASKETS$

- *Variables du modèle.* Le modèle abstrait du système se réduit à la variable *stored*. Elle permet de distinguer les paquets qui ont atteint un bac de sortie. *stored* est une fonction des paquets vers l'ensemble *BASKETS*.

VARIABLES
stored
INVARIANT
$stored \in PARCELS \nrightarrow BASKETS \wedge stored \subseteq adr$
INITIALISATION
$stored := \varnothing$

- *Événements du modèle.* Le modèle du système se réduit au seul événement *Cross_sorting*, qui traduit l'acheminement d'un paquet à son adresse de destination.

Cross_sorting= ANY *p, b* WHERE
$p \in PARCELS \wedge$
$b \in BASKETS \wedge$
$p \notin dom(stored) \wedge$
$p \mapsto b \in adr$
THEN
$stored := stored \cup \{p \mapsto b\}$
END

Le modèle B abstrait est donné en annexe B.

- *Étape 2 : Preuve.*
Le modèle B abstrait est vérifié par l'outil Click_n_Prove et toutes les obligations de preuve générées sont vérifiées d'une manière automatique et interactive. Les statistiques sur les preuves sont résumées dans le tableau de la figure 8.1.

*** Premier raffinement**

- *Étape 1 : Modèle B du système.*
Dans ce raffinement, nous ajoutons de nouvelles variables et de nouveaux événements.

Variables du modèle. Nous introduisons les variables suivantes :

- *channel* modélise le dispositif de tri, elle relie l'entrée à un bac de destination,
- *next* désigne un paquet choisi en entrée mais qui n'est pas libéré dans le dispositif de tri,
- *current* désigne un paquet entré dans le dispositif de tri,
- *adr_next* modélise une mémoire qui lit l'adresse de tout paquet entré,
- *ready_to_sort* est une variable booléenne qui modélise l'état de la trieuse,
- *sorted* modélise l'ensemble des paquets triés.

Nous avons les invariants de typage suivants :

VARIABLES
...
channel, *next*, *current*, *sorted*, *ready_to_sort*, *adr_next*
INVARIANT
...
$channel \in BASKETS \wedge$
$next \in EPARCELS \wedge current \in$ EPARCELS $\wedge$
$ready\_to\_sort \in BOOL \wedge sorted \subseteq PARCELS \wedge adr\_next \in BASKETS$

L'initialisation de ces variables est comme suit :

INITIALISATION
...
$channel :\in BASKETS \;||$
$current :\in UNDEF \;||\; next :\in UNDEF \;||\; sorted := \varnothing \;||$
$ready\_to\_sort := FALSE \;||\; adr\_next :\in BASKETS$

Événements du modèle.
Le dispositif physique. Le dispositif physique est formé d'un sas d'entrée auquel est associée une mémoire représentée par la variable *adr_next* et par un dispositif de tri. Le sas d'entrée a deux fonctions : déterminer l'adresse du prochain paquet à introduire dans la trieuse et libérer ce paquet dans le dispositif de tri par ouverture de la porte de sortie du sas. Le dispositif de tri est représenté par la variable *channel* qui relie l'entrée à un bac de destination ($BASKETS$).

Les paquets. Les paquets faisant partis de l'environnement sont représentés comme éléments de l'ensemble *PARCELS*. L'objectif du système de tri est de diminuer le nombre de paquets à trier et d'augmenter le nombre de paquets triés. La variable *sorted* modélise l'ensemble des paquets triés. *TO_SORT* désigne les paquets restant à trier ; elle est définie par l'expression *PARCELS-sorted*.

DEFINITIONS
TO_SORT == *PARCELS* - *sorted*;
UNDEF == *EPARCELS* - *PARCELS*

Quand un paquet entré est arrivé à destination, la variable *current* qui le désigne prend une valeur indéfinie. La technique employée pour représenter *"l'indéfini"* est d'introduire l'ensemble *EPARCELS* dont *PARCELS* est un sous ensemble et de désigner tout élément indéfini comme élément de *UNDEF*. La sélection et l'identification d'un paquet est modélisée par l'événement *Select_parcel* qui est activé si le dispositif physique est libre et la variable *current* est indéfinie.

Select_parcel=
ANY *p* *Where* *p* $\in$ *TO_SORT* $\wedge$
current $\in$ *UNDEF* $\wedge$
next $\in$ *UNDEF* $\wedge$ *ready_to_sort* = *FALSE*
THEN
adr_next := *adr(p)* || *next* := *p*
END ;

Le contrôleur. Le rôle du contrôleur est d'agir sur la trieuse pour permettre un routage correct, c'est-à-dire pour que tout paquet soit acheminé vers le bac correspondant à son adresse. Pour ce faire, la variable *channel* doit avoir la valeur *adr_next* au moment de l'exécution de l'événement *Release*. Les événements du contrôleur sont *Set_channel* et *Release*. L'événement *Set_channel*, affecte à la variable *channel* la valeur de *adr_next*. D'autre part, l'événement *Release* ne peut être exécuté si un paquet est en cours de tri. Le modèle complet du système est présenté à l'annexe B.

Set_channel=
SELECT
$\neg$ *ready_to_sort* $\wedge$
next $\in$ *PARCELS* $\wedge$
current $\in$ *UNDEF*
THEN
channel := *adr_next* ||
ready_to_sort := *TRUE*
END ;

Release=
SELECT
next $\in$ *PARCELS* $\wedge$
current $\in$ *UNDEF* $\wedge$ *ready_to_sort*
THEN
current := *next* || *next* :$\in$ *UNDEF* ||
ready_to_sort := *FALSE*
END ;

Le déplacement des paquets. Dans ce raffinement, un paquet ne prend pas de temps pendant son parcours de l'entrée vers le bac de sortie. L'événement *Cross_sorting* est raffiné par les variables *current* et *sorted*. Quand cet événement est exécuté, le paquet courant est trié et il devient indéfini (*current* :$\in$ *UNDEF*).

Cross_sorting =
SELECT
current ∈ *PARCELS*
THEN
stored(current) := *channel* ||
sorted := *sorted* ∪ { *current*} ||
current :∈ *UNDEF*
END ;

Pour des raisons de preuve, nous avons ajouté les invariants suivants pour aider le prouveur :

(*ready_to_sort* = TRUE ⇒ *channel* = *adr_next*) ∧
(*current* ∈ PARCELS ⇒ *channel* = *adr(current)*) ∧
(*next* ∈ PARCELS ⇒ *adr_next* = *adr(next)*) ∧

Le modèle du système automatisé doit vérifier l'invariant suivant :

∀ *p.*(*p* ∈ *PARCELS* ∧p ∈ *dom(stored)* ⇒ *stored(p)*= *adr(p)*)

Simulation du modèle par le ProB.
Pour permettre de découvrir les diverses erreurs qui ne peuvent être facilement découvertes par le prouveur, nous avons utilisé le ProB [Leuschel and Butler, 2003b], qui est un simulateur et un model checker pour des spécifications B. Il permet une animation automatique et détecte les erreurs des modèles B ce qui permet aux utilisateurs d'avoir confiance en leur spécification.

Le modèle B du premier raffinement est donné en annexe B.

- *Étape 2 : Preuve.*
Le modèle du premier raffinement est vérifié par le prouveur et toutes les obligations de preuve générées sont vérifiées d'une manière automatique et interactive. Nous trouverons dans le tableau de le figure 8.1 les statistiques sur les preuves.

- *Étape 3 : Modèle B temporel.*
Les propriétés intéressantes sont des propriétés de vivacité. Ces propriétés ne peuvent pas être exprimées au travers des invariants. Pour ce faire, nous faisons une hypothèse que les événements du modèle ont des occurrences qui satisfassent la propriété d'équité faible. Cette propriété est décrite par la formule :

WF(Select_parcel) ∧ *WF(Cross_sorting)* ∧ *WF(Set_channel)* ∧ *WF(Release)*

Il faut entre autre exprimer que :

1. Chaque paquet présenté en entrée arrivera dans un des bacs, cette propriété est décrite par : ∀ *p.*(*p* ∈ *TO_SORT* ⇝ *stored(p)* ∈ *BASKETS)*
2. Chaque paquet entré doit arriver dans le bac correspondant à son adresse de destination, cette propriété est décrite par : ∀ *p.*(*p* ∈ *TO_SORT* ⇝ *stored(p)*= *adr(p))*

- *Étape 4 : Traduction du modèle B temporel en un module TLA*$^+$.
Dans cette section, nous traduisons le modèle B temporel en un module au moyen de B2TLA$^+$ afin de vérifier les propriétés de vivacité. Le module complet TLA$^+$ est donné à l'annexe C. Les figures qui suivent mettent en évidence le modèle B et la traduction correspondante.

REFINEMENT *Parcel_routing*1
REFINES *Parcel_routing*
SETS
EPARCELS;
NODES
CONSTANTS
PARCELS, adr, BASKETS
PROPERTIES
$PARCELS \subset EPARCELS \land$
$PARCELS \neq \varnothing \land$
$BASKETS \neq \varnothing \land$
$adr \in PARCELS \rightarrow BASKETS$

$\Rightarrow$

MODULE *Parcel_routing*1
EXTENDS *Parcel_routing*
CONSTANTS
EPARCELS, PARCELS, BASKETS, NODES, noBaskets, adr
ASSUME
$PARCELS \subseteq EPARCELS$
$\land PARCELS \neq EPARCELS$
$\land PARCELS \neq \{\ \}$
$\land BASKETS \neq \{\ \}$
$\land noBaskets \notin BASKETS$
$\land adr \in [PARCELS \rightarrow BASKETS]$

VARIABLES
stored, channel, next, current, sorted, ready_to_sort, adr_next
INVARIANT
$stored \in PARCELS \nrightarrow BASKETS \land$
$channel \in BASKETS \land$
$next \in EPARCELS \land$
$current \in EPARCELS \land$
$ready\_to\_sort \in BOOL \land$
$sorted \subseteq PARCELS \land$
$adr\_next \in BASKETS$
DEFINITIONS
$TO\_SORT == PARCELS - sorted$;
$UNDEF == EPARCELS - PARCELS$

$\Rightarrow$

VARIABLES
stored, channel, next, current, sorted, ready_to_sort, adr_next
TypeInvariant =
$\land stored \in [PARCELS \rightarrow BASKETS \cup \{\text{noBaskets}\}]$
$\land channel \in BASKETS \cup \{noBaskets\}$
$\land next \in EPARCELS$
$\land current \in EPARCELS$
$\land ready\_to\_sort \in BOOLEAN$
$\land sorted \subseteq PARCELS$
$\land adr\_next \land$
$TO\_SORT = PARCELS \setminus sorted$
$UNDEF = EPARCELS \setminus PARCELS$

INITIALISATION
$stored := \{\} || channel :\in BASKETS ||$
$next :\in UNDEF || current :\in UNDEF ||$
$sorted := \{\} || ready\_to\_sort := FALSE ||$
$adr\_next :\in BASKETS$

$\Rightarrow$

Init =
$\land stored = [p \in PARCELS \mapsto noBaskets\ ]$
$\land channel = noBaskets \land next \in UNDEF$
$\land current \in UNDEF$
$\land sorted = \{\ \} \land ready\_to\_sort = FALSE$
$\land adr\_next = noBaskets$

Select_parcel = **ANY** *p* **Where**
$p \in TO\_SORT \land next \in UNDEF$
THEN
$adr\_next := adr(p) || next := p$
END ;

$\Rightarrow$

Select_parcel =
$\land next \in UNDEF$
$\land \exists p \in UNSORTED : \land next' = p$
$\land UNCHANGED\langle channel, sorted, stored, next, current, ready\_to\_sort\rangle$

Set_channel = SELECT
current ∈ UNDEF∧
next ∈ PARCELS ∧
¬ ready_to_sort
THEN
channel := adr_next ||
ready_to_sort := TRUE
END ;

⇒

Set_channel =
∧ current ∈ UNDEF
∧ next ∈ PARCELS
∧ ¬ ready_to_sort
∧ channel'= adr_next
∧ ready_to_sort'= TRUE
∧ UNCHANGED⟨stored, sorted, next, current, ready_to_sort⟩

Release = SELECT
current = ∈ UNDEF ∧
next ∈ PARCELS ∧ ready_to_sort
THEN
current := next || next :∈ UNDEF ||
ready_to_sort := FALSE
END ;

⇒

Release =
∧ current ∈ UNDEF
∧ next ∈ PARCELS
∧ ready_to_sort
∧ current' = next
∧ next' ∈ UNDEF
∧ ready_to_sort' = FALSE
∧ UNCHANGED⟨channel,stored,sorted, next,current,adr_next⟩

Cross_sorting = SELECT
current ∈ PARCELS
THEN
stored(current) := channel ||
sorted := sorted ∪ { current} ||
current :∈ UNDEF
END ;

⇒

Cross_sorting =
∧ current ∈ PARCELS
∧ stored'=[stored EXCEPT ![current] = channel]
∧ sorted'=sorted ∪ {current}
∧ current' ∈ UNDEF
∧ UNCHANGED⟨channel, ready_to_sort,adr_next⟩

L'invariant à vérifier par le module TLA$^+$ est le suivant :

Invariant1 ==
∧ ready_to_sort ⟹ channel = adr_next
∧ current ∈ PARCELS ⟹ ¬channel = adr[current]
∧ next ∈ PARCELS ⟹ adr_next = adr[next]
∧ ∀ p ∈ PARCELS : p ∈ *DOMAIN stored ⟹* stored[p] = adr[p]

La spécification finale en TLA$^+$ s'exprime de la manière suivante :

Spec1 == *Init1* ∧ □[Next1]$_{tr1vars}$∧ *Fairness1* ∧ *Liveness1* Où

- *Next1*== *Select_parcel* ∨ *Set_channel* ∨ *Release* ∨ *Cross_sorting*
- *tr1vars* == ⟨*channel, current, next, stored, sorted, ready_to_sort,adr_next*⟩
- *Fairness1* == $WF_{tr1vars}$(Select_parcel)∧ $WF_{tr1vars}$(Set_channel)
  ∧ $WF_{tr1vars}$(Release)∧ $WF_{tr1vars}$(Cross_sorting)

Liveness1 ==
∧∀p ∈ TO_SORT ⇝ stored[p] = adr[p]
∧∀p ∈ TO_SORT ⇝ stored[p] ∈ BASKETS

Étape 5 : Vérification des propriétés de vivacité par le model checker TLC.
Pour la validation du module TLA$^+$, nous avons besoin des modules de configuration contenant les propriétés à vérifier par le modèle checker (module MCParcel_routing1.tla) et du module des instances finies des constantes et des ensembles utilisées par le modèle checker (module MCParcel_sorting1.cfg). Ces modules sont donnés respectivement par les tableaux suivants. TLC permet aussi de vérifier le raffinement de modules. Le résultat du modèle checker TLC est donné à l'annexe C.

MODULE *MCParcel_routing1*

EXTENDS Naturals
VARIABLES channel, next, current, stored, sorted, ready_to_sort
PARCELS == 1..5
PPARCELS == 0..5
noBaskets == 9
BASKETS == 7..14
- *ādr* == [$p \in$ PARCELS $\mapsto$
  CASE p=1 $\rightarrow$ 7
  [] p=2 $\rightarrow$ 10
  [] p=3 $\rightarrow$ 12
  [] p=4 $\rightarrow$ 9
  [] p=5 $\rightarrow$ 14]
INSTANCE Parcel_routing1

MODULE MCParcel_routing1

SPECIFICATION Spec1
INVARIANTS TypeInvariant1 Invariant1
PROPERTIES Liveness1

Satisfaction des invariants.
Les invariants sont satisfaits par une spécification TLA$^+$. Soit I l'ensemble de tous les invariants du module TLA$^+$. La spécification TLA$^+$ satisfait I (Spec $\Rightarrow \Box$ I).

- *Propriété.* Soit M un modèle B dont les variables sont représentées par un vecteur x, les conditions initiales Init(x), la liste des événements est E, l'invariant I(x) et les propriétés de sûreté sont $P_1(x), ..., P_k(x)$. Soit TLA(M) la spécification TLA$^+$ construite à partir de M. Alors TLA(M) $\Rightarrow \Box$I(x) et pour tout i de 1 à k, I(x) $\Rightarrow P_i(x)$.
- *Preuve.* La preuve est évidente, puisqu'elle suppose que le modèle M est validé. L'invariance est donc vérifiée et les propriétés de sûreté sont aussi vérifiées.

Satisfaction des propriétés de vivacité.
Le module TLA$^+$ satisfait les propriétés de vivacité et nous avons *Spec1* $\Longrightarrow$ *Liveness1.*

*** Deuxième raffinement**

- *Étape 1 : Modèle B.*

Ce raffinement consiste à introduire un modèle de la trieuse plus proche de la réalité. Le

dispositif concret est formé d'aiguillages interconnectés par des conduits. Un aiguillage a une entrée et deux sorties, chaque sortie est reliée par un conduit à l'entrée d'un autre aiguillage à l'exception des aiguillages dont les sorties débouchent sur des bacs de tri.

Un paquet entré dans un aiguillage en sort par une des sorties ce qui lui permet ensuite d'atteindre le nœud auquel la sortie est interconnectée et ainsi de suite jusqu'au bac de destination. Les paquets entrent dans la trieuse par un seul aiguillage. Le dispositif de tri peut être modélisé par un arbre binaire dont les nœuds correspondent aux aiguillages et les arcs aux conduits reliant les nœuds. Cette structure est équilibrée, c'est-à-dire que toutes les branches de l'arbre sont d'égale longueur.

- *Constantes du modèle.* Pour représenter l'arbre binaire, nous introduisons l'ensemble *switches* pour représenter les nœuds intermédiaires. Un paquet doit passer par une suite de nœuds qui dépend de l'adresse du bac destination.

  Chaque nœud de l'arbre appartenant à *switches* possède deux sorties représentées par les fonctions constantes *left_n* et *right_n*. La fonction constante *access* associe à tout nœud le sous ensemble des bacs qu'il permet d'atteindre.

  CONSTANTS
  rac, access, switches, right_n, left_n, depth, T
  PROPERTIES
  $rac \in NODES \wedge rac \notin BASKETS \wedge switches = NODES - BASKETS \wedge$
  $BASKETS \subset NODES \wedge access \in switches \rightarrow POW1(BASKETS) \wedge$
  $right\_n \in switches \rightarrow NODES \wedge left\_n \in switches \rightarrow NODES \wedge$

- *Nouvelles variables.* Dans ce modèle, nous ajoutons les variables suivantes :
  - *exit* qui à chaque nœud, indique le prochain nœud visité par le paquet qui le traverse,
  - *nd* désigne le nœud courant,
  - *lng* désigne la longueur de l'arbre à partir de la racine,

  La variable *channel* du modèle précédent correspond, dans le modèle concret, à la suite des nœuds qu'un paquet doit traverser pour atteindre son bac de destination. Cette variable est raffinée par la variable *exit*.

  VARIABLES
  ...
  exit, nd, lng /* *Nouvelles variables* */
  INVARIANT
  $exit \in switches \nrightarrow NODES \wedge nd \in NODES \wedge lng \in 0..T$

- *Initialisation.* Nous initialisons les variables du raffinement aux valeurs suivantes :

  INITIALIZATION
  ...
  $nd := rac \parallel exit := \varnothing \parallel lng := 0$

- *Événements.* Dans ce raffinement, nous raffinons les anciens événements et nous ajoutons les événements *Set_switch* et *Cross_node*.

  L'événement *Select_parcel* est raffiné en renforçant sa garde par la condition $nd = rac$.

Select_parcel=
ANY *p*Where $p \in TO\_SORT \wedge$
$next \in UNDEF \wedge current \in UNDEF \wedge$
$ready\_to\_sort = FALSE \wedge exit = \varnothing$
THEN
adr_next := adr(p) || *next* := *p* || *nd* := *rac* || *lng* := *0*
END ;

L'événement *Set_switch* permet de calculer le prochain nœud à visiter par un paquet courant. Il raffine l'événement *Set_channel*. Le choix du prochain nœud à visiter dépend des valeurs de *left_n* et *right_n* pour le nœud courant. Si le bac de destination (*adr_next*) appartient à la valeur *access(right_n(nd))*, alors le prochain nœud à visiter est le fils droit *exit(nd) := right_n(nd)*, sinon le fils gauche.

Set_switch=
ANY *node WHERE node* $\in \{\text{right\_n(nd)}, \text{left\_n(nd)}\} \wedge$
$nd \in switches \wedge node \notin BASKETS \wedge next \in PARCELS \wedge$
$lng < T\text{-}1 \wedge current \in UNDEF \wedge$
$adr\_next \in access(node) \wedge ready\_to\_sort = FALSE$
THEN
exit(nd) := node || *nd := node* || *lng := lng+1*
END ;

L'événement *Set_channel* est raffiné et la variable *channel* est raffinée par la variable *exit*, et nous devons montrer l'invariant de collage : $\text{iterate}(\text{exit}, \text{T})(rac) = channel$, où T correspond à la profondeur de l'arbre.

Set_channel=
ANY *node*WHERE $node \in \{\text{right\_n(nd)}, \text{left\_n(nd)}\} \wedge$
$nd \in switches \wedge node \in BASKETS \wedge$
$adr\_next = node \wedge next \in PARCELS \wedge$
$current \in UNDEF \wedge ready\_to\_sort = FALSE \wedge lng = T\text{-}1$
THEN
exit(nd) := adr_next || *nd := adr_next* ||
ready_to_sort := TRUE || *lng := lng+1*
END ;

L'événement Release est raffiné comme suit :

Release=
SELECT $next \in PARCELS \wedge current \in UNDEF \wedge$
$ready\_to\_sort = TRUE \wedge nd \in BASKETS$
THEN
current := next || *next* : $\in$ *UNDEF* || *nd := rac* ||
ready_to_sort := FALSE|| *lng := 0*
END ;

Dans le modèle précédent, le passage du paquet courant est modélisé par l'événement *Cross_sorting*. Dans ce raffinement, nous considérons le passage du paquet par différents nœuds. Nous introduisons l'événement *Cross_node* pour déterminer le prochain nœud à visiter.

Cross_node=
SELECT *current* ∈ *PARCELS* ∧
exit(nd) ∉ *BASKETS* ∧ *lng* < *T-1*
THEN
nd := *exit(nd)* || *lng* := *lng+1*
END ;

L'événement *Cross_sorting* est raffiné par la suite d'occurrences de l'événement *Cross_node* s'exécutant tant que le paquet n'est pas arrivé à destination. Le positionnement des aiguillages réalise, à partir de la racine, un conduit qui aboutit à un bac de sortie et qui raffine la variable *channel* du modèle précédent.

Cross_sorting=
SELECT *current* ∈ *PARCELS* ∧
exit(nd) ∈ *BASKETS*∧ *lng* = *T-1*
THEN
stored(current) := *exit(nd)* ||
sorted := *sorted* ∪ { *current*} ||
current : ∈ *UNDEF*|| *lng* := *lng+1*
END ;

Dans ce modèle, nous avons retiré la variable *channel* et nous l'avons remplacée par la variable *exit*. Pour ce faire, nous avons à définir un invariant de collage reliant ces deux variables. Ces invariants sont les suivants :

/* Invariants de collage */
(*next* ∈ *PARCELS* ∧ *current* ∉ *PARCELS* ∧ *nd* ≠ *rac* ∧ *nd* ∈ *switches* ∧
(*right_n(nd)* ∈ *BASKETS* ∨ *left_n(nd)* ∈ *BASKETS*)
⇒ *channel* = *exit (nd)*) ∧
(*current* ∈ *PARCELS* ∧ *nd* ≠ *rac* ∧ *exit(nd)* ∈ *BASKETS*
⇒ *channel* = *exit(nd)*) ∧
(*exit* ≠ ∅ ⇒ *iterate(exit,T)(rac)*=*channel)*

Le modèle a été testé par le model checker ProB pour simuler son comportement et vérifier l'invariant.

Le modèle B du deuxième raffinement est donné en annexe B.

- *Étape 2 : Preuve.*
Le modèle est vérifié par le prouveur, tous les invariants sont préservés et toutes les obligations de preuve ainsi que celles du raffinement sont vérifiées de manière automatique et interactive. Les statistiques sur les preuves sont résumées dans le tableau de la figure 8.1.

- *Étape 3 : Modèle B temporel.*
Nous souhaitons exprimer dans ce modèle des propriétés dynamiques. Pour ce faire, nous commençons par donner des hypothèses d'équité sur les événements. Cette propriété est décrite par la formule :

WF(Select_parcel) ∧ *WF(Set_switch)* ∧ *WF(Set_channel)* ∧ *WF(Release)* ∧
WF(Cross_node) ∧ *WF(Cross_sorting).*

Nous souhaitons que le modèle vérifie la propriété suivante :
Tout paquet entré finit par atteindre le bac de destination. Cette propriété est raffinée

comme suit :
$\forall\ p.(p \in TO\_SORT \wedge (right\_n(nd) \in BASKETS\ or\ left\_n(nd) \in BASKETS) \rightsquigarrow stored(p) = exit(nd).$

- *Étape 4 : Transformation du modèle B temporel vers un module TLA$^+$.*
Le module TLA$^+$ obtenu après la transformation du modèle B temporel est donné à l'annexe C.

- *Étape 5 : Vérification des propriétés de vivacité par le model checker TLC .*
Nous vérifions les propriétés de vivacité par le model checker TLC sur des systèmes finis. Nous trouvons les fichiers de configuration et le fichier des instances finies à l'annexe C. TLC vérifie aussi le raffinement des modèles.

*** Troisième raffinement.**

- *Étape 1 : Modèle B.*
Dans ce raffinement, nous ajoutons les événements *Set_gate_right*, *Set_gate_left*, *Cross_right* et *Cross_left*.

- *Constantes du modèle.*
Nous ajoutons dans ce nouveau modèle l'ensemble OUT qui modélise les sens d'ouverture de la porte.

SETS $OUT = \{\mathrm{L}, \mathrm{R}, \mathrm{Indef}\}$

- *Variables du modèle.*
Dans ce raffinement, nous ajoutons les détails sur les nœuds. Chaque nœud possède un capteur d'entrée, deux capteurs de sortie et une porte. Le capteur d'entrée est modélisé par une variable booléenne *in*. Elle est initialement fausse, mise à vrai par le passage d'un paquet et remise à faux par le programme qui l'observe. Les capteurs de sortie sont représentés par les variables *out_R* et *out_L*. Ces variables valent *true* quand elles détectent le passage d'un paquet et comme on ne peut avoir deux sorties à la fois, les capteurs de sortie vérifient $\neg\ (out\_R \wedge out\_L)$. La porte est modélisée par la variable *gate*. Cette variable est une fonction qui associe à chaque porte le sens d'ouverture à droite *(R)* ou à gauche *(L)*.

VARIABLES ... *in*, *out_l* , *out_r* , *gate* INVARIANT $in \in NODES \rightarrow BOOL \wedge out\_r \in switches \rightarrow BOOL \wedge$ $out\_l \in switches \rightarrow BOOL \wedge$ $gate \in switches \rightarrow OUT \wedge dom(in) = NODES \wedge$ $\forall\ i\ .(i \in switches \Rightarrow \neg\ (out\_r(i){=}TRUE \wedge out\_l(i){=}TRUE))$

L'initialisation des variables est comme suit :

INITIALIZATION

...

in := NODES * {FALSE}|| *out_r := switches* * {FALSE}||
out_l := switches * {FALSE}|| *gate := NODES* * {Indef}

- *Événements du modèle.*
Nous raffinons les anciens événements en renforçant leur garde par des prédicats définis sur les nouvelles variables.

Select_parcel =
ANY *p* Where *p* ∈ *TO_SORT* ∧
next ∈ *UNDEF* ∧ *current* ∈ *UNDEF* ∧
ready_to_sort = FALSE ∧ *exit* = ∅
THEN
adr_next := adr(p) || *next := p* || *in(nd) := TRUE*
nd := rac || *lng := 0*
END ;

Les événements *Set_gate_right* et *Set_gate_left* modélisent respectivement, le positionnement de la porte des nœuds pour permettre la sortie vers la droite (en fermant la sortie vers la droite ou inversement).

Set_gate_right=
SELECT *next* ∈ *PARCELS* ∧
((right_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(right_n(nd))) or*
(nd ∈ *switches* ∧ *right_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := R
END ;

Set_gate_left=
SELECT *next* ∈ *PARCELS* ∧
((left_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(left_n(nd))) or*
(nd ∈ *switches* ∧ *left_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := L
END ;

Set_switch=
ANY *node, gate_nd* WHERE *node* ∈ {right_n(nd), left_n(nd)}∧
nd ∈ *switches* ∧ *gate_nd* ∈ {R, L} ∧ *node* ∉ *BASKETS* ∧
next ∈ *PARCELS* ∧ *current* ∈ *UNDEF* ∧ *lng < T-1* ∧
adr_next ∈ *access(node)* ∧ *ready_to_sort = FALSE* ∧
in(nd) = TRUE || ∧ *gate(nd) = gate_nd*
THEN
exit(nd) := node || *nd := node* || *lng := lng+1* || *in(node) := TRUE*
END ;

Set_channel=
ANY *node*WHERE *node* $\in$ {right_n(nd), left_n(nd)}$\wedge$
nd $\in$ *switches* $\wedge$ *node* $\in$ *BASKETS* $\wedge$
adr_next = *node* $\wedge$ *next* $\in$ *PARCELS* $\wedge$ *lng* = *T-1* $\wedge$
current $\in$ *UNDEF* $\wedge$ *ready_to_sort* = *FALSE* $\wedge$ *gate(nd)* $\neq$ *Indef*
THEN
exit(nd) := *adr_next* || *nd* := *adr_next* || *lng* := *lng+1* ||
ready_to_sort := *TRUE* || *in(node)* := *TRUE*
END ;

Release=
SELECT *next* $\in$ *PARCELS* $\wedge$ *current* $\in$ *UNDEF* $\wedge$
ready_to_sort = *TRUE* $\wedge$ *nd* $\in$ *BASKETS*
THEN
current := *next* || *next* : $\in$ *UNDEF* || *ready_to_sort* := *FALSE*
nd := *rac* || *lng* := *0* || *in(nd)* := *TRUE*
END ;

Les événements *Cross_right* et *Cross_left* modélisent la traversée du paquet à travers les nœuds en tenant compte du sens d'ouverture des portes.

Nous avons modélisé un type de routage où nous préparons les aiguillages avant de libérer le paquet retenu dans le sas, nous exécutons successivement les événements *Select_parcel*, puis une suite de *Set_gate_right* ou *Set_gate_left* suivi de *Set_switch*. Une fois tout préparé en avance, nous exécutons successivement les événements suivants : *Release*, une suite de *Cross_right* ou *Cross_left* suivi de *Cross_node* puis *Cross_sorting*.

Cross_right =
SELECT *current* $\in$ *PARCELS* $\wedge$ *nd* $\in$ *switches* $\wedge$
in(nd) = *TRUE* $\wedge$ *gate(nd)*= *R*
THEN
out_l(nd) := *FALSE* || *out_r(nd)* := *TRUE* || *in(nd)* := *FALSE*
END ;

Cross_left =
SELECT
current $\in$ *PARCELS* $\wedge$ *nd* $\in$ *switches* $\wedge$
in(nd) = *TRUE* $\wedge$ *gate(nd)*= *L*
THEN
out_l(nd) := *TRUE* || *out_r(nd)* := *FALSE* || *in(nd)* := *FALSE*
END ;

Cross_node =
SELECT
current $\in$ *PARCELS* $\wedge$ *exit(nd)* $\notin$ *BASKETS* $\wedge$ *lng* $<$ *T-1* $\wedge$
(out_r(nd) = *TRUE* $\vee$ *out_l(nd)* = *TRUE)*
THEN
nd := *exit(nd)* || *lng* := *lng+1* || *in(nd)* := *TRUE*
END ;

Cross_sorting =
SELECT
current ∈ *PARCELS* ∧ *exit(nd)* ∈ *BASKETS* ∧
lng = *T-1*∧ (*out_r(nd)* = *TRUE* ∨ *out_l(nd)* = *TRUE)*
THEN
stored(current) := *exit(nd)* || *sorted* := *sorted* ∪ { *current*} ||
current : ∈ *UNDEF* || *lng* := *lng+1* *in(nd)* := *TRUE*
END ;

Nous résumons par la figure 5.12 les différents niveaux de raffinements et l'enchaînement des événements à chaque niveau.

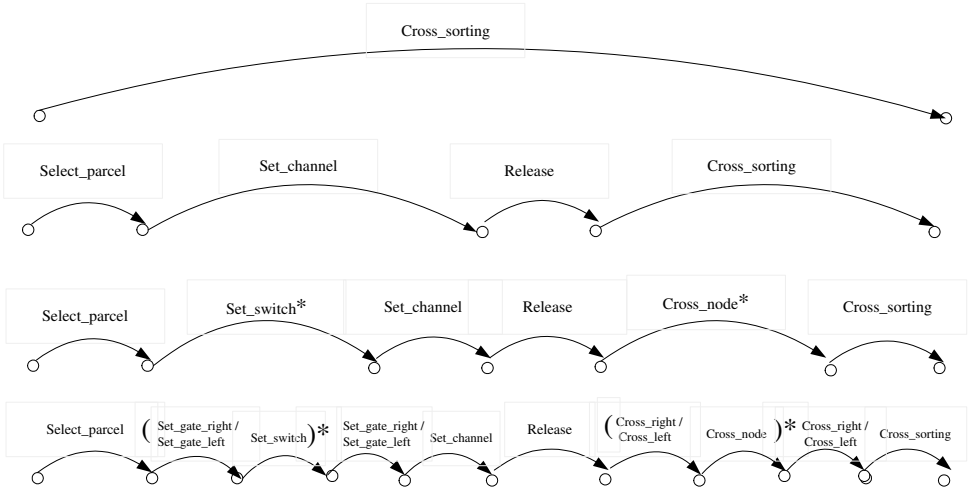


FIG. 5.11 – Séquence d'événements des différents niveaux de raffinement.

Le modèle B complet du troisième raffinement est donné en annexe B.

- *Étape 2 : Preuve.*
Le modèle est prouvé par le prouveur. Les obligations de preuve sont prouvées d'une manière automatique et interactive et le raffinement est vérifié. Nous trouvons les statistiques de preuve dans le tableau de la figure 8.1.

- *Étape 3 : Modèle B temporel.*
Dans le modèle précédent, nous ajoutons les équités sur les événements. La propriété est la suivante :

WF(Select_parcel) ∧ *WF(Set_gate_right)* ∧ *WF(Set_gate_left)* ∧ *WF(Set_switch)* ∧ *WF(Set_channel)* ∧ *WF(Release)* ∧ *WF(Cross_right)* ∧ *WF(Cross_left)* ∧ *WF(Cross_node)* ∧ *WF(Cross_sorting).*

- *Étape 4 : Transformation du modèle B temporel vers un module TLA*$^+$.
Le module TLA$^+$ obtenu après la transformation du modèle B temporel est donné à l'annexe C.

- *Étape 5 : Vérification des propriétés de vivacité.* Nous vérifions les propriétés de vivacité par le model checker TLC sur des systèmes finis. Nous trouvons les fichiers de configuration et le fichier des instances finies à l'annexe C. TLC vérifie aussi le raffinement des modèles.

5.6.3 Statistiques sur les preuves

Chaque pas de raffinement est justifié formellement par des preuves. Les obligations de preuve sont générées automatiquement par les outils. Certaines des preuves sont prouvées automatiquement et d'autres nécessitent une interaction avec l'utilisateur. Les statistiques sur les preuves sont résumées dans le tableau de la figure 8.1.

Le principe de raffinement permet de modéliser le système pas à pas en introduisant les détails au fur et à mesure. Beaucoup de preuves peuvent ainsi être faites à un niveau abstrait. Il est beaucoup plus simple de faire des preuves sur un système contenant peu de variables et en manipulant des données abstraites (ensembles, fonctions...) que sur un système plus concret.

Par ailleurs, la modélisation étape par étape par raffinements successifs permet à la personne en charge de la modélisation d'appréhender le système progressivement, en raffinant seulement une partie du système à chaque pas. Il est plus facile de trouver et corriger une erreur dans un tel système que dans une description bas niveau, noyé par les détails. De plus, posséder un modèle abstrait du système permet une meilleure compréhension de son comportement. Le système aura une meilleure chance d'être compris par une personne que le développeur, ce qui facilitera son intégration et sa réutilisation.

Raffinement	*Nombre d'obligations de preuve*	*Nombre de preuves automatiques*	*Nombre de preuves interactives*
0	4	3	1
1er	8	6	2
2ème	33	27	6
3ème	51	47	4

TAB. 5.10 – Statistiques sur les preuves.

5.6.4 Raffinement des propriétés de vivacité

La condition d'équité à un niveau donné dépend du système global. Si on a à un niveau abstrait, un événement e avec une condition d'équité faible, et à un niveau raffiné cet événement est raffiné en une suite d'événements e_1, e_2, e_3 et e ; alors la condition d'activation de l'événement *e* au niveau raffiné, dépend des autres nouveaux événements et des conditions de leur activation. Donc un événement à équité faible au niveau abstrait peut avoir une condition d'équité forte à un niveau raffiné.

D'autre part, les propriétés de vivacité sont préservées en TLA$^+$. En effet, le raffinement TLA est un moyen de détailler un modèle en s'assurant que les propriétés d'un niveau abstrait sont préservées dans le niveau concret. En TLA$^+$, les conditions d'équité sont des propriétés temporelles et pour prouver le raffinement, il faut vérifier que la spécification concrète implique la spécification abstraite.

Dans la section suivante, nous présentons la notion de diagrammes de prédicats [Cansell *et*

al., 2001a], [Cansell *et al.*, 2001b]. En effet, le model checker TLC travaille sur des instances finies d'un système. Pour la vérification des propriétés de vivacité sur des instances infinies d'un modèle nous utilisons les diagrammes de prédicat et l'outil de développement DIXIT [Fejoz *et al.*, 2005] associé.

5.6.5 Utilisation des diagrammes de prédicats

Dans cette section, nous utilisons les diagrammes de prédicats pour permettre la vérification des propriétés de vivacité sur des systèmes à états infinis et aussi la vérification du raffinement concernant les modèles B temporels. En effet, un diagramme de prédicats est une abstraction d'un système modélisé en TLA$^+$. Il sert à donner une vue abstraite et simple d'un système plus complexe et nous devons montrer que le diagramme de prédicats est une bonne abstraction du système. L'avantage principal de ce diagramme est d'être un graphe fini de prédicats où nous pouvons l'étudier comme un système fini avec des techniques de model checking. Aussi, nous pouvons développer des diagrammes de prédicats en utilisant une relation de raffinement dont l'objectif est de produire un système conforme à cette abstraction.

Ces diagrammes fournissent un cadre pour raisonner sur les systèmes réactifs, basés sur les abstractions booléennes. Dans ces diagrammes, il y a séparation du raisonnement entre états et transitions d'état à partir du raisonnement sur les comportements, qui est requis pour la preuve des propriétés de vivacité. En particulier, toutes les obligations de preuve concernant les propriétés temporelles sont déclarées en termes de systèmes de transition finis et peuvent être par conséquence vérifiées par model checking.

Première abstraction

Dans l'abstraction, l'état du système est représenté par la variable *stored*. Le diagramme de prédicats présenté par la figure 5.12 décrit une abstraction du système de tri d'un paquet. Le nœud *Init* est le nœud initial. Le nœud *Parcel_in_basket* est interprété par le prédicat *NoParcleStored=FALSE*.

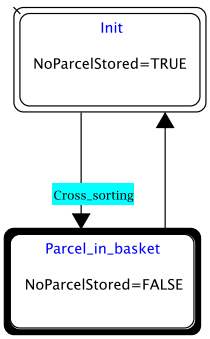


FIG. 5.12 – Première abstraction : diagramme de prédicats du système de tri.

A ce diagramme, nous associons le module TLA$^+$ de la figure 5.13 interprétant les termes et les prédicats du diagramme. *NoParcelStored* est une variable booléenne définie comme suit : $NoParcelStored == stored = [p \in PARCELS \mapsto noBaskets]$. Cette variable vaut TRUE à l'état *init* et FALSE à l'état *Parcel_in_basket*.

```
MODULE Parcel_routing
EXTENDS Naturals
CONSTANTS  EPARCELS, PARCELS, NODES, BASKETS, noBaskets
VARIABLES  stored
ASSUME  adr ∈ [PARCELS → BASKETS] ∧ PARCELS ≠ {} ∧ BASKETS ≠ {}
          ∧ PARCELS ⊆ EPARCELS ∧ PARCELS ≠ EPARCELS ∧ noBaskets ∉ BASKETS
          ∧ BASKETS ⊆ NODES ∧ BASKETS ≠ NODES
-----------------------------------------------------------------------
TypeInvariant ≜  ∧ stored ∈ [PARCELS → BASKETS ∪ {noBaskets}]
-----------------------------------------------------------------------
Init ≜  ∧ stored = [p ∈ PARCELS ↦ noBaskets]
Cross_sorting ≜  ∧ ∃ << p, b >> ∈ PARCELS × BASKETS : p ∉ DOMAIN stored
                 ∧ adr[p] = b
                 ∧ stored′ = [stored EXCEPT ![p] = b]
Next ≜  ∨ Cross_sorting
-----------------------------------------------------------------------
trvars ≜ ⟨stored⟩
Fairness ≜  WF_trvars(cross_sorting)
Spec ≜  Init ∧ □[Next]_trvars ∧ Fairness
Invariant ≜  ∧ ∀ p ∈ PARCELS : p ∈ DOMAIN stored ⇒ stored[p] = adr[p]
-----------------------------------------------------------------------
NoParcelStored == stored = [p ∈ PARCELS ↦ noBaskets]
THEOREM  Spec ⟹ □TypeInvariant
THEOREM  Spec ⟹ □Invariant
```

FIG. 5.13 – Module TLA+ abstrait de la trieuse.

Dans ce modèle, nous n'avons pas de propriété de vivacité à vérifier.

Premier raffinement

Dans le premier raffinement, l'état du système est représenté par six variables : *ready_to_sort, next, current, channel, sorted et stored*. Le diagramme de prédicats présenté par la figure 7.6, décrit le premier raffinement du système de tri d'un paquet. Le nœud *Selected_parcel* est interprété par les prédicats *next* $\in$ *PARCELS* et *adr_next=adr(p)* indiquant respectivement, qu'un paquet est choisi en entrée et que son adresse de destination est mémorisée. Le nœud *Positioned_channel* est interprété par les prédicats *channel=adr_next* et *ready_to_sort=TRUE*, indiquant respectivement, que le canal est positionné à l'adresse de destination et que le système est prêt à libérer le paquet. Le nœud *Gate_opened* interprété par les prédicats *current=next* et *next* $\in$ *UNDEF* indiquant qu'il y a un paquet en cours de tri. Le noeud *Parcel_in_basket* est interprété par les prédicats *NoParcelStored=FALSE* et *sorted=*$\varnothing$.

Le diagramme de prédicats de la figure 7.6 est un raffinement du diagramme de prédicat de la figure 5.12 en éclatant le nœud *Parcel_in_basket* en *Selected_parcel*, *Positioned_channel*, *Gate_opened* et *Parcel_in_basket*.

A ce diagramme, nous associons le module TLA$^+$ du premier raffinement présenté en annexe C, qui interprète les termes et les prédicats du diagramme. Nous trouvons les actions *Select_parcel, Set_channel, Release* et *Cross_sorting* dans le module TLA$^+$ ainsi que le diagramme de prédicats. Ces actions sont nécessaires pour vérifier la conformance entre le diagramme de prédicats et la spécification TLA$^+$ pour la vérification du raffinement ainsi que des propriétés de

vivacité.

Nous faisons l'hypothèse d'équité faible pour les actions *Select_parcel, Set_channel, Release* et *Cross_sorting* et nous devons montrer que tout paquet entré doit fatalement arriver vers le bac de destination. Formellement, la propriété TLA ($\forall p \in TO\_SORT \rightsquigarrow stored[p] = adr[p]$) est prise en compte par le diagramme et a été vérifiée par DIXIT [Fejoz *et al.*, 2005]. En particulier, elle est préservée par chaque raffinement du diagramme.

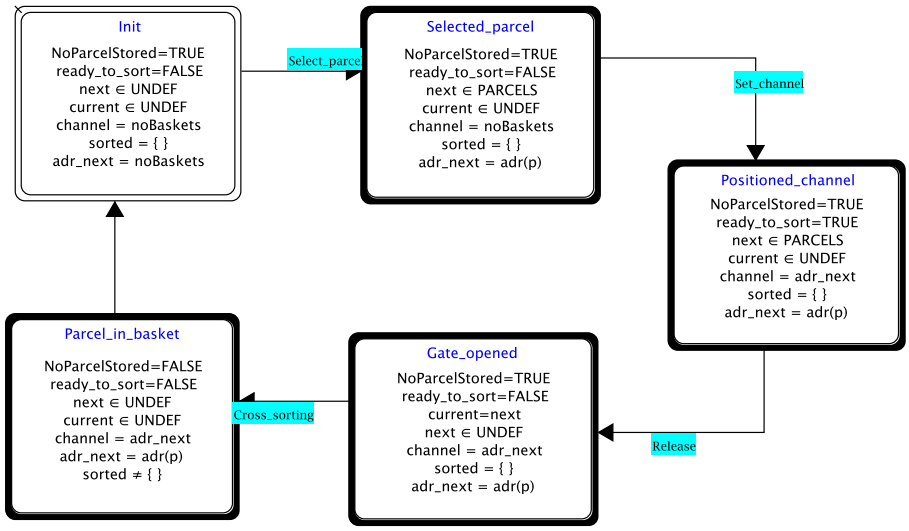


FIG. 5.14 – Premier raffinement : diagramme de prédicats du système de tri.

Les propriétés de vivacité sont exprimées dans une logique temporelle linéaire et peuvent être vérifiées sur un diagramme de prédicats G par la technique de model checking. Pour ce faire, G est considéré comme un système de transitions à états finis définissant un ensemble de comportements avec équité. DIXIT utilise SPIN [Holzmann, 2003] et LWASPIN [Hammer *et al.*, 2005] model checkers.

Deuxième raffinement

Nous introduisons dans ce raffinement de nouveaux états et nous augmentons l'espace d'état en ajoutant les variables *nd*, *pos* et *exit*. Le diagramme de prédicats de la figure 5.15 est le raffinement du diagramme de la figure 7.6 obtenu en éclatant le nœud *Positionned_channel* en *Positionned_channel*, *Path_positionned* et en ajoutant les prédicats concernant les variables *nd, pos* et *exit*. Aussi, le nœud *Gate_opened* est éclaté en deux nœuds : *Gate_opened* et *Crossing_node*. Les nouvelles actions sont *Set_switch* et *Cross_node*.

Le diagramme de prédicats de la figure 5.15 est conforme au module TLA^+ du deuxième raffinement présenté en annexe C.

En utilisant DIXIT, nous pouvons vérifier que le nouveau diagramme raffine le précédent. Premièrement, nous faisons un test structurel qui confirme que les nœuds initiaux du diagramme de raffinement sont reliés aux nœuds initiaux du diagramme raffiné et que les transitions du diagramme de raffinement respectent les transitions du diagramme abstrait. En particulier, la nouvelle transition *Set_switch* raffine une transition de bégaiement à un niveau abstrait. Deuxièmement, des obligations de preuve sont générées pour montrer que les prédicats apparaissant

au niveau des nœuds du raffinement impliquent ceux des nœuds correspondants du diagramme abstrait. Dans notre exemple, ces obligations de preuve sont satisfaites d'une manière triviale parce que nous avons ajouté seulement de nouveaux prédicats. Troisièmement, la technique de model checking garantit que les propriétés d'équité du niveau abstrait sont préservées.

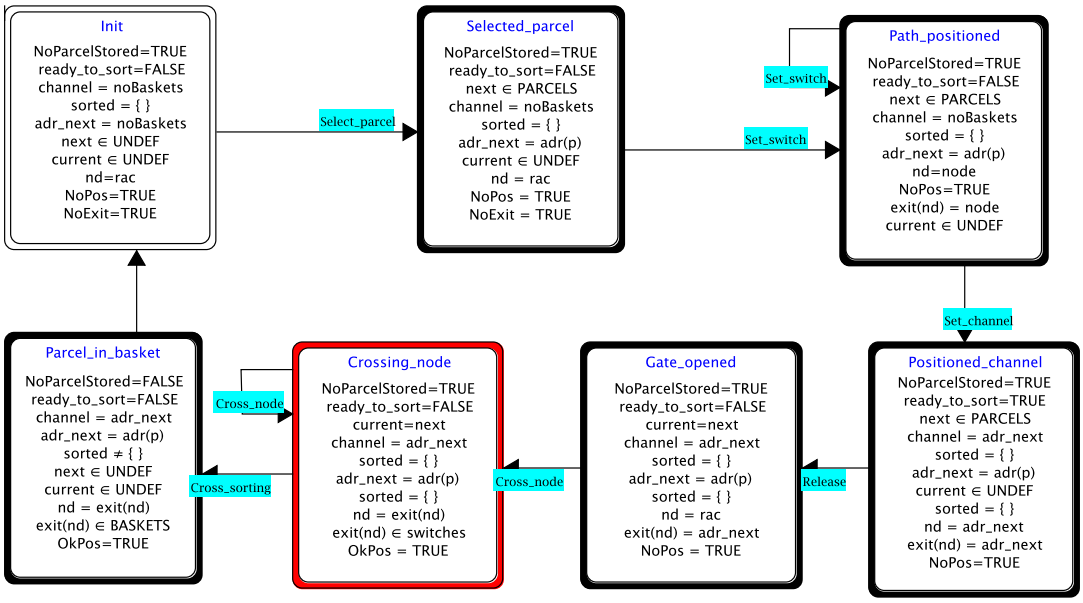


FIG. 5.15 – Deuxième raffinement : diagramme de prédicat du système de tri.

Sur ce modèle, la propriété de vivacité ($\forall$p $\in$ TO_SORT $\leadsto$ stored[p] = adr[p]) est vérifiée par model checking et préservée par chaque raffinement.

Troisième raffinement

Dans ce raffinement, nous introduisons les détecteurs d'entrée et de sortie et les portes au niveau de chaque nœud. Nous obtenons le diagramme de prédicat présenté par la figure 5.16, qui est conforme au module TLA$^+$ du troisième raffinement présenté en annexe C.

Le diagramme de prédicat est obtenu en éclatant le nœud *Path_positionned* en *Path_positionned*, *Gate_pos_right* et *Gate_pos_left* et le nœud *Crossing_node* en *Crossing_node*, *Crossing_right* et *Crossing_left* et en ajoutant des prédicats décrivant les détecteurs d'entrée, de sortie et les portes au niveau de chaque nœud. Les nœuds du diagramme reflètent les différentes phases du système de tri et les arcs représentent les transitions possibles. Par exemple, deux actions sont permises à partir du nœud *Gate_pos_right*. Premièrement, l'action permise est *Set_switch*, qui permet de revenir au nœud *Path_positionned*. La deuxième action permise est *Set_channel* qui permet de transiter vers le nœud *Positioned_channel*.

Nous pouvons aussi vérifier que le nouveau diagramme de prédicat est un raffinement correct du diagramme précédent. Bien que les conditions de raffinement structurelles et l'implication des prédicats soient triviales, le raffinement des conditions d'équité précédentes préserve ces conditions.

En utilisant les diagrammes de prédicat, nous avons pu vérifier les propriétés de vivacité sur des systèmes infinis.

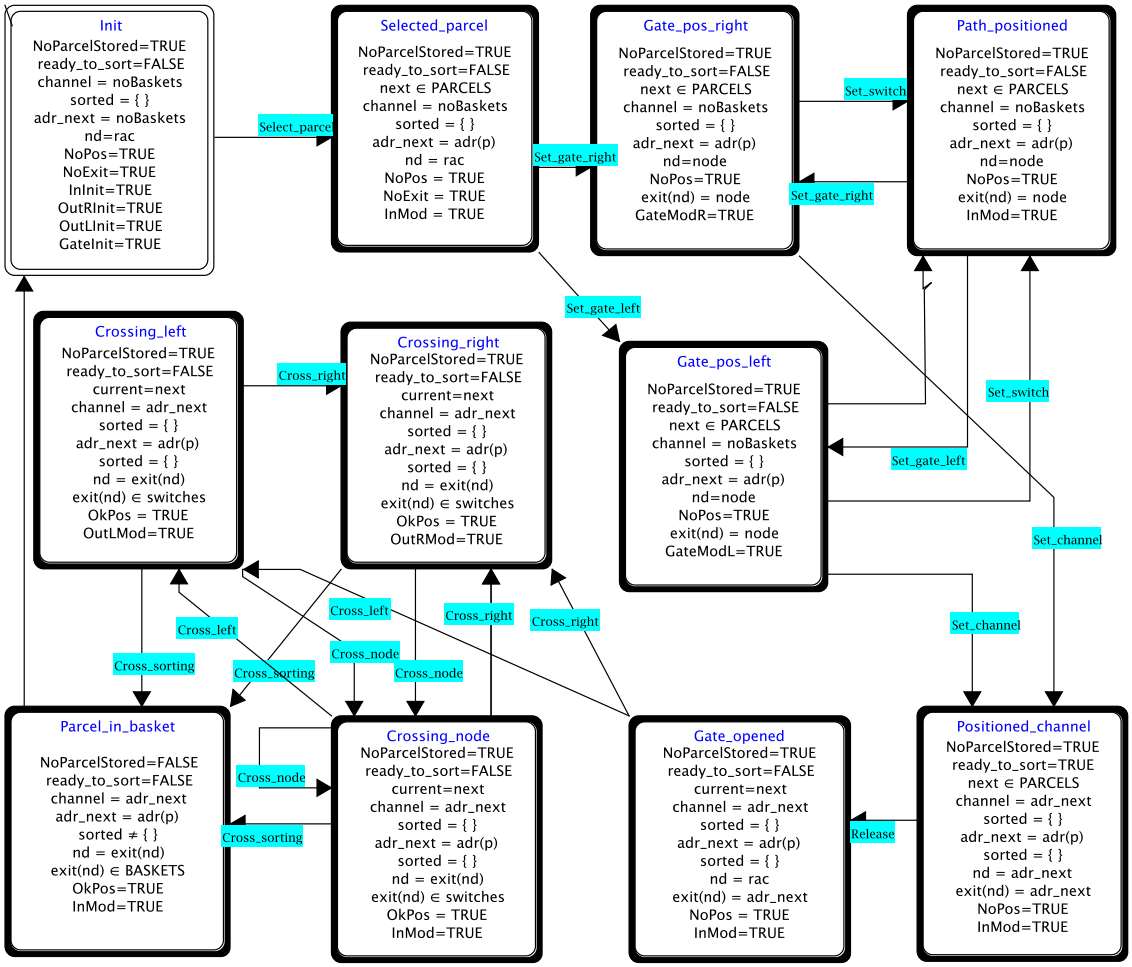


FIG. 5.16 – Troisième raffinement : diagramme de prédicat du système de tri.

5.7 Conclusion

Nous avons présenté dans ce chapitre, une approche qui combine B événementiel et TLA$^+$ pour exprimer et vérifier des propriétés d'invariance et de vivacité sur des systèmes automatisés. Nous profitons ainsi de l'environnement de spécification de B, ainsi que de l'outil de preuve associé pour vérifier des propriétés d'invariance et utiliser l'environnement de TLA$^+$ pour vérifier les propriétés de vivacité qui ne peuvent être facilement exprimées et vérifiées en B. Nous avons proposé dans un premier temps, une extension syntaxique du B événementiel pour prendre en compte les propriétés de vivacité. Ensuite, nous avons donné la sémantique du B événementiel engendrée par l'ajout des nouvelles propriétés. Par la suite, nous avons proposé des règles de preuve pour la vérification de telles propriétés dans l'axiomatique de B. Dans un second temps, nous avons présenté des règles de transformation d'un modèle B vers un module TLA$^+$. Pour automatiser le processus de transformation, nous avons développé le prototype B2TLA$^+$. Enfin, pour la vérification des propriétés de vivacité sur un module TLA$^+$, nous avons utilisé le prouveur de théorèmes Isabelle/TLA et le model checker TLC. L'approche a été testée sur des exemples et a prouvé son efficacité. Nous l'avons testée sur des exemples simples tel que le time-out, sur

des systèmes automatisés tel que le système de tri d'un paquet et sur des systèmes automatisés de production tel que le mixeur que nous allons étudier dans le chapitre 7.

Comme la technique de model checking est limitée à des systèmes à états finis, pour la vérification des propriétés de comportement sur des systèmes à états infinis, nous avons recours à utiliser les diagrammes de prédicats et à l'outil associé.

Chapitre 6

Représentation du temps en B événementiel

6.1 Introduction

Les systèmes réactifs sont des systèmes qui réagissent à des sollicitations provenant de leur environnement en produisant, éventuellement, de nouveaux événements. Les réponses à ces sollicitations, en plus des exigences de correction habituelles, doivent satisfaire des contraintes dites "temps-réel". De ce fait, le bon fonctionnement d'un système temps réel dépend non seulement de la valeur des sorties qu'il produit, mais également de leur instant de production. Les contraintes temporelles font partie des caractéristiques majeures des systèmes temps réel, il est donc nécessaire de les spécifier explicitement. Dans ce qui suit, nous présentons un moyen de prendre en compte des contraintes de temps dans B, que nous appliquons à l'exemple d'un time-out. Dans la section 4, nous présentons une méthode de construction de contrôleurs pour les systèmes automatisés, illustrée par l'exemple du contrôle d'un brûleur à gaz.

6.2 Le temps dans les processus de développement

La description des phénomènes qui se déroulent dans le temps, en particulier l'exécution d'un programme (l'ordre dans lequel les événements apparaissent dans le temps) intéresse les informaticiens. Il existe deux aspects dans le temps tel que nous avons eu l'intuition, un aspect ordinal qui permet d'ordonner des événements dans le temps et un aspect métrique qui permet d'exprimer les durées entre les événements. L'aspect du temps métrique est nécessaire à la spécification de contraintes de temps quantitatives dans les applications temps réel critiques auxquelles nous nous intéressons. Les logiques temporelles ont été utilisées pour la spécification de propriétés de comportement des programmes parallèles comme Unity [K-M. Chandy and J. Misra, 1989].

En Esterel [Berry *et al.*, ; Berry and Cosserat, 1984; Lano *et al.*, 1996a], le temps est défini par les flots des événements que le programme reçoit ou génère. Le temps en ESTEREL a plusieurs bases puisqu'il y a autant d'horloges que de flots d'événements ; il est dit multiforme. Dans les systèmes temporisés, le temps est mesuré grâce à des horloges représentées par des variables à valeurs dans R^+, l'ensemble des réels positifs. Les horloges peuvent être en état de marche ou arrêt ; quand elles sont en marche, elles avancent toutes à la même vitesse.

En B événementiel, le raffinement de systèmes d'événements permet d'introduire la notion

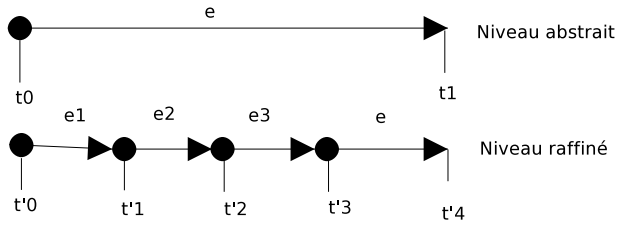


FIG. 6.1 – Raffinement temporel.

de raffinement temporel. L'idée est que par unité de temps, il ne peut se produire qu'un seul événement. Le niveau de détail d'observation du système est donc fonction de la granularité du temps choisie par le spécifieur. Plus le système est abstrait, plus les unités de temps sont grandes. Raffiner le système d'événements consiste alors à "raccourcir" les unités de temps, de telle sorte que des événements cachés auparavant deviennent maintenant observables.

La figure 1 représente un événement e atomique au niveau abstrait qui se décompose en une succession d'événements e_1, e_2, e_3, e au niveau raffiné. Dans cette figure, il n'y a qu'une seule unité de temps (de t_0 à t_1) au niveau abstrait : tout ce qui survient entre ces deux instants est masqué. Au niveau raffiné, avec une granularité du temps plus fine, on peut observer de nouveaux événements e_1 (entre t'_0 et t'_1), e_2 (entre t'_1 et t'_2), et e_3 (entre t'_2 et t'_3). Cette succession de nouveaux événements se termine par un événement e (entre t'_3 et t'_4) qui porte le même nom que l'événement du niveau abstrait. Ainsi, on introduit de nouveaux événements à chaque étape du raffinement. Ils raffinent la substitution qui ne fait rien, c'est à dire la substitution skip.

Le temps n'est pas une primitive de B, mais nous pouvons le représenter par des machines. La vision intuitive qu'on a du temps correspond à une horloge qui émet des "ticks", ces ticks sont suffisamment rapprochés et réguliers pour "compter le temps" de façon satisfaisante. Le temps rythmé par une horloge peut être modélisé par un signal externe représenté par un booléen qui est égal à vrai chaque fois que le signal apparaît.

Les machines B modélisent des systèmes séquentiels et il n'est pas possible de faire "écouler" le temps parallèlement à une autre activité. Une autre difficulté de B événementiel est l'absence d'équité. Une transition gardée par une garde indéfiniment vraie peut être infiniment souvent franchie au détriment des autres transitions qui sont vraies en même temps. Il est intéressant d'expliciter la propriété d'équité en B. La solution consiste à entrelacer les opérations de mise à jour des horloges avec les autres opérations de la machine. Kevin Lano a introduit le temps explicitement en B. Sa technique [Lano, 1996; Lano *et al.*, 1996b; Lano and Haughton, 1996], consiste à "mettre à jour" et à borner la durée de mise-à-jour de l'horloge dans chaque opération car il considère qu'un laps de temps s'est écoulé depuis la dernière mise-à-jour. Nous donnons ci-dessus la machine horloge selon Kevin Lano.

En faisant l'hypothèse qu'un dispositif externe met la variable *tick* à vrai à intervalles réguliers, utiliser *tick*, comme garde d'un compteur de temps, n'est pas une solution satisfaisante. En effet, l'événement peut prendre indéfiniment la main sans se synchroniser avec les autres événements qui sont alors dans l'impossibilité d'observer un temps qui progresse correctement. La vue multiforme du temps de Esterel correspond bien à notre modèle de temps et c'est pourquoi, il nous a semblé intéressant d'étudier le lien entre la programmation Esterel et la modélisation en B évenementiel. Esterel est un langage approprié à la programmation d'un timeout et c'est pourquoi nous avons choisi cet exemple pour mettre en évidence la façon de prendre en compte

MACHINE clock (maxtime)

VARIABLES *now*

INVARIANT *now* $\in$ *TIME*

DEFINITIONS *TIME == 0..maxtime*

INITIALISATION *now := 0*

OPERATIONS

tick(lower, upper) = *PRE lower : TIME* $\wedge$ *upper : TIME* $\wedge$ *lower <= upper* $\wedge$

now + upper <= maxtime THEN now : : now+lower .. now+upper

END ;

END

FIG. 6.2 – Machine clock selon K.Lano.

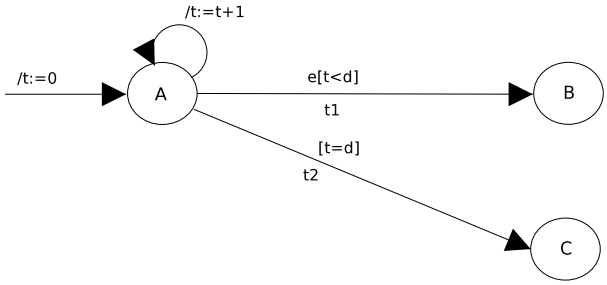


FIG. 6.3 – Statechart représentant le time-out.

la notion de temps.

6.2.1 Etude de cas du timeout avec Esterel et B

Un time-out (chien de garde) est un système réactif qui peut être dans un des trois états A, B, C. Dans l'état A, si un signal de l'environnement apparaît dans un intervalle de d unités de temps, alors le système passe dans l'état B. Si rien ne s'est passé à l'issue de la période d, le système passe à l'état C. Le système est représenté par un statechart à la figure 2.

*** Timeout avec Esterel.** Esterel [Berry *et al.*, ; Berry and Cosserat, 1984] est un langage impératif adapté à la programmation des systèmes réactifs et particulièrement à la programmation de contrôleurs. Ce langage possède des primitives de parallélisation et de communication adaptées aux systèmes réactifs. Il est basé sur l'hypothèse du synchronisme parfait qui suppose que la réaction du système est synchrone avec l'événement qui l'a déclenché. Le programme Esterel correspondant au timeout est le suivant :

```
Module chiendegarde :
input unitedetemps, evt ;
output A,B,C ;
signal termine in
[
await d unitedetemps ;
emit termine
]
||
[
abort
sustain A
when [e or termine ] ;
present termine then
[
sustain C
]
else
[
sustain B
]
end present
[
end signal
end module
```

Notations :
" ;" : opérateur de composition séquentielle de processus,

"||" : opérateur de composition parallèle de processus,

"abort processus .. when signal" : préemption forte, le processus est abandonné dès que le signal apparaît,

"sustain" : émission continue d'un signal,

"present" : détection d'un signal à un moment donné,

"await" : attendre l'occurrence d'un signal,

"emit" : émission d'un signal discret.

Dans ce modèle, deux processus s'exécutent en parallèle. Le premier gère l'horloge et le deuxième émet en continu l'événement A jusqu'à l'occurrence d'un signal *evt* ou l'émission de l'événement termine quand le système atteint d unités de temps. La figure3 explique les instructions Esterel utilisées pour modéliser le timeout.

*** Timeout en B.** Pour tenir compte du temps en B, on a besoin de modéliser deux processus qui se déroulent en parallèle. Le premier processus est caractérisé par les événements qui concernent la gestion du temps et le deuxième processus concerne les autres événements du système. En fait, nous nous appuyons sur une sémantique du parallélisme qui est l'entrelacement des événements. Avec l'entrelacement, nous gérons en B des horloges "quasi-parallèlement" avec les événements du système. Ces horloges doivent s'incrémenter en alternance avec les changements d'état et on assure qu'un événement du premier processus est suivi d'un événement du deuxième.

Le parallélisme "||" d'Esterel est traduit en B par l'alternance d'un processus à l'autre pour assurer l'équité. Le timeout exprimé en B va imposer l'alternance entre l'horloge et les autres événements du système ; le contrôle est donné alternativement à l'horloge et aux autres événements du système. La modélisation du timeout avec B, qui découle de celle de Esterel est la suivante :

- ***Modèle du système : ensembles de base, constantes et variables***
  Nous appelons *Etats*, l'ensemble qui comprend les trois états du système A, B, C et $CTRL$, l'ensemble contenant le contrôle qui est géré soit par l'horloge (*clock*), soit par le système (*sys*). d est une constante qui désigne la durée maximale à passer dans l'état A. Les

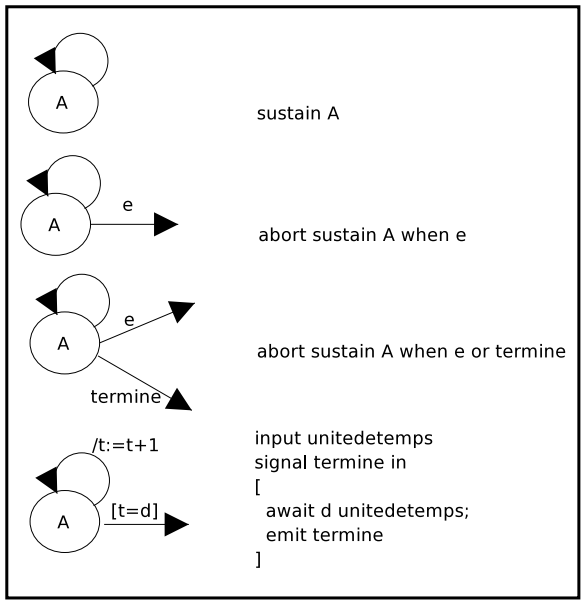


FIG. 6.4 – Explication des instructions Esterel.

variables *etat_sys*, *t*, *evt*, *ctrl* désignent respectivement l'état du système, le compteur du temps, l'occurrence d'un événement et la prise du contrôle. La variable *ctrl* est utilisée pour assurer l'alternance entre les événements qui font écouler le temps et les autres événements du système.

SETS
$Etats = \{A, B, C\}$
$CTRL = \{clock, sys\}$

CONSTANTS
d
PROPERTIES
$d = 10$
VARIABLES
$etat\_sys$, t, evt, $ctrl$

INITIALISATION
$etat\_sys := \{0 \mapsto A\}$
$t := 0$
$evt :=$ FALSE
$ctrl := sys$

- ***Les invariants du système :***
Les invariants suivants sont des invariants de typage. L'état du système est représenté par la fonction partielle *etat_sys* des entiers dans l'ensemble Etats où les entiers représentent les instants.

$etat\_sys \in NAT \nrightarrow Etats$
$dom(etat\_sys) \subseteq 0 \,..\, d{+}1$
$t \in \mathbb{N}$
$evt \in BOOL$
$t \leq d$
$ctrl \in CTRL$

L'invariant P1 exprime que le système reste dans le même état aussi longtemps que *evt* n'est pas arrivé et que le délai d n'est pas échu.

P1 : $(etat\_sys(t) = A \wedge ctrl = clock \wedge evt = \text{FALSE} \wedge t < d \implies etat\_sys(t{+}1) = A)$

L'invariant P2 exprime que s'il y a occurrence de l'événement *evt* alors que le système est à l'état A, que le contrôle est à *clock* et que le compteur t n'a pas atteint la valeur d'échéance d, alors le système est à l'état B à l'instant suivant.

P2 : $(etat\_sys(t) = A \wedge ctrl = clock \wedge evt = \text{TRUE} \wedge t < d \implies etat\_sys(t{+}1) = B)$

L'invariant P3 exprime que si l'échéance est arrivée sans qu'un signal ne soit reçu alors le système passe dans l'état C.

P3 : $(etat\_sys(t) = A \wedge ctrl = clock \wedge evt = \text{FALSE} \wedge t = d \implies etat\_sys(t{+}1) = C)$

- ***Événements :***

L'événement *AtoA* exprime dans quelles conditions le système reste dans l'état A (le contrôle est maintenu par le système de contrôle, pas de reception de l'événement *evt* et la valeur de t n' a pas atteint d unités de temps). L'événement *AtoB* exprime la condition de transit de l'état A vers B quand on a l'occurrence de l'événement *evt*. L'événement *AtoC* exprime la condition de transit de l'état A vers C.

AtoA = SELECT
$etat\_sys(t) = A \wedge$
$ctrl = sys \wedge$
$evt = \text{FALSE} \wedge$
$t < d$
THEN
etat_sys(t + 1) := A ||
$ctrl := clock$
END ;

AtoB = SELECT
$etat\_sys(t) = A \wedge$
$ctrl = sys \wedge$
$evt = \text{TRUE} \wedge$
$t < d$
THEN
etat_sys($t{+}1$) := B
END ;

AtoC = SELECT
$etat\_sys(t) = A \wedge$
$ctrl = sys \wedge$
$evt = \text{FALSE} \wedge$
$t = d$
THEN
etat_sys($t{+}1$) := C
END ;

L'événement *Avance_clock* maintient le contrôle quand *ctrl*= *clock*. Il permet d'avancer le temps et de mettre-à-jour l'horloge et permet aussi de donner une valeur aléatoire à *evt*.

avance_clock= SELECT
$ctrl = clock$
THEN
evt : : { TRUE, FALSE } ||
$t := t{+}1$ ||
$ctrl := sys$
END

- ***Preuves :***

Le modèle a été prouvé par Click_n_prove et les invariants sont préservés par les événements.

6.3 Spécification de systèmes automatisés

Notre objectif est de construire un contrôleur pour un dispositif physique agissant sur un environnement donné. Le problème peut être énoncé comme suit : étant donné un dispositif physique dont le comportement est observable au travers d'un certain nombre de variables et sur lequel il est possible d'agir au moyen d'effecteurs, il s'agit de concevoir un contrôleur ayant pour objectif de maintenir des propriétés sur les variables de l'environnement. La méthode de modélisation que nous proposons consiste à :

1. Construire un modèle B de la partie opérative,
2. Valider le modèle au moyen de l'animateur de l'Atelier B. Cette animation permet de valider une spécification en visualisant l'évolution des variables et des propriétés lors de l'appel des événements et vérifie que les variables prennent bien les valeurs attendues. Par l'animation on s'assure que le comportement du modèle de la partie opérative correspond bien au dispositif réel qu'il modélise,
3. Introduire dans le modèle précédent, des propriétés qui doivent être maintenues à tout instant telles que des propriétés de sûreté, et des contraintes de durée. A ce niveau, nous commençons par construire contrôleur minimal.
4. Raffiner en ajoutant à chaque étape de nouvelles contraintes pour obtenir le système contrôlé spécifié.

Les modèles construits sur cette base sont des modèles fermés modélisant le système et son environnement. La description contient la définition d'un certain nombre de variables décrivant aussi bien les grandeurs physiques que les grandeurs logiques gérées par le système de contrôle ; ceci constitue la définition statique du modèle. Sa dynamique s'exprime au moyen de la définition des diverses transitions (événements) qui montrent comment ces variables peuvent évoluer dans le temps. Pour des propriétés de durée, très vite, on est confronté à la notion de temps, et à sa modélisation. Le temps permet d'ordonner les événements(temps ordinal ou qualitatif) mais il permet aussi de rendre compte de la distance qui sépare deux événements (temps métrique ou quantitatif).

6.4 Etude de cas : Brûleur à gaz

6.4.1 Description informelle du problème

Le processus à contrôler dans cette étude est le système simplifié d'un brûleur à gaz [Lano *et al.*, 1996a] présenté dans la figure 1. Ce système comprend : une vanne d'air pour l'alimentation en air, une vanne de gaz pour l'alimentation en gaz, un détecteur de flamme et un allumeur.

6.4.2 Description des exigences du système automatisé

Le contrôleur d'un brûleur à gaz simplifié doit satisfaire les exigences de sûreté suivantes :

S1 : La vanne de gaz ne doit jamais être ouverte quand la vanne d'air est fermée,

S2 : La flamme ne doit pas apparaître si la vanne d'air est fermée.

Ainsi que des contraintes de durée :

T1 : La vanne de gaz ne doit pas être ouverte pendant plus de 30 secondes en l'absence de flamme,

T2 : Suite à un démarrage non réussi, le système ne doit redémarrer qu'après un laps de temps de 60 secondes.

Pour des raisons d'efficacité, on exige aussi que :
E1 : L'allumeur ne doit pas être actionné si ce n'est pas nécessaire.

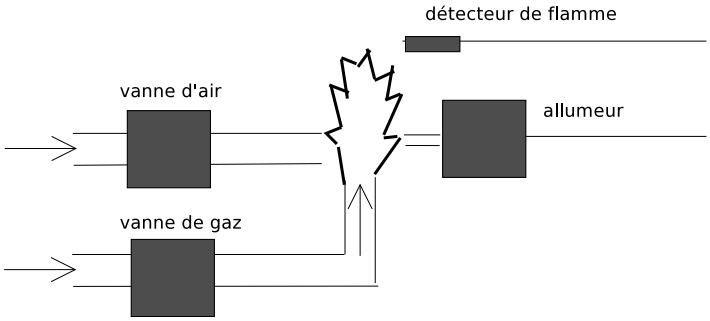


FIG. 6.5 – Les composants du brûleur à gaz.

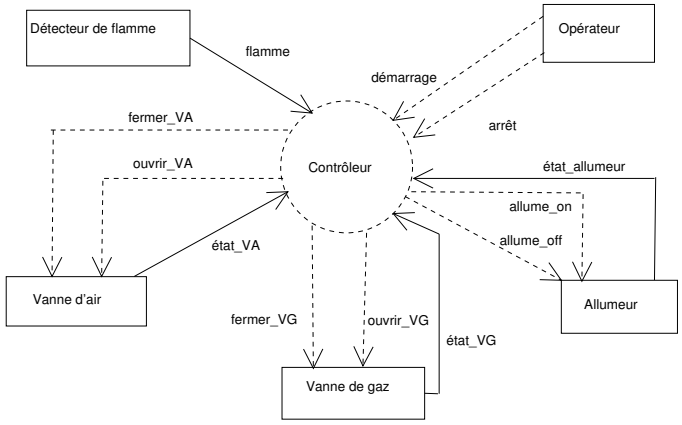


FIG. 6.6 – diagramme de flots de contrôle.

La figure 2 montre un diagramme de flots de contrôle (DFC) qui exprime la relation entre le contrôleur et les différents composants physiques du système. Les traits en gras représentent les informations sur l'état des composants et des variables d'état de l'environnement. Les traits en pointillé représentent les événements générés par le contrôleur pour agir sur les différents composants du procédé physique et de l'environnement.

6.4.3 Modélisation du système automatisé

Nous allons modéliser l'environnement ainsi que le contrôleur du brûleur à gaz en utilisant la démarche proposée dans la section 4.

* Modèle de la partie opérative

Dans ce qui suit, nous modélisons le modèle abstrait bruleur_gaz représentant le comportement des différents composants du dispositif physique et de l'environnement à savoir : la vanne de gaz, la vanne d'air, l'allumeur et la flamme.

Etat_VG et *Etat_VA* représentent respectivement l'ensemble contenant l'état de la vanne de gaz (ouverte ou fermée) et l'ensemble contenant l'état de la vanne d'air (ouverte ou fermée).

MODEl bruleur_gaz
SETS
$Etat\_VG = \{vg\_ouvert, vg\_ferme\}$;
$Etat\_VA = \{va\_ouvert, va\_ferme\}$;
$Etat\_allum = \{on, off\}$;
$Etat\_flamme = \{present, absent\}$
VARIABLES
$etat\_vg$, $etat\_va$, $etat\_allum$, $flamme$
INVARIANT
$etat\_vg \in Etat\_VG$
$etat\_va \in Etat\_VA$
$etat\_allum \in Etat\_allum$
$flamme \in Etat\_flamme$
INITIALISATION
$etat\_vg := vg\_ferme$
$etat\_va := va\_ferme$
$etat\_allum := off$
$flamme := absent$
EVENTS
ouvrir_va = **SELECT** $etat\_va = va\_ferme$ **THEN** $etat\_va := va\_ouvert$ **END** ;
ferme_va = **SELECT** $etat\_va = va\_ouvert$ **THEN** $etat\_va := va\_ferme$ **END** ;
ouvrir_vg = **SELECT** $etat\_vg = vg\_ferme$ **THEN** $etat\_vg := vg\_ouvert$ **END** ;
ferme_vg = **SELECT** $etat\_vg = vg\_ouvert$ **THEN** $etat\_vg := vg\_ferme$ **END** ;
allume_on = **SELECT** $etat\_allum = off$ **THEN** $etat\_allum := on$ **END** ;
allume_off = **SELECT** $etat\_allum = on$ **THEN** $etat\_allum := off$ **END** ;
flamme_devient_presente = **SELECT** $flamme = absent \wedge etat\_vg = vg\_ouvert \wedge etat\_allum = on$ **THEN** $flamme := present$ **END** ;
flamme_devient_absente = **SELECT** $flamme = present \wedge etat\_vg = vg\_ferme$ **THEN** $flamme := absent$ **END**
flamme_s_eteint = **SELECT** $flamme = present$ **THEN** $flamme := absent$ **END**
END

FIG. 6.7 – Modèle de la partie opérative du bruleur à gaz.

Etat_allum et *Etat_flamme* désignent aussi respectivement l'état de l'allumeur qui peut être en marche (*on*) ou en arrêt (*off*) et l'état de la flamme (*on*, *off*). Dans ce modèle, nous n'avons que des invariants de typage.

Les deux événements *ouvrir_va* et *ferme_va* permettent de déclencher respectivement les actions d'ouverture et de fermeture de la vanne d'air. De la même manière, les deux événements *ouvrir_vg* et *ferme_vg* permettent respectivement les actions d'ouverture et de fermeture de la vanne de gaz.

Les événements *allume_on* et et *allume_off* permettent respectivement de déclencher les actions de mise en marche et d'arrêt de l'allumeur. L'événement *flamme-s-eteint* représente une flamme qui s'éteint spontanément sous l'effet de l'environnement.

Les deux événements *flamme_devient_presente* et *flamme_devient_absente* détectent respectivement la présence ou l'absence de la flamme.

Preuve : Tous les invariants de ce modèle sont des invariants de typage, donc le modèle est vérifié automatiquement par le prouveur de l'AtelierB.

Animation du modèle : Après la modélisation de ce dispositif physique et de son environnement, on valide son comportement par l'animateur de l'Atelier B. Avec l'animation, on peut voir évoluer les variables et les propriétés de la spécification et vérifier qu'elles prennent bien les valeurs qui simulent le phénomène physique.

*** Modèle du système de contrôle**

Dans le modèle précédent, nous ajoutons un système de commande minimal représenté par les événements : *demarrage_reussi*, *demarrage_echoue* et *arret*. Les deux premiers événements permettent respectivement l'obtention ou l'échec d'obtention de la flamme suite à un démarrage. L'événement *arret* arrête le système.

demarrage_reussi = SELECT
etat_allum = on ∧
etat_vg = vg_ouvert ∧
flamme = absent
THEN
flamme := present
END;

demarrage_echoue = SELECT
etat_allum = on ∧
etat_vg = vg_ouvert ∧
flamme = absent
THEN
flamme := absent ||
etat_vg := vg_ferme ||
etat_va := va_ferme ||
etat_allum := off
END;

arret = SELECT
etat_vg = vg_ouvert ∧
etat_va = va_ouvert ∧
etat_allum = off
THEN
etat_va := va_ferme ||
etat_vg := vg_ferme
END;

*** Prise en compte de contraintes de sûreté : premier raffinement**

- ***Extension de la spécification informelle :***
  Dans ce raffinement, nous ajoutons les variables, *Operateur* et (*clock*). La variable *Operateur* représente l'ensemble contenant les états de l'opérateur qui peut prendre une des deux valeurs : *ok* (mettre en marche le système) et *ko* (pour commander l'arrêt du système). La variable *CTRL* représente le maintien du contrôle qui va être soit donné à l'horloge (*clock*) soit au système (*sys*). Dans ce raffinement, nous prenons en compte aussi, les deux contraintes S1 et E1 permettant de restreindre l'espace des états du système pour éliminer les états où la vanne d'air est fermée alors que la vanne de gaz est ouverte ou bien les états où la vanne d'air est fermée alors que l'allumeur est en marche.

  S1 : La vanne de gaz ne doit jamais être ouverte quand la vanne d'air est fermée

  E1 : L'allumeur ne doit pas être actionné si ce n'est pas nécessaire

- ***Nouvelles variables et nouveaux invariants :***
  Dans ce deuxième modèle, on ajoute la variable *operateur*. Nous allons avoir un nouvel invariant de typage concernant la variable opérateur. Les deux autres invariants correspondent à la formalisation des deux contraintes citées ci-dessus.

SETS
Operateur = ok, ko
VARIABLES
operateur
INITIALISATION
operateur :=*ko*

INVARIANT
operateur ∈ *Operateur* ∧
(*etat_vg = vg_ouvert* ⟹ *etat_va = va_ouvert*) ∧
(*etat_va = va_ferme* ⟹ *etat_allum = off*)

- ***Événements***
  Nous allons renforcer les gardes des événements précédents de façon à satisfaire les nouveaux invariants.

ferme_va = SELECT
etat_va = va_ouvert ∧
etat_allum = off ∧
etat_vg = vg_ferme
THEN
etat_va := *va_ferme*
END;

ouvrir_vg = SELECT
etat_vg = vg_ferme ∧
etat_va = va_ouvert
THEN
etat_vg := *vg_ouvert*
END;

allume_on = SELECT
etat_allum = off ∧
etat_va = va_ouvert
THEN
etat_allum := *on*
END;

flamme_devient_presente =
SELECT
flamme = absent ∧
etat_allum = on ∧
etat_vg = vg_ouvert
THEN
flamme := *present*
END;

flamme_devient_absente =
SELECT
flamme = present ∧
etat_allum = off ∧
etat_vg = vg_ferme
THEN
flamme := *absent*
END;

arret = SELECT
operateur = ko ∧
etat_vg = vg_ouvert ∧
etat_va = va_ouvert ∧
etat_allum = off
THEN
etat_va := *va_ferme* ||
etat_vg := *vg_ferme*
END

Dans ce raffinement, nous n'allons reprendre que les événements dont les gardes vont être renforcées pour satisfaire les nouveaux invariants et utiliser les nouvelles variables. Les autres événements (*ouvrir_va*, *ferme_vg* et *allume_off*) restent inchangés, alors on ne les réécrit pas au niveau du raffinement.

demarrage_reussi =
SELECT
operateur = ok ∧
etat_allum = on ∧
etat_va = va_ouvert ∧
etat_vg = vg_ouvert ∧
flamme = absent
THEN
flamme := *present*
END;

demarrage_echoue =
SELECT
operateur = ok ∧
etat_allum = on ∧
etat_va = va_ouvert ∧
etat_vg = vg_ouvert ∧
flamme = absent
THEN
flamme := *absent* ||
etat_vg := *vg_ferme* ||
etat_va := *va_ferme* ||
etat_allum := *off*
END;

marche_sys = SELECT
operateur = ko
THEN
operateur := *ok*
END;
arret_sys = SELECT
operateur = ok
THEN
operateur := *ko*
END;

Le nouvel événement est l'événement *marche_sys*, qui permet de mettre-à-jour la valeur de la variable *operateur* de *(ko)* à *(ok)* et de commander la marche du système.

- ***Preuve de raffinement :***
  Les contraintes S1 et E1 représentent les invariants de ce contrôleur. Le modèle est prouvé par Click_n_prove et toutes les obligations de preuve générées sont déchargées par l'outil. Les événements du premier raffinement sont bien les événements du modèle précédent car les actions sont identiques dans le premier modèle et le premier raffinement mais les gardes des événements du raffinement sont plus fortes que celles du premier modèle.

*** Deuxième Raffinement**

- ***Extension de la spécification informelle :***
  Au cours de ce deuxième raffinement, nous allons introduire les deux propriétés de durée T1 et T2. Ces deux contraintes définissent la réponse du système aux actions de démarrage et arrêt de l'opérateur. Les propriétés sont les suivantes :

  T1 : La vanne de gaz ne doit pas être ouverte pendant plus de 30 s en l'absence de flamme

  T2 : Suite à un démarrage échoué, le système ne doit redémarrer qu'après un laps de 60 s.

- ***Nouvelles variables et initialisation :***
  La formalisation passe par l'introduction des nouvelles variables suivantes :

  - *temps_courant* : variable modélisant l'instant courant,
  - *dernier_dem_echoue* : booléen exprimant si le dernier démarrage a échoué ou non,
  - *instant_dernier_dem* : entier modélisant l'instant du dernier démarrage,
  - *dem* : booléen permettant de commander le démarrage du système,
  - *dem_suiv* : entier qui permet de donner l'instant du démarrage suivant au cas où on a un démarrage échoué pour respecter la contrainte T2,
  - *ctrl* : variable qui permet le maintien du contrôle.

VARIABLES
temps_courant,
dernier_dem_echoue,
instant_dernier_dem,
ctrl,*dem*,*dem_suiv*
INITIALISATION
temps_courant :=0 ||
dernier_dem_echoue := FALSE ||
instant_dernier_dem := 0 ||
ctrl := *sys* ||
dem := FALSE ||
dem_suiv := 60

INVARIANT
$temps\_courant \in \mathbb{N} \wedge$
$dem\_suiv \in \mathbb{N} \wedge$
$dem \in BOOL \wedge$
$dernier\_dem\_echoue \in BOOL \wedge$
$instant\_dernier\_dem \in \mathbb{N} \wedge$
$ctrl \in CTRL$

La propriété T2 est exprimée par l'invariant suivant :
$$dernier\_dem\_echoue = TRUE \implies dem\_suiv \geq instant\_dernier\_dem + 60$$

- ***Événements :***
  Nous allons modifier les événements suivants pour prendre en compte les nouvelles variables et les nouveaux événements. L'événement *demarrage_reussi* permet d'obtenir de la flamme tout en respectant la contrainte T1 qui énonce qu'il ne faut pas dépasser 30 secondes pour avoir la réponse du système.

demarrage_reussi = SELECT
dem = TRUE ∧
temps_courant ≥ 0 ∧
temps_courant ≤ 30 ∧
flamme = *absent* ∧
ctrl = *sys* ∧
operateur = *ok* ∧
etat_allum = *on* ∧
etat_va = *va_ouvert* ∧
etat_vg = *vg_ouvert*
THEN
dernier_dem_echoue := FALSE ||
flamme := *present*
END ;

demarrage_echoue = SELECT
dem = TRUE ∧
operateur = *ok* ∧
ctrl = *sys* ∧
temps_courant = 30 ∧
flamme = *absent* ∧
etat_allum = *on* ∧
etat_va = *va_ouvert* ∧
etat_vg = *vg_ouvert*
THEN
etat_vg := *vg_ferme* ||
etat_va := *va_ferme* ||
etat_allum := *off* ||
dernier_dem_echoue := TRUE ||
instant_dernier_dem :=*temps_courant* ||
dem_suiv := *temps_courant* + 60
END ;

L'événement *demarrage_echoue* permet de respecter le fait de ne pas avoir de gaz qui s'échappe sans présence de flamme pour une période de plus de 30 secondes (*temps_courant* = 30). L'invariant T2 est également respecté par l'événement. L'événement *demarrage_en_cours* permet de donner le contrôle à l'horloge tant qu'on n'a pas atteint les 30 unités de temps. L'événement *arret* correspond à l'arrêt du système par l'opérateur.

demarrage_en_cours = SELECT
dem = TRUE ∧
operateur = *ok* ∧
temps_courant ≥ 0 ∧
temps_courant < 30 ∧
flamme = *absent* ∧
ctrl = *sys* ∧
etat_va = *va_ouvert* ∧
etat_vg = *vg_ouvert*
THEN
ctrl := *clock END* ;

arret = SELECT
dem = TRUE ∧
operateur = *ko* ∧
etat_vg = *vg_ouvert* ∧
etat_va = *va_ouvert* ∧
etat_allum = *off*
THEN
etat_va := *va_ferme* ||
etat_vg := *vg_ferme* ||
dem := FALSE
END ;

L'événement *re_init* permet de réinitialiser le système lorsqu'il est en arrêt. L'événement *avance_clock* permet de mettre à jour l'horloge du système et d'avancer le temps et de donner le contrôle au système. De cette manière, on respecte les propriétés d'équité.

re_init = SELECT
dem = FALSE ∧
operateur = *ok*
THEN
dem := TRUE ||
dernier_dem_echoue :=**FALSE** ||
temps_courant := 0
END ;

avance_clock = SELECT
ctrl = *clock*
THEN
temps_courant :=*temps_courant*+1 ||
ctrl := *sys*
END ;

- ***Preuves :***

Les événements du deuxième raffinement respectent les deux propriétés T1 et T2 et sont

bien un raffinement du modèle précédent. Les gardes des événements de ce raffinement sont plus fortes que celles du premier raffinement. Toutes les obligations de preuve générées sont déchargées par le prouveur.

6.5 Conclusion

La technique que nous avons proposée pour représenter le temps quantitativement, s'appuie sur la réalisation d'un entrelacement d'un processus qui gère le temps avec les autres événements du système. Cette technique est inspirée de la notion du temps en Esterel et permet de garantir les propriétés d'équité forte. Elle empêche qu'un événement s'exécute indéfiniment souvent.

Nous avons présenté également, une démarche de construction de systèmes automatisés comprenant la partie opérative et un système de contrôle commande en utilisant l'approche par événements de B(B événementiel). Dans une première étape, nous avons construit un modèle de l'environnement et du procédé physique que nous avons validé par l'animateur de l'AtelierB. Le système de contrôle est raffiné de sorte qu'à chaque étape le système contrôlé satisfasse des propriétés de plus en plus fortes. Le processus prend fin lorsque les propriétés désirées sont atteintes. Nous avons illustré notre méthode sur l'exemple du brûleur à gaz.

L'exemple de brûleur à gaz a été traité par Lamport en TLA$^+$. Il a été traité aussi en duration Calculus. Kevin Lano [Lano, 1996; Lano *et al.*, 1996b; Lano and Haughton, 1996] l'a traité aussi en utilisant le B classique. Après avoir vu les différentes méthodes utilisées pour résoudre le brûleur à gaz, nous pouvons dire que la méthode appropriée pour l'étudier est le calcul de durée puisqu'elle présente des moyens pour expliciter des propriétés de durée sur des intervalles. L'avantage de notre méthode se situe au niveau de l'utilisation du raffinement qui devient un outil très puissant permettant de décrire les systèmes suivant différents degrés de granularité.

Chapitre 7

La composition en B pour le développement de systèmes automatisés

7.1 Introduction

Dans des travaux antérieurs [Mosbahi and Jaray, 2004b; Mosbahi *et al.*, 2006a], nous avons proposé une méthode de développement de contrôleurs de systèmes automatisés qui consistait à partir d'une modélisation concrète du système à contrôler (le contrôlé) que nous avons complété par un contrôleur abstrait minimal ou permissif pour obtenir un premier modèle du système automatisé. Nous nous sommes attachés à démontrer que, parmi les comportements possibles de ce système *certains* ont les propriétés attendues. Nous avons ensuite raffiné le modèle du système automatisé pour obtenir un modèle dont *tous* les comportements satisfont les propriétés attendues. Le raffinement consiste ici à développer ce qui constitue le contrôleur. Le modèle du contrôleur était alors obtenu par séparation des éléments du système automatisé qui n'appartenaient pas au modèle du contrôlé, construit au départ. Jugeant la dernière phase peu naturelle, nous nous sommes intéressés à une autre méthode consistant à garder séparés contrôleur et contrôlé tout au long du développement. Cette méthode pallie la dernière étape, peu naturelle, de la démarche précédente.

L'approche que nous proposons dans ce chapitre rappelle la technique du *co-design* où il s'agit de développer conjointement le contrôleur et le composant physique qu'il contrôle. Elle en diffère par le fait qu'ici, le contrôlé est déjà construit. A la différence de la méthode précédente, nous commençons à construire un modèle abstrait du système à contrôler, le raffinement s'applique ensuite aussi bien au contrôlé qu'au contrôleur. Elle consiste à partir d'un modèle abstrait du contrôleur et d'un modèle abstrait du contrôlé, puis à effectuer une suite de raffinements successifs jusqu'à obtenir un modèle implémentable du contrôleur et un modèle du contrôlé suffisamment concret pour tenir compte de tous les aspects du composant physique. L'objectif du raffinement du contrôlé est d'une part, de converger vers un modèle concret suffisamment proche du système réel qu'on pourra valider par simulation et d'autre part, de mieux gérer son interaction avec le contrôleur dès les premières étapes de développement. A chaque étape, nous composons les deux modèles pour obtenir un modèle du système automatisé sur lequel nous effectuons la vérification des propriétés de comportement. En ce qui concerne la vérification du contrôleur, elle s'effectue de façon indirecte. En effet, le contrôleur est réputé correct s'il forme avec le contrôlé un système automatisé dont le comportement est correct.

Dans le cas qui nous intéresse, les composants (contrôleur et contrôlé) ne sont pas de nature

symétrique ; il s'ensuit que l'opération de composition a des caractéristiques particulières.

Ce chapitre se présente comme suit : la section 2 présente notre démarche de développement par composition, la section 3 étudie les techniques de raffinement et de composition dans le cycle de développement de systèmes automatisés, la section 4 illustre notre approche sur une étude de cas et la dernière section tire un bilan de ce travail.

7.2 La composition dans le développement de contrôleurs

Dans cette section, nous proposons une démarche utilisant à la fois la composition et le raffinement pour développer un contrôleur pour un système automatisé. Nous proposons de développer deux modèles : l'un pour le contrôleur et l'autre pour le contrôlé. Le modèle du système automatisé est obtenu par composition de ces deux modèles. Par la suite, nous enrichissons les deux modèles qu'on compose à chaque niveau de raffinement pour obtenir le modèle du système automatisé sur lequel nous faisons la preuve des propriétés de comportement. A la fin du cycle de développement, nous disposons d'un modèle concret du contrôleur pour être implémenté et d'un modèle du contrôlé où toutes les caractéristiques physiques sont prises en compte.

Cette approche présente plusieurs intérêts tels que :

- la possibilité de réutilisation de la partie opérative pour le développement d'autres systèmes automatisés commandés par d'autres contrôleurs,
- l'obtention plus facile du contrôleur en fin de développement,
- le raffinement séparé du contrôleur et du contrôlé pour mieux gérer leur interaction dès les premières étapes de développement et avant la mise en oeuvre du contrôleur,
- la réduction de la complexité du système automatisé.

La définition de la composition tient compte du rôle dissymétrique joué par les composants. En effet, les événements "contrôlables" du contrôlé ont des gardes activées par le contrôleur qui observe l'état du contrôlé, décide des actions à déclencher et positionne des variables booléennes pour activer des gardes des événements du contrôlé. Autrement dit, un composant commande et l'autre réagit.

7.2.1 Présentation de la démarche

Le principe de base de la méthode consiste à développer le modèle du contrôleur et celui du contrôlé par raffinements successifs, ayant bien à l'esprit que le modèle du contrôleur n'a pas de sens que par son interaction avec le contrôlé. Ceci nous conduit à composer le contrôleur et le contrôlé pour réaliser un modèle du système automatisé. La correction du comportement du système automatisé est nécessaire à la preuve de correction du contrôleur.

A la première étape, nous devons supposer que le modèle du contrôlé que nous avons construit est une abstraction " pertinente " du contrôlé concret. A chaque étape, nous raffinons le contrôlé, nous prouvons que le raffinement est correct et la validation du contrôlé initial ne pourra être faite qu'à la dernière étape du développement quand nous aurons atteint le contrôlé concret. Pour valider le contrôlé concret, nous avons recours à la simulation pour comparer le comportement du modèle concret au comportement du système réel.

A chaque étape, la preuve du contrôleur consiste à démontrer qu'il est un raffinement correct du contrôleur qu'il raffine (cf. invariant de collage) et à démontrer que le modèle du système

automatisé obtenu par composition a le comportement souhaité. Le modèle du système automatisé obtenu à chaque étape doit être un raffinement du modèle du système automatisé de l'étape précédente. En effet, l'approche consiste à raffiner le contrôleur et le contrôlé, chacun muni de son invariant de collage, à les composer et à vérifier que le système automatisé est un raffinement du système automatisé précédent.

La démarche de développement illustrée par la figure 7.1 se présente comme suit :

- Construction d'un modèle B abstrait du contrôlé,
- Construction d'un modèle B abstrait du contrôleur,
- Composition des deux modèles précédents pour obtenir un modèle du système automatisé,
- Preuve des propriétés de comportement sur le système automatisé,
- Le développement continue dans un processus itératif qui s'arrête dès qu'on obtient un modèle implémentable du contrôleur et un modèle suffisamment concret du contrôlé. A chaque étape, nous faisons :
  - Raffinement du contrôlé et preuve,
  - Raffinement du contrôleur et preuve,
  - Construction du modèle du système automatisé par composition et preuve des propriétés de comportement,
- Récupération du modèle concret du contrôleur

7.2.2 Communication entre contrôleur et contrôlé

La communication entre les deux modèles se fait par des *variables partagées* contenues dans chacun des modèles, elles y sont déclarées dans la clause SHARED VARIABLES que nous ajoutons à un modèle B. Ces variables peuvent être :

- des *variables booléennes de commande* servant au contrôleur à établir les conditions de déclenchement d'un événement du contrôlé.
- des *variables d'échange de valeurs* permettant au contrôleur de passer des consignes au contrôlé.
- des *variables d'acquittement* permettant d'informer le contrôleur de la fin d'execution des événements du contrôlé. L'usage de ces variables est nécessaire dans le cas où l'exécution d'une commande par le contrôlé met en œuvre un processus qui prend un temps indéterminé. Ces variables vont être utilisées dans le chapitre suivant.

7.2.3 Modélisation du contrôleur et du contrôlé

Dans un système automatisé, le système physique dispose d'actionneurs qui sont activés par le contrôleur ainsi que de capteurs qui signalent au contrôleur des états distingués dans lesquels il se trouve (température ambiante égal à une valeur donnée, porte fermée, ...). Le contrôleur est aussi amené à communiquer au contrôlé des consignes qui guident son comportement.

Les actionneurs dans le contrôlé sont représentés en B par des événements dont la garde contient une variable booléenne partagée qui est mise à vrai par une action du contrôleur pour lancer la commande. L'action modélise l'effet de la commande. Les capteurs sont représentés par

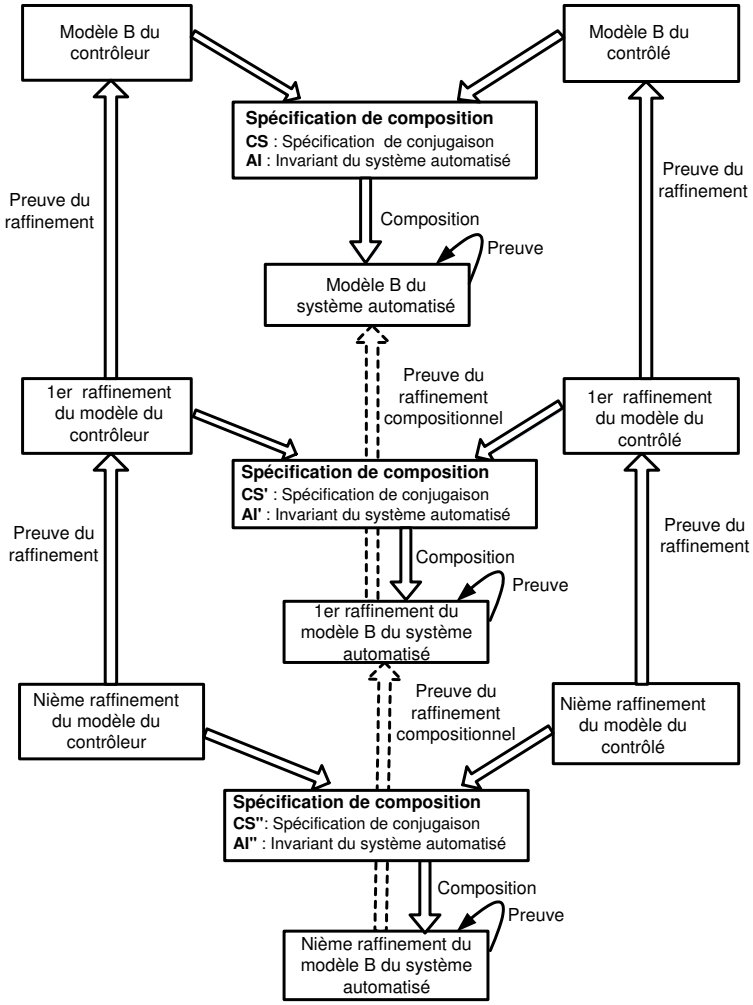


FIG. 7.1 – Développement avec composition

des événements du contrôlé dont la garde est mise à vrai dans l'état qui doit être signalé, l'action consiste à mettre à vrai une variable booléenne partagée accessible en lecture par le contrôleur.

En toute généralité, le contrôlé a un comportement qui lui est propre auquel participent des événements dont l'activation ne dépend pas du contrôleur. Nous dirons que ces événements sont libres. Symétriquement, tous les événements du contrôleur ne correspondent pas à des réactions à des signaux émis par le contrôlé, certains événements pouvant correspondre à une évolution autonome du contrôleur (événements libres au niveau du contrôleur).

Dans ce qui suit, nous notons par E_c et E_{op} respectivement les ensembles des définitions des événements du contrôleur et du contrôlé et par V_c et V_{op} respectivement les ensembles des variables propres au contrôleur et au contrôlé. Ces variables sont différentes des variables partagées que nous introduisons. Soit *Names* la fonction qui fournit les noms des événements à partir de leurs définitions.

Types d'événements au niveau du contrôlé :

Les événements du contrôlé correspondent à :

- des *événements libres* : ces événements échappent au contrôle exercé par le contrôleur. Ce sont des événements dont les gardes ne contiennent pas des prédicats sur les *variables partagées*. Soit $Names(\mathrm{E_{Lop}})$ la fonction qui fournit les noms des événements libres du contrôlé, où $\mathrm{E_{Lop}}$ correspond à l'ensemble de définitions des événements libres.
- des *événements contrôlés* : ces événements correspondent à des événements dont la garde contient des prédicats sur des *variables partagées* booléennes. Soit $Names(\mathrm{E_{Cop}})$, la fonction qui fournit les noms des événements contrôlés, où $\mathrm{E_{Cop}}$ correspond à l'ensemble de définitions des événements contrôlés.

Nous avons $Names(\mathrm{E_{op}}) = Names(\mathrm{E_{Lop}}) \cup Names(\mathrm{E_{Cop}})$.

Types d'événements au niveau du contrôleur :

Les événements du contrôleur correspondent à :

- des *événements de contrôle* : ce sont des événements dont l'action consiste à modifier les valeurs des *variables partagées* et dont la garde dépend de ces variables modifiées par le contrôlé. Soit $Names(\mathrm{E_{Cc}})$, la fonction qui fournit les noms des événements de contrôle du contrôleur, où $\mathrm{E_{Cc}}$ correspond à l'ensemble de définitions des événements de contrôle.
- des *événements libres* : ce sont des événements indépendants de l'état du contrôlé (événements sans correspondance dans la partie opérative). Soit $Names(\mathrm{E_{Lc}})$, la fonction qui fournit les noms des événements libres du contrôleur, où $\mathrm{E_{Lc}}$ correspond à l'ensemble de définitions des événements libres.

Nous avons $Names(\mathrm{E_{c}}) = Names(\mathrm{E_{Lc}}) \cup Names(\mathrm{E_{Cc}})$.

La figure 7.2 présente un schéma qui décrit la communication entre contrôleur et contrôlé via des *variables partagées*.

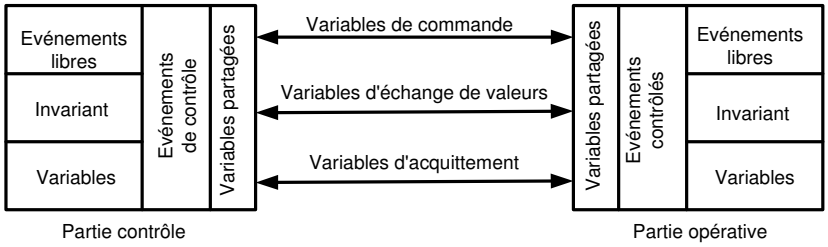


FIG. 7.2 – Communication entre contrôleur et contrôlé via des variables partagées

7.2.4 Construction du système automatisé

Dans cette section, nous montrons comment construire le modèle du système automatisé à partir des modèles de ses composants. Le modèle d'interaction entre un contrôleur et un contrôlé

peut être modélisé de façon classique par une boucle de contrôle telle qu'elle est décrite dans la figure 7.3 où :

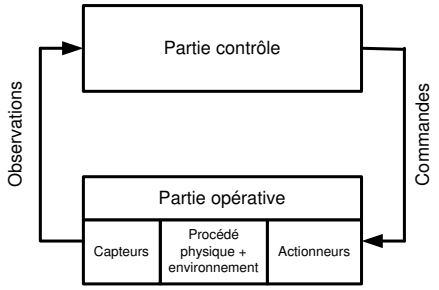


FIG. 7.3 – Système automatisé dans une boucle fermée

- un actionneur est un élément du contrôlé qui reçoit des ordres du contrôleur pour réaliser une action.
- un capteur est un élément fournissant des informations sur l'état du contrôlé.

La boucle de contrôle induit un ordre de déclenchement sur les événements du système automatisé. Comme nous faisons l'hypothèse que le contrôlé réagit immédiatement à une commande du contrôleur, il ne peut y avoir d'autres événements qui viennent s'intercaler entre le lancement d'une commande et son exécution. C'est ainsi que l'exécution d'un événement du contrôleur mettant à jour une *variable de commande* doit être suivie immédiatement par l'exécution de l'événement du contrôlé dont la garde contient cette variable. Cette remarque justifie la fusion des deux événements en un seul. Cette fusion nous permet d'éliminer des *variables partagées*.

Pour construire le système automatisé, il est nécessaire de repérer les couples d'événements dont l'un modélise une commande du contrôleur et l'autre son exécution par le contrôlé. Nous avons choisi d'expliciter les couples d'événements à fusionner dans un langage qui permet d'ajouter des informations supplémentaires pour réaliser la fusion par un langage de spécification de composition.

Les événements du système automatisé proviennent :

- de la fusion des couples d'événements de la forme (e_c, e_{op}), où e_c désigne un événement du contrôleur et e_{op} un événement du contrôlé tels que l'action du contrôleur active la garde de l'événement e_{op} du contrôlé,
- des *événements libres* du contrôlé qui s'exécutent sans dépendre du contrôleur. Les gardes de *certains* de ces événements sont renforcées par des conditions supplémentaires sur les variables d'état du contrôleur,
- des *événements libres* du contrôleur qui n'ont pas de contrôle sur les événements du contrôlé, comme des événements qui servent à faire des calculs par exemple.

*** Définition de la spécification de composition** (SC)

La spécification de composition SC définit les règles de composition des événements des modèles du contrôleur et du contrôlé. Cette spécification nous permet d'associer l'événement

e_c du contrôleur à l'événement e_{op} qu'il contrôle, autrement dit l'événement e_c dont l'action active la garde de e_{op}. Cet ensemble constitue ce que nous notons l'ensemble des événements conjugués de la spécification (*ensemble des conjugaisons*). Un autre élément de la spécification est l'*invariant du système automatisé* qui définit la propriété qu'il doit satisfaire. La spécification de composition permet d'obtenir automatiquement le modèle du système automatisé à partir des modèles de ses composants contrôleur et contrôlé.

- Définition de l'ensemble des conjugaisons CS**.**
L'*ensemble des conjugaisons* CS est un sous ensemble de couples de noms d'événements de type (commande, exécution). Les gardes des *événements contrôlés* de la partie opérative utilisent des variables partagées modifiées par les substitutions généralisées des *événements de contrôle* du contrôleur. Cet ensemble est défini comme suit :

$$\mathbf{CS} \subset \mathbf{\{}(e_c, e_{op}) \mid e_c \in \text{Names}(E_{Cc}) \wedge e_{op} \in \text{Names}(E_{Cop})\mathbf{\}}.$$

Exemple.
L'exemple suivant présente la fusion de deux événements conjugués e_c du contrôleur et e_{op} du contrôlé. Au niveau de l'événement résultant, les variables partagées disparaissent.

e_c = SELECT
    $\neg go \wedge G$
    THEN
    go := true ||
    GS_c
    END ;

e_{op} = SELECT
    go
    THEN
    go := false ||
    GS_{op}
    END ;

Ensemble de conjugaison :
$e = (e_c, e_{op})$

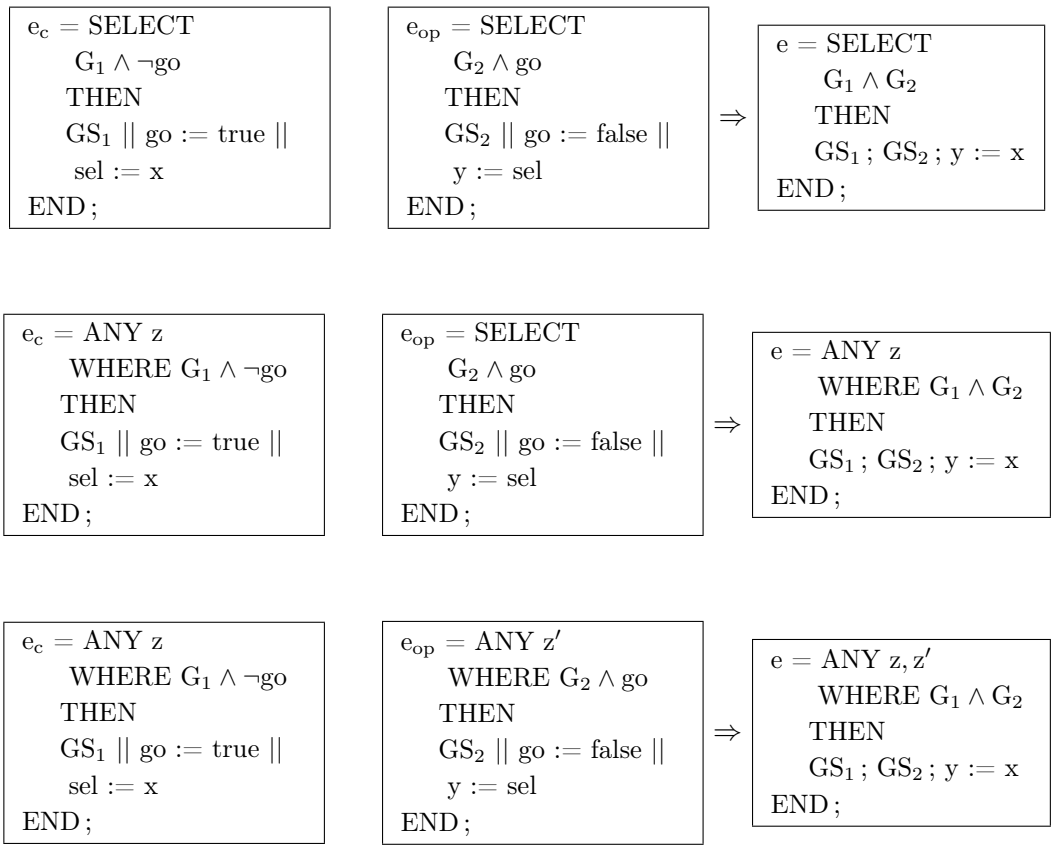
Résultat :
e = SELECT G THEN $GS_c ; GS_{op}$

Où :

go : une variable de commande,
G : une garde,
GS, GS' : des substitutions généralisées,
e_c : événement de contrôle,
e_{op} : événement contrôlé,
e : événement du système automatisé

Le résultat de la fusion est un séquencement de deux substitutions correspondant à la notion naturelle d'enchaînement de deux transitions. L'état *post* de la première est l'état *pré* de la seconde. En d'autres termes, GS doit établir la faisabilité et la terminaison de GS'.

Dans ce qui suit, nous donnons les scénarios possibles de fusion des événements conjugués des parties contrôle et opérative. Soient go et sel deux *variables partagées*, go étant une variable de commande et sel une variable servant à échanger des valeurs entre contrôleur et contrôlé. Soient e_c, e_{op} respectivement deux événements tels que $e_c \in \text{Names}(E_{Cc})$ et $e_{op} \in \text{Names}(E_{Cop})$. Les variables x et y appartiennent respectivement aux modèles du contrôleur et du contrôlé. Il existe trois schémas de fusion des événements :



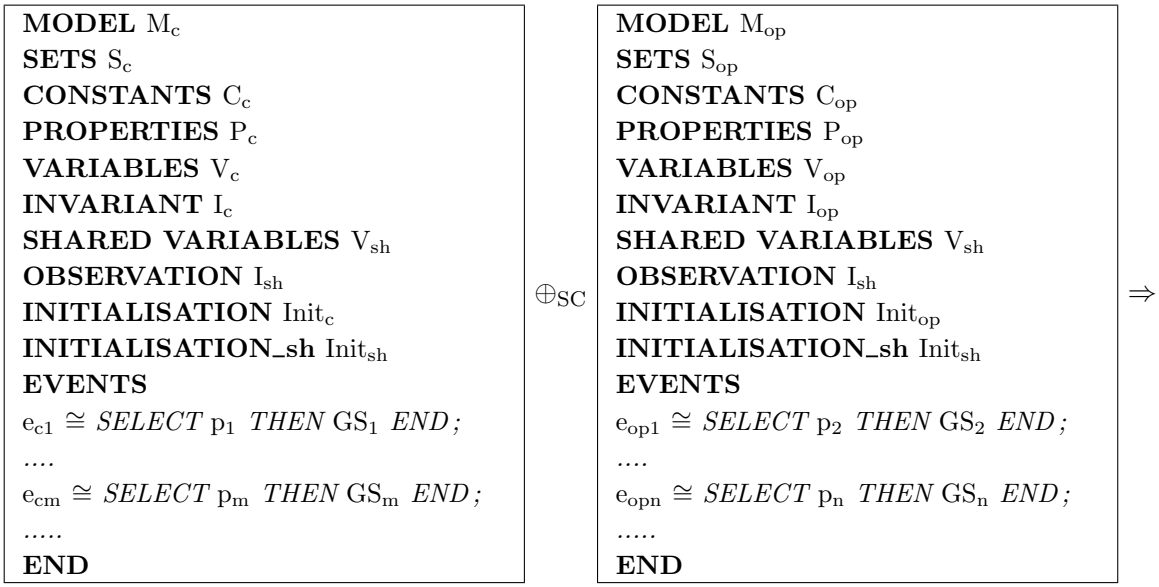
- Définition de l'invariant du système automatisé I_{aut}.
L'invariant du système automatisé contient les propriétés à vérifier par le système composite. Il s'exprime comme un invariant en B et porte sur les variables d'état des deux modèles. Cet invariant ne contient pas des variables partagées (variables de commande, variables d'acquittement).

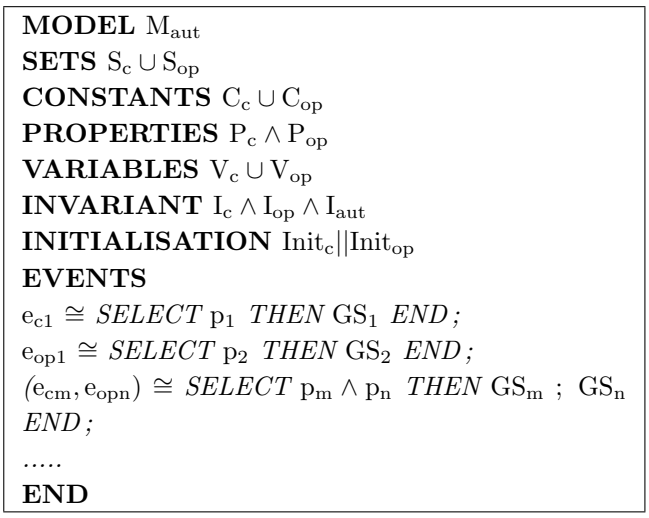
*** Composition des modèles B**

Dans cette section, nous donnons les règles de construction du modèle du système automatisé à partir des modèles du contrôleur et du contrôlé. Soient M_c et M_{op} deux modèles B respectivement du contrôleur et du contrôlé. Soit $\oplus$ l'opérateur de composition. Cet opérateur qui à M_c et M_{op} associe M_{aut} ($\oplus : M_c \times M_{op} \rightarrow M_{aut}$).

Nous proposons une extension syntaxique des modèles B par les clauses SHARED VARIABLES, OBSERVATION et INITIALISATION_sh. La clause SHARED VARIABLES modélise les variables partagées, la clause OBSERVATION modélise les invariants de typage sur les variables partagées et la clause INITIALISATION_sh modélise l'initialisation de ces variables. Ces clauses disparaissent dans le modèle du système automatisé.

La composition de M_c et M_{op} sous la spécification de composition SC est définie comme suit :





La réalisation de la composition a pour effet d'obtenir un modèle où les nouvelles clauses disparaissent. En effet, en composant les deux modèles des contrôleur et contrôlé, les variables partagées disparaissent du modèle du système automatisé par des règles de simplification introduites au moment de la fusion des événements conjugués. Ceci induit des règles de simplification des invariants que nous n'avons pas explicité. De ce fait, intuitivement l'invariant I_{sh} portant sur ces variables (invariant de typage sur les variables partagées) disparaît.

* Règles de composition de modèles B

Soient M_c et M_{op} deux modèles B représentant respectivement le contrôleur et le contrôlé d'un système automatisé. Un événement B est défini par sa garde (noté *grd*) et par sa substitution généralisée (noté *GS*). Soient V_c et V_{op} respectivement les variables des modèles du contrôleur et du contrôlé. Soient I_c et $Init_c$ respectivement l'invariant et l'initialisation des variables du contrôleur. Soient I_{op} et $Init_{op}$ respectivement l'invariant et l'initialisation des variables du contrôlé. Soit E l'ensemble des définitions des événements du système automatisé. Soient LS,

CS respectivement les ensembles des événements libres et des événements conjugués du système automatisé.

La composition de M_c et M_{op} sous la spécification de composition SC est définie comme suit :

1. $V = V_c \cup V_{op}$
2. $I = I_c \wedge I_{op} \wedge I_{aut}$
3. $Init = Init_c || Init_{op}$
4. Les événements de E sont construits de la manière suivante :
   (a) [CR1] $e_i \in$ LS, $e_i \in Names(E_{Lop})$ alors $e = e_i$ et $e_i \in Names(E)$ et $grd(e) = grd(e_i)$ et $GS(e) = GS(e_i)$
   (b) [CR2] $e_j \in$ LS, $e_j \in Names(E_{Lc})$ alors $e = e_j$ et $e_j \in Names(E)$ et $grd(e) = grd(e_j)$ et $GS(e) = GS(e_j)$
   (c) [CR3] $(e_i, e_j) \in CS, e_i \in Names(E_{Cc}), e_j \in Names(E_{Cop})$ alors $e = (e_i, e_j)$ et $e \in Names(E)$ et $grd(e) = grd(e_i) \wedge grd(e_j)$ et $GS(e) = GS(e_i); GS(e_j)$

7.3 Raffinement et composition

Un intérêt de la composition est la possibilité de s'appuyer sur la vérification des composants pour vérifier les propriétés du système automatisé. Pour ce faire, il faut s'acquitter des obligations de preuve pour prouver qu'un système composé de composants raffinés est un raffinement du système composé des composants abstraits correspondants, sachant que les raffinements des composants ont été prouvés. Dans le cas qui nous intéresse, la relation particulière qui existe entre le contrôleur et le contrôlé est telle qu'à partir du moment où les raffinements du contrôleur et du contrôlé sont prouvés, la preuve de raffinement du système composé est établie dès que l'ensemble des conjugaisons au niveau abstrait (CS) est inclue dans l'ensemble des conjugaisons au niveau raffiné ($CS \subset CS'$). Le système raffiné doit en outre satisfaire les invariants introduits dans la spécification de composition.

Nous allons prouver ce qui vient d'être énoncé en tenant compte des caractéristiques particulières des composants qui entrent en jeu dans la composition. Ces propriétés découlent de la méthode de développement. Tout d'abord, il faut remarquer que le raffinement est dirigé par le raffinement du contrôlé. En effet, un raffinement du contrôlé sert à modéliser, par de nouveaux événements, de nouveaux capteurs et de nouveaux actionneurs. A ces nouveaux événements vont correspondre, dans le contrôleur raffiné, des événements dont certains vont observer les signaux émis par les capteurs et d'autres vont émettre des signaux d'activation des actionneurs. Les autres événements qui sont introduits dans les raffinements sont des événements libres, on les trouve aussi bien dans le contrôleur que dans le contrôlé. Il n'y a pas de lien entre ces événements au sens où ils ne modifient pas de variables communes.

Soient CS, CS′ les ensembles d'*événements conjugués* aux niveaux abstrait et raffiné ; LS, LS′ les ensembles d'*événements libres* aux niveaux abstrait et raffiné. Soient E_c, E_{op} les ensembles des définitions des événements des contrôleur et contrôlé. Soient E_{Lc}, E_{Lop} les ensembles des définitions des événements libres des contrôleur et contrôlé, E_{Cc}, E_{Cop} les ensembles des définitions des événements de contrôle et des événements contrôlés.

Soient N, N_c et N_{op} les ensembles de nouveaux événements respectivement du système automatisé, du contrôleur et du contrôlé.

Le processus de développement induit un certain nombre de propriétés :

- [P1] : les événements du contrôlé sont libres ou sous contrôle du contrôleur. Dans le second cas, ils seront couplés à un événement de contrôle du contrôleur.

  $\forall a, b.a \in Names(E_c)$ et $b \in Names(E_{op}),$ on a, $(a, b) \in CS$ ou $a \in LS$ et $b \in LS$.

- [P2] : un événement qui apparaît libre dans le système composé à un niveau de développement, le restera dans la composition des raffinés.

  $\forall a.a \in Names(E_{Lc}) \cup Names(E_{Lop}),$ on a, $a \in LS \Leftrightarrow a \in LS'$.

- [P3] : un événement qui apparaît conjugué sous la forme (e_c, e_{op}) dans le système composé à un niveau de développement, le restera dans la composition des raffinés.

  $\forall a, b.a \in Names(E_{Cc})$ et $b \in Names(E_{Cop}),$ on a, $(a, b) \in CS \Leftrightarrow (a, b) \in CS'$;

- [P4] : les nouveaux événements peuvent se synchroniser entre eux et pas avec des anciens événements (à chaque nouvel événement sous contrôle du contrôlé, est associé un nouvel événement de contrôle du contrôleur).

  $\forall a, b.a \in Names(E_{Cc}) \cup N_c$ et $b \in Names(E_{Cop}) \cup N_{op},$ on a, $(a, b) \in CS' \Rightarrow a \in N_c$ et $b \in N_{op}$ ou $a \in Names(E_{Cc})$ et $b \in Names(E_{Cop})$.

- [P5] : un invariant de collage relatif à un composant (contrôleur ou contrôlé) est construit sur les variables partagées (variables de commande et d'acquittement), ou sur les variables d'états du composant mais il ne peut pas être construit sur les deux types de variables à la fois.

  I_{cc} ::= sp tel que sp $\in SP_{v_c}$ ou sp $\in SP_{v_{sh}}$.
  I_{cp} ::= sp tel que sp $\in SP_{v_{op}}$ ou sp $\in SP_{v_{sh}}$.
  Où sp désigne un prédicat d'état et SP_{v_c}, $SP_{v_{op}}$ désignent respectivement les ensembles des prédicats sur les variables d'état (différentes des variables partagées) des contrôleur et contrôlé. $SP_{v_{sh}}$ désigne l'ensemble des prédicats sur les variables partagées.

Les propriétés qui viennent d'être énoncées étant propres à notre méthode de développement, le théorème suivant indique sous quelles conditions il est possible de déduire que le composé des raffinements du contrôleur et du contrôlé est un raffinement du composé des contrôleur et contrôlé abstraits.

- Théorème de composition des raffinements

Soient C_a, OP_a les modèles abstraits des contrôleur et contrôlé et C_r, OP_r respectivement leur raffinement. Soient SC, SC′ les spécifications de composition aux niveaux abstrait et raffiné.

Nous notons par τ, un nouvel événement du système automatisé, τ_c, un nouvel événement du contrôleur et par τ_{op}, un nouvel événement du contrôlé.

Le théorème de composition des raffinements entre contrôleur et contrôlé sous les propriétés P1, P2, P3, P4 et P5 s'énonce de la manière suivante :

$$\frac{\begin{array}{l} C_a \sqsubseteq_{I_{cc}} C_r \\ OP_a \sqsubseteq_{I_{cp}} OP_r \\ ([P1], [P2], [P3], [P4] \text{et} [P5]) \end{array}}{C_a \oplus_{SC} OP_a \sqsubseteq_{I_{cc} \wedge I_{cp}} C_r \oplus_{SC'} OP_r}$$

Où $\sqsubseteq_I$ dénote la relation de raffinement où l'invariant de collage est I

- Preuve du théorème de composition des raffinements

Soient E_{ac}, E_{ap} les ensembles des définitions des événements des contrôleur et contrôlé au niveau abstrait, E_{rc}, E_{rp} les ensembles des définitions des événements des contrôleur et contrôlé au niveau raffiné.

Soient e_{ac}, e_{ap}, e_{rc} et e_{rp} quatre événements tels que $e_{ac} \in Names(E_{ac})$, $e_{ap} \in Names(E_{ap})$, $e_{rc} \in Names(E_{rc})$ et $e_{rp} \in Names(E_{rp})$. Soient $e_{ac} \sqsubseteq e_{rc}$ et $e_{ap} \sqsubseteq e_{rp}$. Soient $(e_{ac}, e_{ap}) \in CS$ et $(e_{rc}, e_{rp}) \in CS'$.

Pour prouver le théorème de composition des raffinements, nous devons montrer :

1. Que les événements du système automatisé raffiné raffinent les événements du système automatisé abstrait.
2. Pas plus de blocage dans le modèle du système automatisé raffiné.
3. Les nouveaux événements ne prennent pas le contrôle indéfiniment dans le modèle du système automatisé raffiné

* ***Preuve de 1*** :

Pour montrer la preuve de 1 dans le théorème de compositionnalité, il suffit de montrer la relation de raffinement entre les événements du système automatisé raffiné et abstrait. Les événements du système automatisé raffiné correspondent à des événements conjugués, ou des événements libres du contrôleur et du contrôlé. Pour ce faire, nous introduisons un ensemble S qui exprime la relation entre les événements des systèmes automatisés abstrait et raffiné, et nous montrons que l'ensemble S vérifie la relation de raffinement en respectant les invariants de collage I_{cc} et I_{cp}, où :

$S = \{((e_{ac}, e_{ap}), (e_{rc}, e_{rp})) \mid e_{ac} \sqsubseteq e_{rc} \wedge e_{ap} \sqsubseteq e_{rp}\} \cup \{(e_{ac}, e_{rc}) \mid e_{ac} \sqsubseteq e_{rc}\} \cup \{(e_{ap}, e_{rp}) \mid e_{ap} \sqsubseteq e_{rp}\}$

- Si on suppose que $((e_{ac}, e_{ap}), (e_{rc}, e_{rp})) \in S$ alors $(e_{ac}, e_{ap}) \in CS$ et $(e_{rc}, e_{rp}) \in CS'$ et $(e_{ac}, e_{ap}) \sqsubseteq (e_{rc}, e_{rp})$, c.à.d. $e_{ac} \sqsubseteq e_{rc}$ et $e_{ap} \sqsubseteq e_{rp}$.
- Si on suppose que $(e_{ac}, e_{rc}) \in S$ alors $e_{ac} \in LS$ et $e_{rc} \in LS'$ et $e_{ac} \sqsubseteq e_{rc}$.
- Si on suppose que $(e_{ap}, e_{rp}) \in S$ alors $e_{ap} \in LS$ et $e_{rp} \in LS'$ et $e_{ap} \sqsubseteq e_{rp}$.

Deux cas se présentent :

- Cas du raffinement strict.

  i. Événements conjugués :
  Supposons que $e_r = (e_{rc}, e_{rp}) \in CS'$, nous devons montrer qu'il existe un événement e_a tel que $e_a = (e_{ac}, e_{ap}) \in CS$ et $e_{ac} \sqsubseteq e_{rc}$ et $e_{ap} \sqsubseteq e_{rp}$.
  Par [CR3], on a $e_r = (e_{rc}, e_{rp}) \in CS'$, $e_{rc} \in Names(E_{Cc})$ et $e_{rp} \in Names(E_{Cop})$ et $grd(e_r) = grd(e_{r_c}) \wedge grd(e_{r_p})$ et $GS(e_r) = GS(e_{r_c}); GS(e_{r_p})$.
  Puisque $C_a \sqsubseteq_{I_{cc}} C_r$, il existe un événement e_{ac} tel que $e_{ac} \sqsubseteq e_{rc}$. Similairement, comme $OP_a \sqsubseteq_{I_{cp}} OP_r$, il existe un événement e_{rp} tel que $e_{ap} \sqsubseteq e_{rp}$. Or $e_r = (e_{rc}, e_{rp})$ donc selon [P3], il existe un événement $e_a \mid e_a = (e_{ac}, e_{ap}) \in CS$ et $(e_{ac}, e_{ap}) \sqsubseteq (e_{rc}, e_{rp})$ et par [CR3], $e_a = (e_{ac}, e_{ap})$ et $(e_{ac}, e_{ap}) \in CS$ et $grd(e_a) = grd(e_{a_c}) \wedge grd(e_{a_p})$ et $GS(e_a) = GS(e_{a_c}); GS(e_{a_p})$.

  ii. Événements libres :

  A. Les événements du contrôlé :
  Supposons que $e_{rp} \in LS'$, nous devons montrer qu'il existe un événement e_{ap} tel que $e_{ap} \in LS$ et $e_{ap} \sqsubseteq e_{rp}$.

Par [CR1], on a, $e_{rp} \in LS'$. Puisque $OP_a \sqsubseteq_{I_{cp}} OP_r$, il existe un événement e_{ap} tel que $e_{ap} \sqsubseteq e_{rp}$. Selon [P1] et [P2], il existe un événement $e_{ap} \mid e_{ap} \in LS$ et $e_{ap} \sqsubseteq e_{rp}$.

B. Les événements du contrôleur :
Supposons que $e_{rc} \in LS'$, nous devons montrer qu'il existe un événement e_{ac} tel que $e_{ac} \in LS$ et $e_{ac} \sqsubseteq e_{rc}$.
Par [CR2], on a, $e_{rc} \in LS'$. Puisque $C_a \sqsubseteq_{I_{cc}} C_r$, il existe un événement e_{ac} tel que $e_{ac} \sqsubseteq e_{rc}$. Selon [P2] et [P1], il existe un événement $e_{ac} \mid e_{ac} \in LS$ et $e_{ac} \sqsubseteq e_{rc}$.

- Cas du raffinement avec bégaiement.

i. Événements conjugués :
Supposons que $e_r = (e_{rc}, e_{rp}) \in CS'$ et $e_{rc} = \tau_c$ et $e_{rp} = \tau_{op}$ et montrons que e_r raffine skip. Selon [P1], e_r est un nouvel événement donc $e_r = \tau = (\tau_c, \tau_{op})$.
Puisque C_r raffine C_a et e_{rc} un nouvel événement, alors il raffine skip. Similairement, comme OP_r raffine OP_a, et e_{rp} un nouvel événement, alors il raffine skip.
Selon [P4], on synchronise que de nouveaux événements et pas un nouvel événement avec un ancien. Donc, $e_r = (e_{rc}, e_{rp})$ et e_r raffine skip.

ii. Événements libres :

A. Événements du contrôlé :
supposons que e_{rp} est un nouvel événement. Par [CR1], $e_{rp} \in LS'$ et $e_{rp} \in Names(E_{Lp})$. Selon [P1], e_{rp} n'apparaît pas conjugué avec un événement du contrôleur. Donc e_{rp} raffine skip.

B. Événements du contrôleur :
(pareil pour les événements libres du contrôleur).

* ***Preuve de 2*** : Pas plus de blocage dans le modèle du système automatisé

La preuve est une conséquence immédiate qu'il est établi qu'aucun nouveau blocage n'a été introduit dans les modèles des contrôleur et contrôlé raffinés (disjonction des gardes des événements abstraits des contrôleur et contrôlé implique la disjonction des gardes des événements concrets des contrôleur et contrôlé).
En effet, les événements du système automatisé sont construits à partir des événements de ses composants contrôleur et contrôlé. Deux cas se présentent :

- les événements conjugués : la garde de ses événements correspond à celle de l'événement de contrôle.
- les événements libres : sont traités de la même manière que dans les contrôleur et contrôlé.

* ***Preuve de 3*** : Les nouveaux événements ne prennent pas le contrôle indéfiniment dans le modèle du système automatisé raffiné.

La preuve est une conséquence immédiate que les nouveaux événements ne prennent pas le contrôle indéfiniment dans les modèles des contrôleur et contrôlé raffinés. Dans les modèles des contrôleur et contrôlé, il y a manipulation des variants qui doivent

décroître strictement à chaque activation de chaque nouvel événement afin de garantir l'absence de cycles de nouveaux événements. Dans le modèle du système automatisé, ces variants restent valables puisqu'ils concerne les mêmes événements.

7.4 Etude de cas : tri des paquets postaux

Nous illustrons dans ce qui suit notre méthode sur l'exemple du système de tri des paquets (Figure 7.4) présenté au chapitre 5.

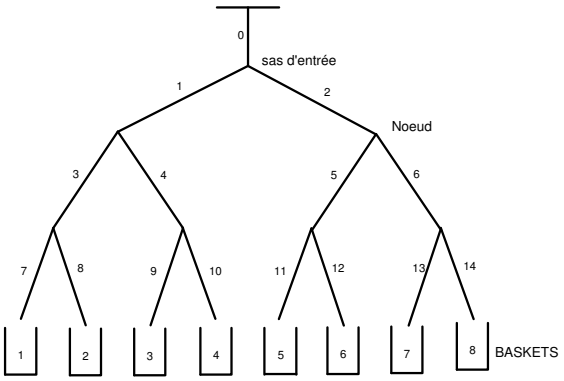


FIG. 7.4 – Système de tri de paquets

7.4.1 Modèle abstrait du système

Dans notre méthode, nous discrétisons le contrôlé (partie opérative) et nous commençons par construire une abstraction aussi bien du contrôleur que du contrôlé.

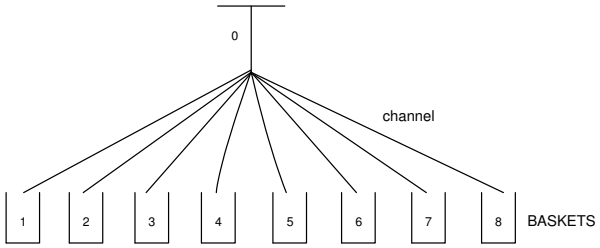


FIG. 7.5 – Abstraction de la partie opérative

*** Modèle abstrait du contrôlé**

Le modèle abstrait du contrôlé est présenté figure 7.5.

Variables : Les variables modifiables par le modèle du contrôlé sont :

- *channel* modélise le dispositif de tri qui relie l'entrée à un bac de destination,
- *next* désigne un paquet choisi en entrée mais qui n'est pas libéré dans le dispositif de tri,
- *current* désigne un paquet entré dans le dispositif de tri,
- *adr_next* modélise une mémoire qui lit l'adresse de tout paquet entré,
- *sorted* modélise les paquets qui ont atteint un bac de sortie,
- *stored* modélise l'ensemble des paquets triés.

Événements : Le contrôlé contient des *événements libres* et des *événements contrôlés*. A ce niveau d'abstraction, nous pouvons observer quatre événements : *Select_parcel*, *Set_channel_exe*, *Release_exe* et *Cross_sorting*. L'événement *Select_parcel*, modélise le choix du paquet (le paquet en tête de la file des paquets à trier) et la lecture de son adresse de destination (*adr_next := adr(p)*) ; l'événement *Set_channel_exe*, met le canal à l'adresse de destination définie par le contrôleur ; l'événement *Release_exe* modélise l'ouverture du sas d'entrée et l'événement *Cross_sorting* modélise le passage du paquet dans le système de tri.

- Événements libres : les événements libres sont ceux qui n'attendent pas des ordres pour être exécutés, autrement dit dont les gardes ne peuvent pas être modifiées par le contrôleur. Ces événements sont Select_parcel et Cross_sorting.

Select_parcel = **ANY** p
WHERE $p \in TO\_SORT \wedge$
$next \in UNDEF \wedge$
$current \in UNDEF$
THEN
adr_next := adr(p) ||
$next$:= p
END ;

$TO\_SORT \triangleq PARCELS$ - $sorted$
$UNDEF \triangleq EPARCELS$ - $PARCELS$
$PARCELS \subset EPARCELS$
$BASKETS \subset NODES$
$adr \in$ PARCELS$\rightarrow BASKETS$

Cross_sorting = **SELECT**
$current \in PARCELS$
THEN
stored(current) := channel ||
sorted := sorted $\cup$ current ||
$current : \in UNDEF$
END ;

$next \in EPARCELS$
$current \in EPARCELS$
$stored \in PARCELS \nrightarrow BASKETS$
$sorted \subseteq PARCELS$
$channel \in BASKETS$
$adr\_next \in BASKETS$

Où TO_SORT désigne l'ensemble de paquets à trier, UNDEF l'ensemble de paquets indéfinis, NODES l'ensemble des nœuds et BASKETS les nœuds destination (feuilles de l'arbre). Le prochain paquet à trier est désigné par la variable next. La variable current désigne le paquet en cours de tri et stored les paquets triés.

- Événements contrôlés : les événements contrôlés attendent l'ordre de leur exécution de la part de leurs *conjugués* au niveau du contrôleur. Nous ajoutons des *variables de commande* (*go_set*, *go_release*) pour modéliser l'ordre dans lequel s'exécutent ces événements. D'autre part, le contrôleur interagit aussi avec le contrôlé pour passer des valeurs grâce à des *variables d'échange*. Les événements contrôlés sont : *Set_channel_exe* et *Release_exe*.

Set_channel_exe =
SELECT
go_set = *TRUE*
THEN
channel := sel_channel ||
go_set := *FALSE*
END ;

Release_exe =
SELECT
go_release = *TRUE*
THEN
current := *next* ||
next: ∈ UNDEF ||
go_release := *FALSE*
END ;

Variables partagées
Variables de commande
go_set ∈ *BOOL*
go_release ∈ *BOOL*
Variables d'échange
sel_channel ∈ *BASKETS*

Le modèle du contrôlé est présenté en annexe D. Il est vérifié par Click_n_prove.

*** Modèle abstrait du contrôleur**

Variables : la variable modifiable par le contrôleur est la variable booléenne *ready_to_sort* qui modélise l'état de la porte du sas d'entrée.

Événements : aux événements contrôlés de la partie opérative (Set_channel_exe et Release_exe) correspondent leurs conjugués les *événements de contrôle* du contrôleur (Set_channel_cmd et Release_cmd). Ces événements permettent respectivement de déterminer le canal que le paquet devra emprunter et la commande d'ouverture du sas d'entrée.

Set_channel_cmd ≙ **SELECT**
¬ *ready_to_sort* ∧
next ∈ *PARCELS* ∧
current ∈ *UNDEF* ∧ ¬ *go_set*
THEN
sel_channel := *adr_next* ||
ready_to_sort := *TRUE* ||
go_set := *TRUE*
END ;

Release_cmd ≙ **SELECT**
next ∈ *PARCELS* ∧
current ∈ *UNDEF* ∧
ready_to_sort ∧
¬ *go_release*
THEN
ready_to_sort := *FALSE* ||
go_release := *TRUE*
END ;

Nous donnons dans la figure 7.6, le diagramme de transitions correspondant à l'exécution des événements du contrôleur et du contrôlé.

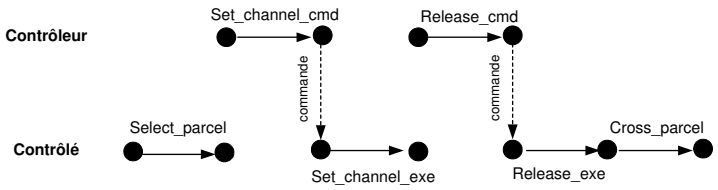


FIG. 7.6 – Diagramme de transitions correspondant au modèle abstrait.

Le modèle du contrôleur est présenté en annexe D. Il est vérifié par Click_n_prove.

*** Spécification de composition**

Dans le système composite, nous souhaitons que tout paquet introduit dans le magasin finisse par arriver dans le bac de réception correspondant à son adresse de destination. Cette propriété est décrite par l'invariant suivant : $\forall p.(p \in \text{PARCELS} \Rightarrow \text{stored}(p) = \text{adr}(p))$.

Le couplage entre le contrôleur et le contrôlé est défini par l'ensemble des couples suivants qui constituent la spécification de composition.

$$CS = \{(\text{Set\_channel\_cmd}, \text{Set\_channel\_exe}), (\text{Release\_cmd}, \text{Release\_exe})\}$$

Le système automatisé résultant de cette composition contient les événements suivants : *Set_channel* et *l'événement Release*.

- ***Événement Set_channel***.

  Cet événement résulte de la fusion des événements conjugués Set_channel_cmd du contrôleur et Set_channel_exe du contrôlé.

***Set_channel_cmd*= SELECT**
¬ ready_to_sort ∧
next ∈ PARCELS ∧
current ∈ *UNDEF* ∧
¬go_set
THEN
sel_channel := adr_next ||
ready_to_sort := TRUE ||
go_set :=TRUE
END;

;

***Set_channel_exe* = SELECT**
go_set
THEN
channel := sel_channel ||
go_set := FALSE
END;

⇒

***Set_channel* = SELECT**
¬ ready_to_sort ∧
next ∈ PARCELS ∧
current ∈ *UNDEF*
THEN
channel := adr_next ||
ready_to_sort := *TRUE*
END;

L'événement *Set_channel* est obtenu de la manière suivante :

$$\begin{aligned}
\text{Set_channel} &\triangleq (\text{Set_channel_cmd}; \text{Set_channel_exe}) \\
&\triangleq \text{grd(Set_channel_cmd) THEN GS(Set_channel_cmd)} \\
&\quad ; \\
&\quad \text{grd(Set_channel_exe) THEN GS(Set_channel_exe)} \\
&\quad \triangleq \text{go_set} = \text{FALSE} \wedge \text{ready_to_sort} = \text{FALSE} \wedge \\
&\quad \text{next} \in \text{PARCELS} \wedge \text{current} \in \text{UNDEF} \\
&\quad \{ \text{sel_channel} := \text{adr_next} || \text{ready_to_sort} := \text{TRUE} || \text{go_set} := \text{TRUE} \} \\
&\quad \text{go_set} = \text{TRUE} \\
&\quad ; \\
&\quad \text{go_set} = \text{TRUE} \\
&\quad \{\text{channel} := \text{sel_channel} \parallel \text{go_set} := \text{FALSE} \}
\end{aligned}$$

go_set = FALSE
≙ ready_to_sort = FALSE ∧ next ∈ PARCELS ∧ current ∈ UNDEF
{ sel_channel := adr_next||ready_to_sort := TRUE}
sel_channel = adr_next
;
{channel := sel_channel } *channel = sel_channel*
≙ ready_to_sort = FALSE ∧ next ∈ PARCELS ∧ current ∈ UNDEF
{ ready_to_sort := TRUE}
;
{channel := adr_next }

- ***Événement Release***.

L'événement *Release* est la composition asynchrone des événements *Release_cmd* et *Release_exe*. Il est obtenu de la manière suivante : Release ≙ (Release_cmd; Release_exe).

Release_cmd = ***SELECT***
ready_to_sort = TRUE ∧
current ∈ *UNDEF* ∧
¬go_release
THEN
go_release := *TRUE* ||
ready_to_sort := FALSE
END;

;

Release_exe = ***SELECT***
go_release
THEN
current := next_p ||
next : ∈ *UNDEF* ||
go_release := FALSE
END;

⇒

Release = **SELECT**
ready_to_sort = TRUE ∧
current ∈ *UNDEF*
THEN
current := next ||
next : ∈ *UNDEF* ||
ready_to_sort := FALSE
END;

Le modèle du système automatisé est donné à l'annexe D.

Preuve : Nous vérifions sur le modèle du système automatisé les propriétés de comportement souhaitées par l'utilisateur.

Dans ce qui suit, nous continuons à raffiner séparément les modèles du contrôleur et contrôlé, puis nous procédons à une composition des deux modèles à chaque niveau du raffinement pour construire celui du système automatisé.

7.4.2 Premier raffinement du système de tri

Ce raffinement consiste à ajouter à la trieuse des éléments physiques qui avaient été omis. Le dispositif concret de tri est formé d'aiguillages interconnectés par des conduits. Un aiguillage a une entrée et deux sorties, chaque sortie est reliée par un conduit à l'entrée d'un autre aiguillage à l'exception des aiguillages dont les sorties débouchent sur des bacs de tri. Un paquet entré dans un aiguillage en sort par une des sorties, ce qui lui permet ensuite d'atteindre le nœud auquel la sortie est interconnectée et ainsi de suite jusqu'au bac de destination (Figure 7.7).

Le dispositif de tri peut être vu comme un arbre binaire dont les nœuds correspondent aux aiguillages et les arcs aux conduits reliant les nœuds. Cet arbre est équilibré, c'est-à-dire que toutes les branches de l'arbre sont d'égale longueur. Autrement dit, pour atteindre sa destination tout paquet traverse un nombre fixe d'aiguillages qui ne dépend pas de son adresse.

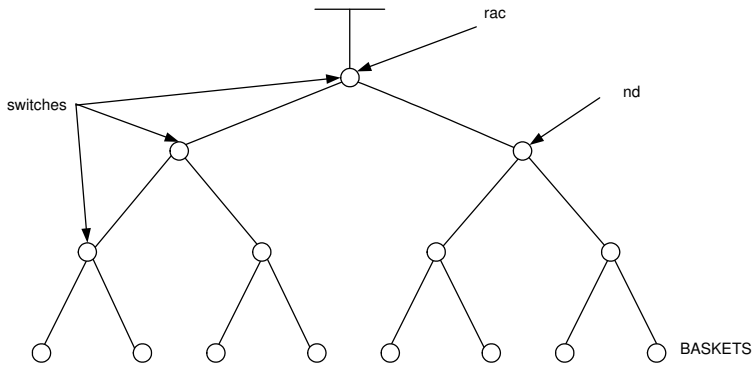


FIG. 7.7 – Premier raffinement du contrôlé

*** Raffinement du contrôlé**

Variables : dans le premier raffinement, nous ajoutons des détails sur le dispositif physique. Nous introduisons les nœuds intermédiaires que nous appelons switches et nous ajoutons les nouvelles variables *exit*, *nd*, *rac* et *lng*. La variable channel est raffinée par la fonction exit qui modélise le prochain nœud à visiter à partir du nœud courant ; nd désigne un nœud intermédiaire différent de la racine et des bacs de destination, rac la racine de l'arbre et *lng* modélise la profondeur du nœud par rapport à la racine.

Événements : nous raffinons les anciens événements et nous ajoutons deux nouveaux événements : *Set_switch_exe* et *Cross_node*.

- Événements contrôlés de la partie opérative : le nouvel événement contrôlé est Set_switch_exe. Cet événement modélise le prochain nœud à visiter calculé par son conjugué au niveau du contrôleur. Il raffine l'événement Set_channel_exe. A cet événement contrôlé, nous ajoutons les variables partagées *go_switch* et *sel_exit*, la première étant une variable de commande et la deuxième une variable d'échange de valeurs.

Set_switch_exe = **SELECT** *go_switch* **THEN** *exit(nd) := sel_exit* || *nd := sel_exit* || *go_switch := FALSE* || *lng := lng+1* **END ;**

switches = NODES - BASKETS *nd ∈ NODES* *exit ∈ switches ⇸ NODES* *lng ∈ 0..T* *T ∈ NATURAL1* **Variables partagées** *go_switch ∈ BOOL* *sel_exit ∈ NODES*

- Événements libres : le nouvel événement libre de la partie opérative est Cross_node. Cet événement modélise le passage d'un paquet au travers des nœuds. En effet, l'événement *Cross_sorting* au niveau abstrait est raffiné par la suite d'occurrences de l'événement *Cross_node* s'exécutant tant que le paquet n'est pas arrivé à destination.

Cross_node = **SELECT**
current ∈ *PARCELS* ∧
exit(nd) ∉ *BASKETS* ∧
nd ∈ *switches* ∧
lng < *T-1*
THEN
nd := *exit(nd)* ||
lng := *lng+1*
END ;

rac ∈ *NODES*
exit ∈ *switches* → *NODES*

Le modèle du premier raffinement est donné en annexe D. Ce modèle est un raffinement du modèle abstrait et toutes les obligations de preuve générées sont déchargées par le prouveur.

*** Raffinement du contrôleur**

Événements : le nouvel événement de contrôle est *Set_switch_cmd* (raffine l'événement *Set_channel_cmd*) et il est le conjugué de l'événement *Set_switch_exe*. Il permet de calculer le prochain nœud à visiter par un paquet courant. Ce choix dépend des valeurs des constantes *left_n* et *right_n* pour le nœud courant. Si le bac de destination (*adr_next*) appartient à la valeur *access(right_n(nd))*, alors le prochain nœud à visiter est le fils droit *exit(nd) := right_n(nd)*, sinon le fils gauche. Où :

- *right_n* : fonction constante qui associe à chaque nœud son successeur droit,
- *left_n* : fonction constante qui associe à chaque nœud son successeur gauche,
- *access(n)* : fonction constante qui associe à tout nœud le sous ensemble des bacs accessibles.

Set_switch_cmd =
ANY *node* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧
node ∉ *BASKETS* ∧
nd ∈ *switches* ∧
next ∈ *PARCELS* ∧
lng < *T-1* ∧ *current* ∈ *UNDEF* ∧
ready_to_sort = *FALSE*
adr_next ∈ *access(node)* ∧
go_switch = *FALSE*
THEN
sel_exit := *node* ||
go_switch := *TRUE*
END ;

rac ∈ *NODES* ∧ *rac* ∉ *BASKETS* ∧
switches = *NODES* - *BASKETS* ∧
BASKETS ⊂ *NODES* ∧
access ∈ *switches* → *POW1(BASKETS)*

Le modèle du contrôleur est donné en annexe D. Il est un raffinement du modèle précédent et toutes les obligations de preuve sont déchargées par le prouveur et sont vérifiées d'une manière automatique et intéractive.

*** Spécification de composition**

La spécification de la composition est aussi raffinée en fonction du raffinement des modèles du contrôleur et contrôlé. Cette spécification est définie comme suit :

$$CS' = CS \cup \{(\text{Set\_switch\_cmd}, \text{Set\_switch\_exe})\}$$

Set_switch_cmd =
ANY *node* **WHERE**
node $\in$ {right_n(nd), left_n(nd)}$\wedge$
node $\notin$ *BASKETS* $\wedge$
nd $\in$ *switches* $\wedge$
next $\in$ *PARCELS* $\wedge$
lng $<$ *T-1* $\wedge$ *current* $\in$ *UNDEF* $\wedge$
ready_to_sort = *FALSE*
adr_next $\in$ *access(node)* $\wedge$
go_switch = *FALSE*
THEN
sel_exit := *node* ||
go_switch := *TRUE*
END ;

Set_switch_exe = **SELECT**
go_switch $\wedge$
nd $\in$ *switches*
THEN
exit(nd) := *sel_exit* ||
nd := *sel_exit* ||
go_switch := *FALSE* ||
lng := *lng+1*
END ;

L'événement résultant obtenu est le suivant :
Set_switch $\triangleq$ (*Set_switch_cmd* ; *Set_switch_exe*)

Set_switch $\triangleq$ **ANY** *node* **WHERE** *node* $\in$ {right_n(nd), left_n(nd)}$\wedge$
nd $\in$ *switches* $\wedge$ *node* $\notin$ *BASKETS* $\wedge$
next $\in$ *PARCELS* $\wedge$ *lng* $<$ *T-1* $\wedge$
current $\in$ *UNDEF* $\wedge$ *adr_next* $\in$ *access(node)* $\wedge$
ready_to_sort = *FALSE*
THEN
exit(nd) := *node* ||
nd := *node* ||
lng := *lng+1*
END ;

Le modèle de raffinement du système automatisé est donné en annexe D.

Preuve : Nous vérifions sur le modèle du système automatisé les propriétés de comportement souhaitées par l'utilisateur. La preuve de raffinement est déduite à partir du raffinement du contrôleur et du contrôlé.

7.4.3 Deuxième raffinement du système de tri

Dans ce raffinement, nous ajoutons les détails sur les nœuds. Chaque nœud possède un capteur d'entrée, deux capteurs de sortie et une porte.

*** Raffinement du contrôlé**

Variables : dans ce raffinement, nous ajoutons les variables suivantes :

- *in* modélise le capteur d'entrée,
- *out_R* modélise un capteur de sortie droit,
- *out_L* modélise un capteur de sortie gauche,
- *gate* modélise une porte au niveau de chaque nœud. Cette variable est une fonction qui associe à chaque porte le sens d'ouverture à droite *(R)* ou à gauche *(L)*.

Ces variables sont modifiées par le modèle du contrôlé. La variable *in* est initialement fausse, mise à vrai par le passage d'un paquet. Les capteurs de sortie valent *true* quand ils détectent le passage d'un paquet et comme nous ne pouvons avoir deux sorties à la fois, les capteurs de sortie vérifient $\forall i.(i \in switches \Rightarrow \neg(out\_R(i) \wedge out\_L(i))$.

$OUT = \{R, L\}$
$gate \in NODES \rightarrow OUT$
$in \in NODES \rightarrow BOOL$
$out\_r \in switches \rightarrow BOOL$
$out\_l \in switches \rightarrow BOOL$

Événements : nous avons à la fois des événements libres et des événements contrôlés.

- Événements contrôlés : les nouveaux événements contrôlés sont *set_gate_right_exe* et *set_gate_left_exe*. Ces événements attendent de recevoir la commande de la part des événements du contrôleur *set_gate_right_cmd* et *set_gate_left_cmd* qui modélisent respectivement le positionnement de la porte des nœuds pour permettre la sortie vers la droite ou inversement.

A ces événements contrôlés, nous associons les variables partagées *sel_gate*, *go_gate_r* et *go_gate_l*. La variable *sel_gate* est une *variable d'échange* qui est modifiée par le contrôleur et passée au contrôlé. Les variables *go_gate_r* et *go_gate_l* sont des *variables de commande* qui servent à donner l'ordre au contrôlé pour commander l'ouverture des portes au niveau des nœuds.

Set_gate_right_exe $\triangleq$
SELECT
go_gate_r = TRUE
THEN
gate(nd) := sel_gate ||
go_gate_r := FALSE
END ;

Set_gate_left_exe $\triangleq$
SELECT
go_gate_l = TRUE
THEN
gate(nd) := sel_gate ||
go_gate_l := FALSE
END ;

Variables partagées
$go\_gate\_r \in BOOL$
$go\_gate\_l \in BOOL$
$sel\_gate \in OUT$

- Événements libres : les nouveaux événements libres de la partie opérative sont *Cross_right* et *Cross_left*. Ils modélisent la traversée du paquet à travers les nœuds selon le sens d'ouverture des portes.

Cross_right≙ SELECT
nd ∈ switches ∧
in(nd) = TRUE ∧
gate(nd)= R
THEN
out_l(nd) := FALSE ||
out_r(nd) := TRUE ||
in(nd) := FALSE
END;

Cross_left≙ SELECT
nd ∈ switches ∧
in(nd) = TRUE ∧
gate(nd)= L
THEN
out_l(nd) := TRUE ||
out_r(nd) := FALSE ||
in(nd) := FALSE
END;

Le modèle du contrôlé est donné en annexe D. Il est un raffinement du modèle précédent et toutes les obligations de preuve générées sont vérifiées par le prouveur.

*** Raffinement du contrôleur**

Les événements du contrôleur mettent à jour les variables partagées qui vont être utilisées par le contrôlé. Les événements de contrôle sont *set_gate_right_cmd* et *set_gate_left_cmd*. Ils commandent le positionnement de la porte au niveau de chaque nœud.

Set_gate_right_cmd =
SELECT *next ∈ PARCELS ∧*
((right_n(nd) ∈ switches ∧ adr_next ∈ access(right_n(nd))) or
(nd ∈ switches ∧ right_n(nd) ∈ BASKETS ∧
adr_next ∈ access(nd)))∧ go_gate_r = FALSE
THEN
sel_gate := R ||
go_gate_r := TRUE
END;

Set_gate_left_cmd =
SELECT *next ∈ PARCELS ∧*
((left_n(nd) ∈ switches ∧ adr_next ∈ access(left_n(nd))) or
(nd ∈ switches ∧ left_n(nd) ∈ BASKETS ∧
adr_next ∈ access(nd)))∧ go_gate_l = FALSE
THEN
sel_gate := L ||
go_gate_l := TRUE
END;

le modèle du contrôleur est donné en annexe D. Il est un raffinement du modèle précédent et toutes les obligations de preuve de raffinement sont déchargées par le prouveur.

*** Spécification de composition**

La spécification de composition est définie comme suit :

$$\begin{aligned} CS'' = CS' \cup \{&(\text{set\_gate\_right\_cmd}, \text{set\_gate\_right\_exe}),\\ &(\text{set\_gate\_left\_cmd}, \text{set\_gate\_left\_exe})\} \end{aligned}$$

En combinant les événements conjugués des deux modèles, nous obtenons les événements suivants :

$Set\_gate\_right \triangleq (Set\_gate\_right\_cmd\,;\, Set\_gate\_right\_exe)$

Set_gate_right=
SELECT *next* ∈ *PARCELS* ∧
((right_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(right_n(nd))) or*
(nd ∈ *switches* ∧ *right_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := *R*
END;

$Set\_gate\_left \triangleq (Set\_gate\_left\_cmd\,;\, Set\_gate\_left\_exe)$

Set_gate_left=
SELECT *next* ∈ *PARCELS* ∧
((left_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(left_n(nd))) or*
(nd ∈ *switches* ∧ *left_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := *L*
END;

Le modèle du système automatisé est donné en annexe D.

Preuve : Les propriétés de comportement sont vérifiées par le modèle du système automatisé.

Par la technique de composition, nous avons une meilleure interaction entre contrôleur et contrôlé dès les premières étapes de développement. Au niveau implémentation, nous pouvons récupérer le code du contrôleur que nous pourrons réutiliser dans des applications similaires.

7.5 Conclusion

Dans ce chapitre nous avons présenté une méthode de développement formel pour un contrôleur d'un système automatisé dont le comportement souhaité est connu. Le modèle du contrôleur est obtenu à l'issue d'un cycle de développement où, à chaque étape un modèle du contrôleur et un modèle du contrôlé sont obtenus par raffinement des contrôleur et contrôlé de l'étape précédente. S'il est possible de prouver que le contrôleur satisfait bien la relation de raffinement (cf : invariant de collage) qui le lie à son prédécesseur dans le cycle de développement, sa correction ne peut être établie qu'à partir du moment où on a prouvé que le système automatisé, qu'il forme avec le contrôlé a lui-même un comportement correct. Pour ce faire, nous avons dû définir une opération de composition qui tient compte de l'interaction entre le contrôleur et le contrôlé.

Une autre méthode proposée dans des travaux que nous avions menés plus tôt consistait à raffiner le système automatisé. Le système automatisé initial était construit du modèle concret du contrôlé sous contrôle d'un contrôleur minimal ou permissif. A l'issue du cycle de développement du système automatisé, une opération assez peu naturelle est nécessaire qui consiste à extraire le contrôleur.

La méthode proposée présente un certain nombre d'avantages en ce qui concerne la maîtrise de la complexité, la séparation entre contrôleur et contrôlé tout au long du développement. Elle met bien en évidence dans le contrôleur les événements qui correspondent à des actionneurs et dans le contrôlé ceux qui correspondent à des capteurs.

A défaut de disposer d'un opérateur de composition pour faire coopérer deux systèmes B, nous avons défini comment obtenir un système automatisé à partir du contrôleur et du contrôlé en indiquant les noms des événements à coupler, ainsi que l'invariant que le système construit doit satisfaire.

Nous envisageons de développer un logiciel qui effectue automatiquement la fusion et qui cachera des manipulations qui rebuteraient à coup sûr le développeur. Ce logiciel pourrait, dans une seconde étape de conception, identifier automatiquement les événements à coupler, ce qui présenterait un avantage supplémentaire pour le développeur.

Dans l'étude de cas que nous avons choisie, un seul composant a suffi pour modéliser le contrôlé. Pour des problèmes plus complexes, il est concevable d'avoir plusieurs composants ayant chacun son contrôleur. Nous aurions probablement à nous poser la question de la communication et de synchronisation entre les composants du système automatisé à travers les contrôleurs correspondants aux composants du contrôlé.

Chapitre 8

Modélisation des processus continus et patrons de conception

8.1 Introduction

Rappelons que les systèmes automatisés ou de contrôle-commande sont constitués d'un système sous contrôle (le contrôlé) et d'un système de contrôle (contrôleur) qui observe et agit sur le premier afin de contrôler son comportement. Le contrôlé évolue à la suite de l'exécution d'une action répondant à une commande du contrôleur en effectuant soit :

- un changement d'état qu'on peut considérer en fonction de la granularité du temps comme instantané (fermeture d'un relai, allumage d'une flamme, ...). Une telle action conduit au résultat attendu sans autre émission de commande. Nous dirons que l'action est à *réalisation immédiate*.
- l'activation d'un *processus continu* qui, sous réserve que les conditions d'activation soient maintenues, conduit le système vers un état escompté (température de consigne atteinte, niveau de remplissage atteint, ...). Ici, le plus souvent on souhaite stabiliser le système dès qu'un état est atteint ce qui nécessite de stopper le processus. Nous parlons d'action à *réalisation différée*.

Ces actions peuvent être modélisées par des événements. Au niveau abstrait, il n'y a pas lieu de faire la distinction entre les deux types d'événements (réalisation immédiate et réalisation différée) étant donné qu'on cherche à atteindre un état : relai fermé d'un côté, température de consigne atteinte de l'autre. Au niveau concret, s'il n'y a pas lieu de raffiner le premier type d'événement, il en va autrement du second où l'événement abstrait est raffiné en un événement qui active un processus, un événement qui observe la progression du processus tant que l'effet attendu n'est pas réalisé, un événement qui observe que l'état souhaité est atteint et un événement qui à l'avènement de l'état souhaité stoppe le processus.

Prenons l'exemple d'une cuve que nous souhaitons remplir d'une certaine quantité q de liquide. Au niveau abstrait, l'événement se définit comme suit :

Remplir = SELECT
contenu = 0
THEN
contenu := q
END ;

Au niveau raffiné, nous introduisons les quatre événements suivants : *Init_remplissage*, *Laisser_remplir*, *Detect_niveau* et *Remplir* pour raffiner l'événement abstrait *Remplir*. Ces événements s'enchaînent dans le temps.

Init_remplissage=	**Laisser_remplir**=	**Detect_niveau**=	**Remplir** =
SELECT	SELECT	SELECT	SELECT
volume = 0 ∧	*vanne = open* ∧	*volume = q* ∧	*detect = TRUE* ∧
vanne = closed	volume < q	*vanne = open*	*vanne = open*
THEN	THEN	THEN	THEN
vanne := open	*volume := volume+1*	*detect := TRUE*	*vanne := closed*
END ;	END ;	END ;	END ;

Les variables *vanne* et *volume* modélisent respectivement la vanne et le volume de remplissage. La variable *volume* raffine la variable *contenu*.

Concernant l'événement *Laisser_remplir* :

- il s'agit, bien sûr, de la discrétisation d'un processus continu,
- il ne correspond pas à une opération physique à réaliser comme une ouverture ou une fermeture de vanne mais il permet de rendre compte du passage du temps et de raisonner sur une propriété temporelle (vivacité) : une fois le processus initié, on atteindra fatalement un état où la cuve sera pleine.
- enfin, cette approche met bien en évidence le concept de temps multiforme, un des fondements de langages synchrones comme Esterel où, à chaque phénomène physique, correspond une observation du temps qui s'écoule. Nous avons souligné ce point dans [Mosbahi and Jaray, 2004a] où des processus physiques différents correspondent à des observations différentes de l'écoulement du temps.

La modélisation des processus continus peut être réalisée dans des patrons de conception. En effet, les spécifications formelles sont de plus en plus utilisées dans l'industrie et il devient intéressant de réutiliser en partie ces spécifications dans de nouveaux projets. Réutiliser une spécification formelle signifie d'abord définir un patron de spécification formelle et aussi la façon de combiner ces patrons lors de la construction d'une nouvelle application.

Nous proposons dans ce chapitre, l'utilisation des patrons de conception prouvés pour la modélisation des processus continus. Dans la section 2, nous présentons les patrons de conception prouvés et leur utilisation dans le développement des processus continus. Dans la section 3, nous étudions l'automatisation d'un cas industriel en B événementiel en utilisant les patrons de conception prouvés. Nous faisons aussi, appel à TLA$^+$ et son environnement pour prouver les propriétés de vivacité et palier la carence de B. Dans la section 4, nous appliquons la technique de composition en B pour la modélisation des systèmes automatisés complexes où de tels systèmes sont obtenus en faisant coopérer plusieurs sous systèmes plus simples. Chaque sous système est formé lui même d'un contrôleur et d'un contrôlé. A la fin du développement, nous avons un contrôleur formé de sous contrôleurs composés et d'un contrôlé formé de sous contrôlés composés. L'enchaînement entre les sous systèmes est effectué à travers le contrôleur. Enfin, nous terminons par une conclusion.

8.2 Patrons de conception prouvés

La réutilisation de spécifications pour développer un logiciel a d'abord été employée lors de la phase d'implémentation du logiciel. Elle a également été adaptée à la phase de conception.

Ainsi, il est aujourd'hui courant de réutiliser des patrons de conception exprimés en UML pour concevoir un nouveau système d'information. De nombreux patrons de conception applicables à divers domaines ont été définis. Bien que ces patrons soient connus de la plupart des concepteurs, ils n'ont pas de définition formelle précise.

En B, nous pouvons spécifier formellement la notion de patron de conception [Fowler, 1997] et nous pouvons bénéficier des outils de vérification de ces spécifications. Ainsi, un concepteur pourrait non seulement réutiliser des patrons de conception, mais également des preuves concernant ces spécifications (patrons de conception prouvés).

Un patron de conception est un ensemble de constantes, des propriétés relatives à ces constantes, de variables, d'événements et d'invariants qui vont permettre de modéliser une notion que l'on veut voir figurer dans le système que l'on construit. Ces éléments de modèle seront directement insérés dans le modèle du système en cours de construction, après instantiation des éléments du patron de conception (variables, invariants, constantes et propriétés).

Un patron de conception est prouvé lorsque les événements décrits dans ce patron vérifient les propriétés invariantes exprimées sur ces variables. De ce fait, n'importe quelle instantiation correcte du patron de conception va conduire à la production d'éléments de modèle dont la correction est acquise. Un ou plusieurs patrons peuvent être appliqués une ou plusieurs fois pour un niveau de modèle donné, produisant autant de nouveaux éléments de modélisation.

Nous nous sommes inspirés des travaux sur B et les patrons de conception [Abrial and Hallerstede, 2005] pour définir et utiliser un raffinement récurrent des processus continus dans des systèmes de production manufacturière. Ce type de raffinement consiste à raffiner une transition abstraite réalisée au niveau concret par un processus continu qu'on active dans l'état de départ de la transition et qu'on stoppe lorsque le but est atteint.

8.2.1 Patron de raffinement

Bien que ce que nous appelons patron de raffinement peut être classé parmi les patrons de conception, nous avons préféré ce nouveau terme qui rend mieux compte de nos intentions. Il s'agit ici de raffiner un ou plusieurs événements abstraits par des événements concrets pour réaliser une opération (par exemple une opération de fabrication).

A la différence des patrons de conception, nous n'utilisons pas d'invariants mais nous associons à notre patron un variant qui capte le fait que le processus continu progresse pour atteindre le but de la transition.

Ce patron introduit pour la modélisation du système automatisé une déclinaison spécifique quand nous distinguons le contrôleur et le contrôlé.

La transition associée à un événement au niveau abstrait peut être réalisée au niveau concret par un processus qu'il faut lancer, maintenir actif jusqu'à ce que l'état d'arrivée soit atteint et enfin le stopper. L'événement abstrait va être alors raffiné en quatre événements :

- activation du processus,
- progression (simulation discrète de la progression du processus),
- détection de l'état souhaité,
- arrêt du processus.

Ce patron associe à un événement abstrait les événements concrets qui le raffinent ainsi qu'un *variant* qui décroît, à chaque exécution de l'événement progression.

Il reste à prouver ce patron de raffinement. Pour cela, il faut prouver que toutes les obligations de preuve générées sont vérifiées ; et notamment, que l'état souhaité est atteint quand le variant a

atteint sa borne inférieure. Le comportement du raffinement engendré par ce patron est représenté par le diagramme de la figure 8.1.

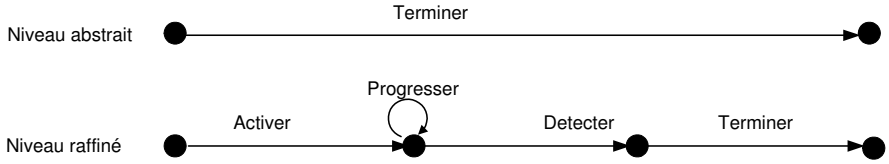


FIG. 8.1 – Diagramme de transition correspondant à l'application du patron

Le patron de raffinement ainsi défini avec un variant générique et les quatre événements *Activer*, *Progresser*, *Detecter* et *Terminer* peut être qualifié de "*meta-patron*". L'action de l'événement générique *Progresser* fait décroître le variant. Il est nécessaire de l'instancier pour obtenir un patron qui pourra être vérifié par l'environnement B.

Dans ce qui suit, nous présentons un exemple de patron de raffinement.

8.2.2 Exemple d'un patron de raffinement

Nous présentons ci-dessous un exemple d'un patron de raffinement prouvé. Il a été développé dans le cadre d'un système de production manufacturière. Il s'agit d'un patron simple de remplissage d'une cuve par une certaine quantité q de liquide (processus continu). Cet exemple est présenté à l'introduction.

Au niveau abstrait, nous avons l'événement *Remplir*.

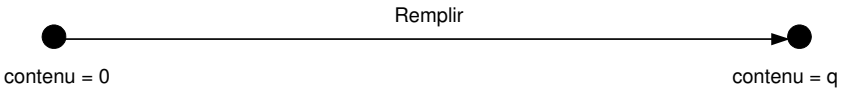


FIG. 8.2 – Événement Remplir

A un niveau raffiné, le patron de raffinement est défini par les quatre événements : *Init_remplissage*, *Laisser_remplir*, *Detect_niveau* et *Remplir* décrits précédemment. Il est aussi défini par les invariants suivants :

pat1 : $vanne \in \{closed, open\}$
pat2 : $volume \in (0..q)$
pat3 : $detect \in BOOL$

Nous obtenons le schéma de transition ci-dessous. Les flèches indiquent les relations de précédences entre événements.

L'invariant de collage reliant les variables abstraites (*contenu*) et concrètes (vanne, volume, detect) est défini comme suit :

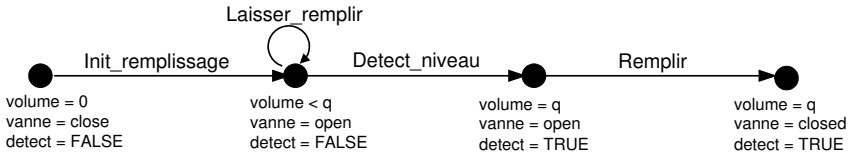


FIG. 8.3 – Diagramme de transition correspondant au patron de remplissage d'une cuve

$$(0 \leq volume < q \Rightarrow contenu = 0) \wedge$$
$$(volume = q \wedge detect = TRUE \wedge vanne = open \Rightarrow contenu = 0) \wedge$$
$$(volume = q \wedge detect = TRUE \wedge vanne = closed \Rightarrow contenu = q)$$

L'événement *Laisser_remplir* est l'événement qui modélise la progression et qui fait décroître le variant *q-volume* en exécutant l'action *volume := volume+1*.

Concernant la réutilisation des preuves, elle n'est pas automatique et prise en compte par les outils Click_n_prove ou Rodin. Ce que nous entendons par réutilisation des preuves quand on applique un patron est que : on définit pour le patron de raffinement l'ensemble des constantes, des propriétés sur les constantes, des variables, des invariants, d'invariant de collage et des événements. Au cours du raffinement, il y a des obligations de preuve de raffinement qui sont générées. Ces obligations de preuve sont vérifiées de la même manière si on applique un patron de raffinement, parce qu'on instancie à chaque fois les variables et les noms des événements.

8.3 Modélisation d'un système automatisé de production manufacturière

Dans cette section, nous proposons dans un premier temps de modéliser un cas industriel en utilisant le B événementiel. Par la suite, pour la vérification des propriétés de vivacité, nous avons recours au langage de modélisation TLA$^+$.

8.3.1 Description du système

Le système à automatiser que nous nous proposons d'étudier et dont le dispositif physique est décrit figure 8.4 permet de fabriquer un produit en malaxant, pendant une certaine durée ts, une certaine quantité d'un solide avec un mélange dosé de deux liquides. Les liquides, que nous identifierons par A et B, sont contenus dans des réservoirs (Res_A et Res_B) dont nous admettrons, pour simplifier le modèle, qu'ils ont une capacité infinie. Chaque réservoir est muni d'une vanne qui permet de verser du liquide dans une cuve C associée à un dispositif de pesée (doseur C). Deux signaux sont émis lorsque le poids de la cuve atteint respectivement W_A et W_B, ce qui correspond à deux niveaux remarquables dans la cuve m_a et m_b. La cuve C est munie d'une vanne *vc* pour vider le mélange obtenu dans un bol mixeur. Un convoyeur constitué d'un tapis roulant permet de faire tomber des briques une à une dans le mixer. Le mixeur, en position relevée pendant la fabrication, peut être incliné pour verser le produit fini à l'endroit où il doit

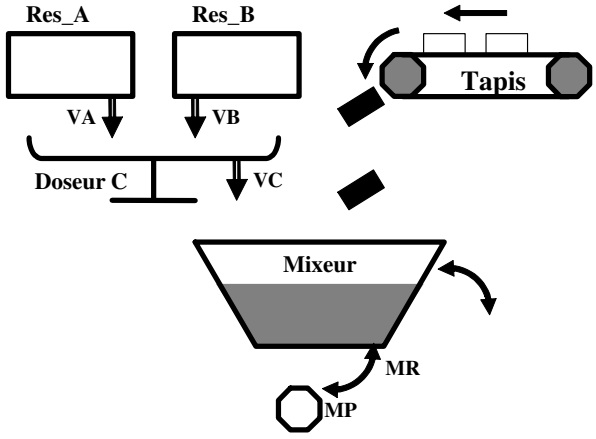


FIG. 8.4 – Mixeur

être livré (on peut penser à un camion). Le procédé de fabrication impose d'obtenir dans la cuve une quantité de liquide A de poids W_A, à laquelle on ajoute une quantité de liquide B de poids $W_B - W_A$ puis de verser ce mélange dans le bol mixeur et enfin de faire fonctionner le dernier pendant un temps t_s.

Nous nous attacherons dans la modélisation, à différencier ce que l'installation *peut* faire d'une part et ce qu'elle *doit* faire d'autre part. C'est le procédé de fabrication qui nous guide pour déterminer ce que l'installation doit faire. La modélisation de l'installation qui consiste à exprimer ce qu'elle peut faire se limitera à des comportements "raisonnables". Il est en effet possible de laisser échapper le liquide de la vanne vc pendant que le mixer est en position basse, mais ce n'est certainement pas souhaitable. Notre approche permet de réutiliser le modèle de l'installation pour d'autres fabrications, ce qui pourrait s'avérer intéressant sur le plan méthodologique dans le cadre des ateliers flexibles. Nous restreindrons le modèle de l'installation aux comportements acceptables au moyen des gardes des actions ainsi que des invariants. D'un autre côté, nous utiliserons le raffinement pour construire le modèle. L'abstraction mettra en évidence les grandes fonctions : dosage de liquide, fourniture de n briques, malaxage et livraison.

8.3.2 Les contraintes du problème

Le procédé physique doit respecter les contraintes suivantes :

Les propriétés de sûreté :
Ces propriétés sont jugées critiques et doivent rester toujours vraies, sinon des catastrophes pourraient se produire. Ces propriétés conduisent à interdire :

- d'ouvrir la vanne *vc* quand le bol mixeur n'est pas en position verticale,
- de laisser le convoyeur en marche quand le bol mixeur n'est pas en position verticale,
- de malaxer quand le bol mixeur est en position horizontale.

Les propriétés de sûreté suivantes sont liées à un objectif de fonctionnement raisonnable de l'installation. C'est ainsi qu'on s'interdit :

- d'ouvrir les vannes *va* et *vb* simultanément, faute de quoi le dosage des produits n'est pas possible,
- d'ouvrir les vannes *va* ou *vb* quand la vanne *vc* est ouverte, pour la même raison
- d'ouvrir la vanne *vc* quand le tapis est mis en marche,
- d'ouvrir la vanne *vc* pendant le malaxage,
- de mettre le convoyeur en marche pendant le malaxage,

Les propriétés de vivacité :
Les propriétés de vivacité sont des propriétés de comportement et se vérifient en analysant l'ensemble des traces possibles du système. Ces propriétés sont les suivantes :

- lorsque le système est en marche, il atteindra un état correspondant à la livraison du produit,
- quand le moteur du convoyeur est en marche, la chute d'une brique sera fatalement détectée,
- quand la vanne *va* est ouverte, nous atteindrons un état où la cuve sera remplie d'une quantité *ma*du liquide *A*,
- quand la vanne *vb* est ouverte, nous atteindrons un état où la cuve sera remplie de la quantité *mb*du liquide *mb*,
- quand la cuve contient un mélange (ma + mb) des liquides *A* et *B*, le mixeur étant vide, la vanne *vc* étant ouverte, nous atteindrons fatalement un état où la cuve est vide et le mixeur contient un mélange (ma + mb) des liquides *A* et *B*.

8.3.3 Modélisation du système

La description informelle de l'installation étant donnée, nous nous proposons de construire un programme de contrôle-commande pour piloter l'installation : (commandes d'ouverture-arrêt de vannes, de mise en route et d'arrêt de moteurs). Une des difficultés du problème est de trouver la bonne abstraction. Deux possibilités se présentent :

- modéliser les différents états de l'installation au cours d'une production caractérisant un cycle de fabrication : marche-arrêt des moteurs, ouverture-fermeture des vannes, ...
- modéliser le produit dans ses différentes phases de production en faisant abstraction des éléments de l'installation qui le contiennent : mélange de liquides, mélange de liquide et de solide, produit malaxé,... Dans le cas qui nous intéresse, l'ordre des états intermédiaires du produit est imposé par le processus de fabrication.

Nous pouvons également spécifier l'enchaînement au moyen d'invariants sur les états du produit. Notre objectif, est de modéliser le fonctionnement de l'installation. Après nous être rendu compte du grand nombre de comportements à décrire nous nous proposons de nous restreindre à des comportements en contraignant le fonctionnement par des invariants.

*** Modèle abstrait :**
Le modèle initial du problème du mixeur consiste en un cycle fabrication-livraison d'un certain produit dont les caractéristiques de fabrication seront données au cours de raffinements. La fabrication consiste en un malaxage, pendant un temps donné t_m, d'un mélange d'une quantité ma d'un liquide A, d'une quantité mb d'un liquide B et d'un nombre n de briques solides. Le dosage des liquides est effectué au moyen d'une cuve permettant de recueillir et mesurer des

quantités de liquide A et B. L'adjonction de n briques s'effectue dans le bol mixer dans lequel on a déversé le liquide.

Au cours de la fabrication, un produit est un quadruplet (qa $\mapsto$ qb $\mapsto$ nb $\mapsto$ ts) qui renseigne sur sa composition et son temps de malaxage où :

- qa désigne la quantité de liquide A,
- qb désigne la quantité de liquide B,
- nb désigne le nombre de briques tombées,
- ts désigne le temps de malaxage.

Dans l'abstraction, nous considérons que des événements qui transforment les produits en faisant passer la quantité d'un composant de la valeur (0 $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0) à la valeur du produit final (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ tm). Nous modélisons également la localisation du produit au cours de sa fabrication. Nous introduisons la variable loc_produit qui localise le produit pendant son processus de fabrication. Si le produit correspond à (0 $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0), son emplacement est indéfini (valeur *no_where*). S'il est en cours de fabrication, son emplacement est à l'intérieur du système (valeur *in*) et une fois fabriqué son emplacement est à l'extérieur (valeur *out*).

La figure 8.5 présente le modèle abstrait du système de production. L'événement *Start* met en marche le processus de fabrication et réinitialise le produit à (0 $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0). L'événement *Fabriquer* transforme le produit de (0 $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0) en (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ tm) qui correspond au produit final. L'événement Livrer met à jour la localisation du produit à *out*. L'événement *Finir* stoppe le processus de fabrication une fois le produit est livré.

Pour découvrir les erreurs qui ne peuvent être facilement découvertes par le prouveur et pour ne pas se plonger dans les preuves fausses et voir si le système se bloque (deadlock), nous avons utilisé le model checker ProB pour simuler le comportement du système.

Le modèle est valide et toutes les obligations de preuve générées sont déchargées par le prouveur d'une manière automatique. Nous trouverons dans le tableau de la figure 8.1 les statistiques sur les preuves.

*** Premier raffinement :**

Dans le premier raffinement, nous modélisons le produit dans ses différentes phases de production. En effet, le produit est fabriqué de la manière suivante :

(0 $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0) $\xrightarrow{Obtenir\_VA}$ (ma $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0) $\xrightarrow{Obtenir\_VB}$ (ma $\mapsto$ mb $\mapsto$ 0 $\mapsto$ 0) $\xrightarrow{Obtenir\_briques}$ (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ 0) $\xrightarrow{Malaxer}$ (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ tm).

Ce raffinement traduit cet enchaînement de fabrication du produit. Les nouveaux événements sont : *Obtenir_VA*, *Obtenir_VB*, *Obtenir_briques* et *Malaxer*. L'événement *Obtenir_VA* élabore le produit (ma $\mapsto$ 0 $\mapsto$ 0 $\mapsto$ 0), l'événement *Obtenir_VB* élabore le produit (ma $\mapsto$ mb $\mapsto$ 0 $\mapsto$ 0), l'événement *Obtenir_briques* fournit le produit (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ 0) et l'événement *Malaxer* élabore le produit (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ tm).

Pour obtenir le produit (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ 0), il faut ajouter n briques à (ma $\mapsto$ mb $\mapsto$ 0 $\mapsto$ 0) dans le mixeur. Pour obtenir le produit (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ tm), il faut faire fonctionner le mixeur contenant (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ 0) pendant tm unités de temps.

Le malaxage n'a pas de sens que quand la cuve est remplie du mélange (ma $\mapsto$ mb $\mapsto$ n $\mapsto$ 0), ce qui interdit, par exemple, une transition de (ma $\mapsto$ mb $\mapsto$ 0 $\mapsto$ 0) à (ma $\mapsto$ mb $\mapsto$ 0 $\mapsto$ tm). Autrement dit, il faut spécifier le produit mais aussi le processus de fabrication. Le modèle du premier raffinement est donné en annexe E.

Ce modèle est simulé avec ProB et il est valide et toutes les obligations de preuve de correction du modèle et du raffinement générées sont déchargées par le prouveur d'une manière

MODEL M_0
SETS
LIEUX = {no_where,in,out,mixeur,cuve}
CONSTANTS *ma, mb, n, tm*
PROPERTIES
$ma : NATURAL1 \wedge mb \in NATURAL1 \wedge n \in NATURAL1 \wedge tm \in NATURAL1$
VARIABLES *produit, en_marche, loc_produit*
INVARIANT
$en\_marche \in BOOL \wedge$
$produit \in (0..ma)^* (0..mb)^*(0..n)^*(0..tm) \wedge loc\_produit \in LIEUX$
INITIALISATION
$produit := (0 \mapsto 0 \mapsto 0 \mapsto 0) \;||$
$en\_marche := FALSE \;||\; loc\_produit := no\_where$
EVENTS
Start = SELECT *en_marche=FALSE*
THEN $en\_marche := TRUE \;||\; loc\_produit := no\_where \;||$
$produit := (0 \mapsto 0 \mapsto 0 \mapsto 0)$
END ;
Fabriquer = SELECT $en\_marche = TRUE \wedge$
$loc\_produit = no\_where$
THEN $produit := (ma \mapsto mb \mapsto n \mapsto tm) \;||\; loc\_produit := in$
END ;
Livrer = SELECT $loc\_produit = in$
THEN $loc\_produit := out$
END ;
Finir = SELECT $loc\_produit = out \wedge en\_marche = TRUE$
THEN $en\_marche := FALSE$
END
END

FIG. 8.5 – Modèle abstrait du mixeur.

automatique et interactive. Nous trouverons dans le tableau de la figure 8.1 les statistiques sur les preuves.

*** Deuxième raffinement :**
A ce niveau, nous raffinons la localisation du produit au cours de sa fabrication et nous ajoutons les nouveaux emplacements *cuve* et *mixeur*. En effet, la variable *loc_produit* est raffinée par *loc_prod*. Les nouveaux événements sont : l'événement *Transvaser* qui permet de vider la cuve et remplir le mixeur et l'événement *Basculer* qui modélise le déplacement du mixeur d'une position verticale vers une position horizontale. Le modèle du deuxième raffinement est donné en annexe E.

Le modèle est valide et toutes les obligations de preuve de correction du modèle et du raffinement générées sont déchargées d'une manière automatique et interactive par Click_n_Prove.

*** Troisième raffinement :**
Dans ce raffinement, nous décrivons en détail chaque étape dans le processus de fabrication du produit. L'idée que nous introduisons ici est le raffinement d'un événement atomique à réalisation différée au niveau abstrait en un ensemble d'événements qui modélisent l'action du processus qui se déroule dans le temps. En effet, le temps peut être représenté de différentes manières selon le processus effectué. Le temps peut être représenté par le remplissage ou le vidage d'une cuve, par

des briques qui se déplacent sur le convoyeur pour se rapprocher du point de chute, par un bol mixeur qui bascule d'une position verticale vers une position horizontale. Nous utilisons ainsi les patrons de conception pour modéliser et raffiner les événements à réalisation différée.

Pour l'étape de remplissage de la cuve par le liquide A, l'événement *Obtenir_VA* est raffiné par les événements suivants :

- *Ouvrir_VA* : est l'événement qui active le processus de remplissage de la cuve par le liquide A, il permet l'ouverture de la vanne *va*.
- *Remplissage_cuveAC* : est l'événement qui observe la progression du processus de remplissage de la cuve (variable *contenu(cuve)*) par le liquide A,
- *Detect_WA* : est l'événement qui correspond à l'observation du but atteint ($WA := TRUE$),
- *Obtenir_VA* : est l'événement qui désactive le processus en fermant la vanne *va*.

Ouvrir_VA = SELECT
step3 = ouvrir_VA ∧ va = close ∧ contenu(cuve) = (0↦0↦0↦0)
THEN *va := open || step3 := verser_VA*
END ;
Remplissage_cuveAC = ANY *x* WHERE
x ∈ 0..ma-1 ∧ contenu(cuve) = (x↦0↦0↦0) ∧ va=open
THEN *contenu(cuve) := (x+1↦0↦0↦0)*
END ;
Detect_WA = SELECT
contenu(cuve) = (ma↦0↦0↦0)
THEN *WA := TRUE*
END ;
Obtenir_VA = SELECT
step3 = verser_VA ∧ WA = TRUE ∧ va = open
THEN *step3 := ouvrir_VB || va := close*
END ;

L'utilisation de la variable *loc_prod* (voir modèle en annexe E) pour la description de l'installation n'est pas suffisante à ce niveau, c'est l'événement *Transvaser* qui nous en a fait prendre conscience. En effet, au cours de l'événement *Transvaser*, il y a du produit dans la cuve et dans le mixeur. Pour cela, nous ajoutons la fonction *contenu* qui à chaque emplacement $\{\text{cuve}, \text{mixeur}\}$ associe un quadruplet $(\text{q}_\text{a}, \text{q}_\text{b}, \text{nb}, \text{t}_\text{s})$. La variable *step3* modélise l'enchaînement des événements.

L'événement ***Remplissage_cuveAC*** fait décroître le *variant ma-x*. A défaut de pouvoir exprimer ce *variant* dans le système Event-B (cette option est prévue dans le projet RODIN), nous avons recours à un stratagème qui consiste à introduire au niveau abstrait, le nouvel événement qui explicite la décroissance du *variant*. Ce nouvel événement est modélisé de la manière suivante :

Remplissage_cuveAC =
BEGIN
x :|(x ∈ NATURAL ∧ ma-x < ma-x$0)
END ;

De la même manière et pour l'étape de remplissage de la cuve par le liquide B, l'événement *Obtenir_VB* est raffiné par les événements suivants :

- *Ouvrir_VB* : est l'événement qui active le processus de remplissage de la cuve par le liquide B, il permet l'ouverture de la vanne *vb*.
- *Remplissage_cuveBC* : est l'événement qui observe la progression du processus de remplissage de la cuve par le liquide B,
- *Detect_WB* : est l'événement qui correspond à l'observation du niveau atteint,
- *Obtenir_VB* : est l'événement qui désactive le processus de remplissage en fermant la vanne *vb*.

Le nouvel événement *Remplissage_cuveBC* qui fait décroître le *variant* est défini au niveau abstrait de la même manière que *Remplissage_cuveAC*.

Le patron de raffinement présenté dans la section 2 est mis en œuvre dans ce système de production. Ce patron est appliqué pour créer de nouveaux éléments de modélisation (variables, événements) et enrichir les propriétés invariantes. Il est instancié avec de nouvelles variables et événements tels que décrits ci-dessous. Les paramètres du patron sont : les noms des événements (*Init_remplissage*, *Laisser_remplir*, *Detect_niveau* et *Remplir*), les noms des variables (*vanne*, *volume*, *detect*) et les valeurs des variables (q).

Init_remplissage	$\rightsquigarrow$	*Ouvrir_VA*
Laisser_remplir	$\rightsquigarrow$	*Remplissage_cuveAC*
Detect_niveau	$\rightsquigarrow$	*Detect_WA*
Remplir	$\rightsquigarrow$	*Obtenir_VA*
volume	$\rightsquigarrow$	contenu(cuve)
vanne	$\rightsquigarrow$	va
detect	$\rightsquigarrow$	WA
q	$\rightsquigarrow$	$(\text{ma} \mapsto 0 \mapsto 0 \mapsto 0)$

Init_remplissage	$\rightsquigarrow$	*Ouvrir_VB*
Laisser_remplir	$\rightsquigarrow$	*Remplissage_cuveBC*
Detect_niveau	$\rightsquigarrow$	*Detect_WB*
Remplir	$\rightsquigarrow$	*Obtenir_VB*
volume	$\rightsquigarrow$	contenu(cuve)
vanne	$\rightsquigarrow$	vb
detect	$\rightsquigarrow$	WB
q	$\rightsquigarrow$	$(\text{ma} \mapsto \text{mb} \mapsto 0 \mapsto 0)$

Ce patron est appliqué deux fois, une fois au moment du remplissage de la cuve par le liquide A et une autre fois par le liquide B.

Pour l'étape de vidage de la cuve, l'événement *Transvaser* est raffiné par les événements suivants :

- *Init_transvaser* : est l'événement qui active le processus de vidage de la cuve, il s'effectue par l'ouverture de la vanne vc.
- *Laisser_transvaser* : est l'événement qui observe la progression du processus de vidage de la cuve et du remplissage du mixeur,
- *Detect_WC* : est l'événement qui correspond à l'observation du niveau atteint,
- *Transvaser* : est l'événement qui désactive le processus de vidage en fermant la vanne vc.

L'événement *Obtenir_briques* est aussi raffiné de la même manière par les événements suivants :

- *Init_convoyer* : est l'événement qui active le processus d'obtention des briques en mettant en marche le moteur du convoyeur.
- *Laisser_tomber_briques* : est l'événement qui observe la progression du processus qui va faire tomber les briques,

- *Detect_briques* : est l'événement qui observe que le nombre de briques est atteint,
- *Obtenir_briques* : est l'événement qui désactive le processus en arrêtant le moteur du convoyeur.

L'événement *Malaxer* est raffiné par les événements suivants :
- *Init_malaxer* : est l'événement qui active le processus de malaxage en mettant en marche son moteur.
- *Laisser_malaxer* : est l'événement qui observe la progression du temps,
- *Detect_temps* : est l'événement qui observe que le temps souhaité est atteint,
- *Malaxer* : est l'événement qui désactive le processus en arrêtant le moteur de malaxage.

L'événement *Basculer* est raffiné par les événements suivants :
- *Init_basculer* : est l'événement qui active le processus qui fait basculer le mixeur de sa position verticale vers sa position horizontale en mettant en marche son moteur.
- *Laisser_basculer* : est l'événement qui observe la progression du temps,
- *Detect_angle* : est l'événement qui détecte la position horizontale,
- *Basculer* : est l'événement qui désactive le processus en arrêtant le moteur.

L'événement *Livrer* est raffiné par les événements suivants :
- *Init_livrer* : est l'événement qui active le processus de livraison du produit en mettant en marche son moteur.
- *Laisser_livrer* : est l'événement qui observe la progression du processus de livraison,
- *Detect_produit* : est l'événement qui détecte que la livraison a été bien faite,
- *Livrer* : est l'événement qui stoppe le processus de livraison en arrêtant le moteur correspondant.

Le modèle du troisième raffinement est présenté en annexe E. Il est simulé avec ProB et prouvé par le click_n_prove et toutes les obligations de preuve de correction du modèle et de raffinement ainsi que les propriétés de sûreté sont vérifiées d'une manière automatique et intéractive. Les statistiques sur les preuves sont données dans le tableau de la figure 8.1.

Tous les processus introduits précédemment sont des processus continus qui correspondent à des observations du temps mais de différentes manières.

*** Quatrième raffinement :**
Dans ce raffinement, nous détaillons le comportement du convoyeur des briques. En effet, quand le moteur du convoyeur est mis en marche, la brique se déplace jusqu'à atteindre un détecteur, puis elle tombe dans le mixeur. De ce fait, la chute de la dernière brique doit arrêter le moteur. Nous ajoutons les événements *Avancer_brique* et *Detect_brique*. Le premier modélise le déplacement de la brique sur le tapis jusqu'à atteindre le détecteur. Le deuxième modélise la détection de la brique.

Le modèle du quatrième raffinement est présenté en annexe E. Les obligations de preuve de raffinement ainsi que les propriétés de sûreté sont vérifiées.

8.3.4 Statistiques sur les preuves

Chaque pas de raffinement est justifié formellement par des preuves. Les obligations de preuve sont générées automatiquement par l'outil. Certaines des preuves sont prouvées automatiquement et d'autres nécessitent une interaction avec l'utilisateur. Les statistiques sur les preuves sont résumées dans le tableau de la figure 8.1.

Raffinement	*Nombre d'obligations de preuve*	*Nombre de preuves automatiques*	*Nombre de preuves interactives*
0	12	12	0
1er	43	39	4
2ème	31	24	7
3ème	344	280	64
4ème	8	7	1

TAB. 8.1 – Statistiques sur les preuves.

Dans ce qui suit, nous nous intéressons à la vérification des propriétés de vivacité sur ce système automatisé de production.

8.3.5 Vérification des propriétés de vivacité

Nous présentons dans cette section les propriétés de vivacité à vérifier par notre système. Nous commençons d'abord par donner les modèles B temporels obtenus par l'ajout de ce type de propriétés, par la suite, nous donnons les modules TLA$^+$ correspondant aux modèles B temporels. Enfin, nous vérifions ces propriétés par le model checker TLC.

* Modèles B temporels

Nous donnons dans ce qui suit, les propriétés de vivacité à vérifier au niveau du modèle abstrait ainsi que les différents raffinements.

- **Modèle abstrait :**
  Nous donnons une équité faible pour l'exécution des événements *Fabriquer* et *Livrer* : *WF(Fabriquer)* $\wedge$ *WF(Livrer)*. Cette condition d'équité apparaît au niveau de la clause FAIRNESS.
  Nous avons à vérifier la propriété suivante : lorsque le système est mis en marche, il atteindra un état où le produit est livré, cette propriété est décrite par la formule suivante :
  $en\_marche = TRUE \wedge produit = (0 \mapsto 0 \mapsto 0 \mapsto 0) \leadsto produit = (ma \mapsto mb \mapsto n \mapsto tm) \wedge loc\_produit = out$.
  Cette propriété est ajoutée au niveau de la clause EVENTUALITY.

- **Modèle du premier raffinement :**
  Nous donnons une équité faible pour l'exécution des événements *Obtenir_VA*, *Obtenir_VB*, *Obtenir_briques*, *Fabriquer* et *Livrer* :
  $WF(Obtenir\_VA) \wedge WF(Obtenir\_VB) \wedge WF(Obtenir\_briques) \wedge WF(Fabriquer) \wedge WF(Livrer)$.
  Par cette condition d'équité, nous garantissons la préservation des propriétés de vivacité.
  Nous avons à vérifier la propriété suivante : lorsque le système est à l'étape *commencer*, il atteindra un état où le produit sera livré. Cette propriété est décrite par la formule suivante : $step = commencer \wedge prod = (0 \mapsto 0 \mapsto 0 \mapsto 0) \rightsquigarrow loc1\_produit = out \wedge prod = (ma \mapsto mb \mapsto n \mapsto tm)$.

- **Modèle du deuxième raffinement :**
  Nous donnons une hypothèse d'équité faible que nous ajoutons à la clause FAIRNESS.
  $WF(Obtenir\_VA) \wedge WF(Obtenir\_VB) \wedge WF(Transvaser) \wedge WF(Obtenir\_briques) \wedge WF(Fabriquer) \wedge WF(Basculer) \wedge WF(Livrer)$.

- **Modèle du troisième raffinement :**
  Nous donnons dans ce modèle une équité faible aux événements suivants : *Ouvrir_VA, Detect_WA, Obtenir_VA, Ouvrir_VB, Detect_WB, Obtenir_VB, Init_transvaser, Detect_WC, Transvaser, Init_convoyeur, Detect_briques, Obtenir_briques, Init_malaxer, Detect_temps, Fabriquer, Init_basculer, Detect_angle, Basculer, Init_livrer, Detect_produit et Livrer*. Nous avons à vérifier les propriétés suivantes présentées dans la section 1.3.2 :

  - lorsque le système est mis en marche, il atteindra un état où le produit est livré, cette propriété est décrite par :
    $step3 = Commencer \wedge contenu(out) = (0 \mapsto 0 \mapsto 0 \mapsto 0) \rightsquigarrow contenu(out) = (\mathrm{ma} \mapsto \mathrm{mb} \mapsto \mathrm{n} \mapsto \mathrm{tm})$.
  - quand la cuve est remplie par la quantité (ma + mb) et le mixeur est vide, nous atteindrons un état où la cuve est vide et le mixeur est plein. Cette propriété est décrite par :
    $contenu(cuve) = (ma \mapsto mb \mapsto n \mapsto tm) \wedge contenu(mixeur) = (0 \mapsto 0 \mapsto 0 \mapsto 0) \rightsquigarrow contenu(cuve) = (0 \mapsto 0 \mapsto 0 \mapsto 0) \wedge contenu(mixeur) = (ma \mapsto mb \mapsto n \mapsto tm)$.
  - quand la vanne *va* est ouverte, la cuve sera remplie d'une quantité *ma* du liquide *A*. Cette propriété est décrite par : $va = open \rightsquigarrow contenu(cuve) = (ma \mapsto 0 \mapsto 0 \mapsto 0)$.
  - quand la vanne *vb* est ouverte, la cuve sera remplie d'une quantité *mb* du liquide *B*. Cette propriété est décrite par : $vb = open \rightsquigarrow contenu(cuve) = (ma \mapsto mb \mapsto 0 \mapsto 0)$.
  - quand la vanne *vc* est ouverte, le mixeur sera rempli d'une quantité *ma+mb* des liquides *A* et *B*. Cette propriété est décrite par : $vc = open \rightsquigarrow contenu(mixeur) = (ma \mapsto mb \mapsto 0 \mapsto 0)$.

  Ces propriétés sont ajoutées au niveau de la clause EVENTUALITY.

- **Modèle du quatrième raffinement :**
  Nous ajoutons dans ce modèle une équité faible aux deux nouveaux événements *Avancer_brique* et *Detect_brique*. Nous avons à vérifier la propriété suivante : quand le moteur du convoyeur est mis en marche, fatalement une brique sera détectée, cette propriété est décrite par :
  $moteur\_convoyeur = on \Rightarrow \Diamond\ detect = TRUE$.
  Nous ajoutons cette propriété au niveau de la clause EVENTUALITY.

*** Les modules TLA$^+$**

Les modules TLA$^+$ obtenues sont présentés en annexe E. Ces propriétés de vivacité sont vérifiées par le model checker TLC.

Dans ce qui suit, nous appliquons la technique de composition en B pour le développement de contrôleurs des systèmes automatisés complexes.

8.4 Décomposition et composition en B pour la modélisation des systèmes automatisés complexes

Dans ce qui précède nous avons construit un modèle d'un système automatisé de fabrication manufacturière. Nous avons appliqué à plusieurs reprises un patron de raffinement pour raffiner un événement abstrait qui réalise une transition du système abstrait en un ensemble d'événements correspondant au démarrage d'un processus, à son maintien en activité et à sa terminaison quand le but est atteint.

Dans cette section, nous proposons de :

- décomposer la fabrication en différentes phases, chaque phase étant effectuée par un sous système automatisé,
- décomposer chacun de ces sous systèmes en contrôleur et contrôlé,
- modéliser l'enchaînement des sous systèmes de fabrication,
- revisiter le patron de raffinement pour tenir compte de la distribution des événements entre le contrôleur et le contrôlé.

L'intérêt de cette démarche, déjà évoquée auparavant, est le fait, qu'à l'issue du processus de développement, nous disposons du contrôleur sans avoir à l'extraire du système automatisé. Nous faisons ainsi usage d'un moyen de maîtrise de la complexité et nous nous donnons la possibilité de réutiliser des sous-systèmes automatisés pour d'autres fabrications (utilité dans la conception d'ateliers flexibles). A la fin du processus de développement, nous avons un contrôleur formé de la composition des différents contrôleurs.

Dans un système automatisé, contrôleur et contrôlé interagissent en formant une boucle de contrôle : au niveau abstrait le contrôleur émet une commande et le contrôlé l'exécute en réalisant une transition d'états. Sachant qu'au niveau concret, le contrôlé va activer un processus qui prendra un certain temps pour réaliser la transition, nous introduisons une variable d'acquittement pour permettre au contrôlé d'informer le contrôleur de la réalisation de la transition. La transition abstraite du contrôlé est raffinée par les événements suivants :

- un événement qui active le processus et qui modélise un effecteur (événement *start_op* dans la figure 8.7),
- un événement qui a pour garde la condition d'activation du processus et qui simule sa progression en faisant décroître un variant, aussi longtemps que les conditions d'activité sont maintenues (événement *progress* dans la figure 8.7),
- un événement qui modélise un capteur, qui détecte et signale au contrôleur que l'état souhaité est atteint (dans cet état le variant a atteint sa borne inférieure) (événement *detect* dans la figure 8.7),
- un événement activable par le contrôleur pour stopper le processus (événement *arrêt_op* dans la figure 8.7).

Les premier et dernier événements sont sous le contrôle du contrôleur. Il faut noter qu'il est essentiel d'arrêter le processus dès que l'état souhaité est atteint. Au niveau abstrait, le lancement du processus, la progression, la détection de l'état souhaité et l'arrêt sont des événements qui peuvent être contractés en un seul événement dont la garde est liée à la garde du lancement du processus et dont la post-condition est liée à l'arrêt. A ce niveau abstrait, le contrôleur qui commande une action met à vrai la garde de l'événement qui doit réagir.

De la même manière, au niveau du contrôleur, l'événement est raffiné en deux événements : un événement qui commande le déclenchement du processus et l'autre, qui commande l'arrêt du processus une fois l'état souhaité atteint.

Nous donnons dans les figures 8.6 et 8.7, le comportement des processus continus au niveau abstrait et raffiné.

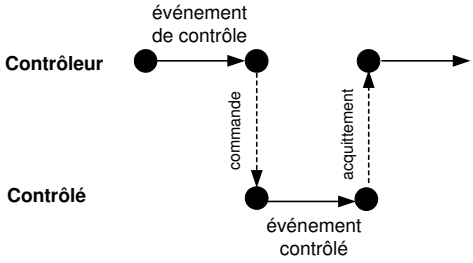


FIG. 8.6 – Représentation des processus continus au niveau abstrait.

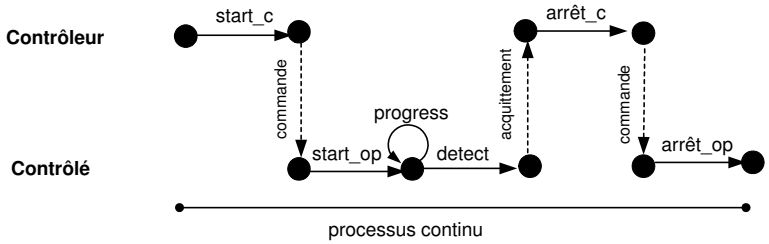


FIG. 8.7 – Représentation des processus continus au niveau raffiné.

Patron pour le raffinement d'un couplage contrôleur-contrôlé :

- Au niveau abstrait :
  - Côté contrôleur, l'événement de commande est de la forme :
    ordre_c = SELECT cmd = FALSE $\wedge$ P_c THEN cmd := TRUE || GS_c END ;
    où P_c est un prédicat d'état égal à vrai quand la décision d'émettre la commande doit être prise, cmd une variable de commande et GS_c une substitution généralisée. Notons que l'exécution du prochain événement du contrôleur doit prendre en compte la terminaison de l'opération au niveau du contrôlé ($ack = TRUE$).
  - Côté contrôlé, l'événement d'exécution de la commande est :
    execution_op = SELECT cmd = TRUE THEN

cmd := FALSE || ack := TRUE || GS_{op} END ;

où GS_{op} est une substitution généralisée et *ack* est une variable d'acquittement. Cet événement exécute la commande du contrôleur et met à vrai une variable d'acquittement.

- Au niveau raffiné :
  Le raffinement de cette interaction, dans le cas où la réalisation de la commande prend du temps se définit comme suit :
  - Côté contrôleur :
    start_c = SELECT cmd_r = FALSE ∧ P'_c THEN cmd_r := TRUE || GS'_c END ;
    arrêt_c = SELECT *signal* = TRUE ∧ *cmd* = FALSE ∧ P''_c
    THEN *cmd* := TRUE || *signal* := FALSE || GS''_c END ;
    où cmd_r est une variable de commande au niveau raffiné ; P'_c, P''_c sont des prédicats d'états ; GS'_c, GS''_c sont des substitutions généralisées et *signal* une variable modélisant un signal émis par le contrôlé quand il détecte l'état souhaité. Elle raffine la variable *ack* au niveau abstrait.
  - Côté contrôlé :
    start_op = SELECT cmd_r = TRUE THEN GS'_{op} || cmd_r := FALSE END ;
    progress = SELECT g_p THEN GS_p END ; (cet événement fait décroître un variant)
    detect = SELECT g_d (variant atteint sa borne inférieure) THEN *signal* := TRUE END ;
    arrêt_op = SELECT *cmd* = TRUE THEN GS_a || *cmd* := FALSE END ;
    où GS'_{op}, GS_p et GS_a sont des substitutions généralisées et g_p, g_d sont des prédicats d'état. La substitution GS_a correspond à l'arrêt du processus et la substitution GS'_{op} établit la garde g_p.

Les événements *arrêt_c* et *arrêt_op* au niveau raffiné correspondent aux événements *ordre_c* et *execution_op* au niveau abstrait.

8.4.1 Décomposition du système automatisé

La décomposition d'un système automatisé en un ensemble de sous-systèmes automatisés est une tâche délicate du cycle de développement. Cette décomposition décrit en détail ces sous-systèmes ainsi que les interfaces supportant leurs interactions. Ces interfaces sont caractérisées par les protocoles contrôlant les flux de données échangés, les contraintes de sûreté ainsi que les propriétés temporelles sur de tels échanges. Notre décomposition est dirigée par le procédé. Chaque sous-système réalise une fonction (par exemple dans notre étude de cas, une étape de la fabrication) et il contient au moins un actuateur (actionneur) et un détecteur (capteur).

Les différents éléments d'un sous système automatisé sont :

- le dispositif physique,
- le contrôleur associé,
- l'interface interne supportant la communication entre le contrôlé et son contrôleur,
- l'interface externe supportant les interactions entre les différents sous-systèmes via leurs contrôleurs.

Pour des raisons de modularité et de réutilisation, chaque contrôleur doit avoir un événement qui détecte la terminaison de toutes les actions que le sous-système automatisé doit réaliser

(événement *Terminer*). Cet événement est activé quand toutes les gardes des autres événements sont fausses.

Réaliser la composition séquentielle des différents sous-systèmes peut s'effectuer simplement en disposant d'un jeton : une variable qui, à tout moment, contient le numéro d'ordre dans le séquencement des sous-systèmes. Appelons *hand* cette variable. Pour réaliser la composition, on renforce les gardes des événements du sous-système i par le prédicat $hand = i$; et on complète l'action de l'événement *Terminer* du contrôleur i par $hand := i + 1$ sauf si c'est le dernier sous-système.

8.4.2 Application de la technique de décomposition au cas du mixeur

Dans cette section, nous appliquons la technique de composition que nous avons proposée dans le chapitre précédent pour le cas du problème de mixeur. Dans le cadre de la démarche de développement, nous proposons de décomposer le système automatisé en sous systèmes et de faire coopérer ces sous-systèmes pour réaliser le système automatisé global. Notre démarche propose aussi de décomposer chaque sous-système en un contrôleur et son contrôlé.

Au niveau de chaque sous-système, nous construisons le contrôleur par raffinements successifs ainsi que le modèle du contrôlé à un niveau d'abstraction en adéquation avec le contrôleur. Nous composons ensuite le contrôleur et le contrôlé pour obtenir un modèle du sous-système automatisé afin de vérifier si le comportement de ce dernier est conforme à ce qui est attendu. La composition des sous-systèmes automatisés est, ici, essentiellement séquentielle mais dans un cas plus général, nous pourrions avoir à spécifier des sous-systèmes qui peuvent ou doivent fonctionner en parallèle.

Il convient ici de remarquer que l'enchaînement entre les sous systèmes ne concerne que les contrôleurs. Rappelons aussi que notre objectif est de construire un contrôleur.

Dans l'étude de cas du mixer, on s'aperçoit assez facilement qu'il est possible de décomposer l'installation en différentes parties :

- les réservoirs et la cuve de pesée munis de vannes pour doser et fournir un mélange de liquides,
- une bande transporteuse (convoyeur de briques) pour fournir des solides,
- un bol mixeur pour mixer le produit et le livrer

Chacune des parties permet de réaliser une fonction. Chacune des fonctions peut être réalisée par un sous système automatisé. On obtiendra ainsi un contrôleur pour chacune des parties. Le pilotage de l'ensemble de l'installation consiste ensuite à coordonner l'activité des différents contrôleurs.

Dans le cas du mixeur, la coordination consiste en un enchaînement séquentiel des opérations réalisées par chacun des contrôleurs. Ici, l'opérateur de séquentialisation suffit pour décrire la coordination des contrôleurs. Dans un cas plus général, il pourrait être utile de disposer d'un langage de composition utilisant les opérateurs de composition parallèle, de choix indéterministe, ...(CSP joue ce rôle dans le travail de Treharne [Treharne *et al.*, 2003]). C'est un travail que nous envisageons de poursuivre.

* Modèle abstrait

Au niveau abstrait, nous ne considérons que deux sous-systèmes modélisant l'application :

- le sous-système de fabrication : il effectue le dosage, la fourniture des briques et le malaxage des produits bruts,

- le sous-système de livraison : il interagit avec le premier pour recevoir le produit final.

1. **Sous-système de fabrication** S_{fab} **:**

- Modèle du contrôleur :

Le contrôleur du sous-système de fabrication contient les événements *Start_C* et *Fabriquer_C*. Le premier met en marche le processus de fabrication et le deuxième commande la fabrication du produit.

Start_C = SELECT *en_marche* = FALSE ∧ *go_start* = FALSE
THEN *en_marche* := TRUE || *go_start* := TRUE
END ;
Fabriquer_C = SELECT *en_marche* = TRUE ∧ *go_fabriquer* = FALSE ∧
loc_produit = *no_where* ∧ *init_done* = TRUE
THEN *qa* :=*ma* || *qb* := *mb* || *nb* := *n* || *t* := *ts* ||
go_fabriquer := TRUE || *init_done* := FALSE
END ;

Les variables *go_start* et *go_fabriquer* sont des variables de commande et *init_done* et *fabrication_done* sont des variables d'acquittement. Les variables *qa, qb, nb* et *t* sont partagées entre le contrôleur et le contrôlé.

- Modèle du contrôlé :

Le modèle du contrôlé est formé par les deux événements *Start_OP* et *Fabriquer_OP*. Ces deux événements sont les conjugués des événements *Start_C* et *Fabriquer_C*. Chacun des événements du contrôlé reçoit l'ordre de son exécution de la part du contrôleur (au moyen des variables de commande) et émet un acquittement (au moyen des variables d'acquittement) pour informer la terminaison de l'exécution d'un événement donné.

Start_OP=
SELECT *go_start* = TRUE
THEN *produit* := *(0↦0↦0↦0)* || *loc_produit* := *no_where* ||
go_start := FALSE || *init_done* := TRUE
END ;
Fabriquer_OP =
SELECT *go_fabriquer* = TRUE
THEN *produit* := *(qa↦qb↦nb↦t)* || *loc_produit* := *in* ||
fabrication_done := TRUE || *go_fabriquer* := FALSE
END ;

- Modèle du sous-système automatisé de fabrication :

Soit CS_f l'ensemble de conjugaison correspondant aux événements du contrôleur et du contrôlé du sous-système automatisé de fabrication. Cet ensemble se présente comme suit :

CS_f = {*(Start_C,Start_OP), (Fabriquer_C,Fabriquer_OP)*}

Notons que les variables de commande, d'échange de valeurs et d'acquittement disparaissent du modèle du sous-système automatisé. Ce modèle doit vérifier l'invariant indiquant que lorsque le système est mis en marche alors le produit sera fabriqué et il aura la forme *(ma↦mb↦n↦tm)*. Le modèle du sous-système automatisé de fabrication contient les événements suivants :

Start = SELECT *en_marche=FALSE*
THEN *en_marche :=TRUE* || *loc_produit := no_where* ||
produit := (0↦0↦0↦0)
END ;
Fabriquer = SELECT *en_marche = TRUE* ∧
loc_produit = no_where
THEN *produit := (ma↦mb↦n↦tm)* || *loc_produit := in*
END

2. **Sous-système de livraison** S_{liv} **:**

- Modèle du contrôleur :

Le contrôleur du sous-système de livraison contient les événements *Livrer_C* et *Finir_C*. Le premier commande la livraison du produit et le deuxième arrête le processus de fabrication.

Livrer_C = SELECT *go_livrer* = FALSE ∧ *loc_produit* = in
THEN *go_livrer* := TRUE
END ;
Finir_C = SELECT *livraison_done* = TRUE ∧ *loc_produit* = *out*
THEN *en_marche* := FALSE || *livraison_done* := FALSE
END ;

La variable *go_livrer* est une variable de commande alors que la variable *livraison_done* est une variable d'acquittement.

- Modèle du contrôlé :

Le modèle du contrôlé contient l'événement *Livrer_OP*. Cet événement est le conjugué de *Livrer_C*. Il est défini comme suit :

Livrer_OP =
SELECT *go_livrer* = TRUE
THEN *loc_produit* := *out* || *livraison_done* := TRUE || *go_livrer* := FALSE
END ;

- Modèle du sous-système automatisé de livraison :

Soit CS_l l'ensemble de conjugaison correspondant aux événements du contrôleur et du contrôlé du sous-système automatisé de livraison. Cet ensemble se présente comme suit :

CS_l = {(Livrer_C, Livrer_OP)}

Le modèle du sous-système automatisé de livraison contient les événements suivants :

Livrer = SELECT *loc_produit = in*
THEN *loc_produit := out*
END ;
Finir = SELECT *loc_produit = out* ∧ *en_marche = TRUE*
THEN *en_marche := FALSE*
END

3. **Composition séquentielle des deux sous systèmes automatisés :**

Le modèle du système automatisé du mixeur (S_{mix}) est une composition séquentielle de ses deux sous systèmes de fabrication et de livraison : $S_{mix} = S_{fab}; S_{liv}$. Aussi, le modèle du contrôleur du système automatisé est une composition séquentielle des deux contrôleurs des sous-systèmes de fabrication et de livraison.

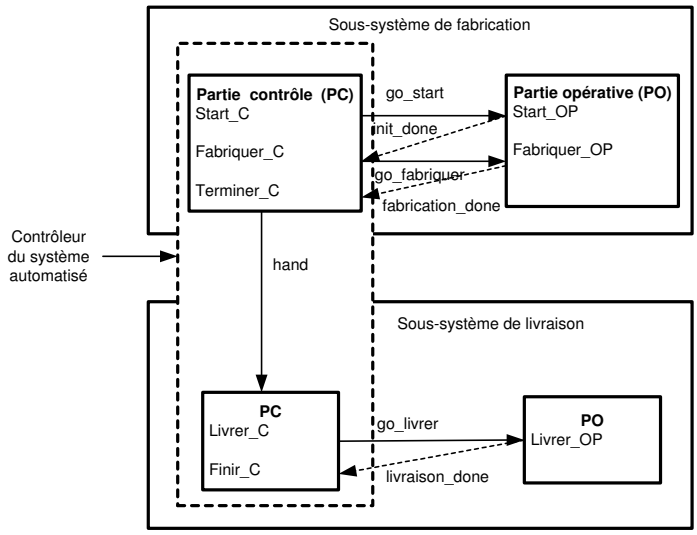


FIG. 8.8 – Les sous-systèmes du modèle abstrait.

Dans la figure 8.8, le sous-système de fabrication dispose de ses propres contrôleur et contrôlé, d'une interface interne constituée des variables de commande et des variables d'acquittement et d'une interface externe formée par la variable *hand*. Cette variable permet l'enchaînement séquentiel des événements au niveau des contrôleurs des sous-systèmes.

Le modèle du contrôleur du système automatisé est présenté à l'annexe F et toutes les obligations de preuves générées sont vérifiées. Nous donnons les statistiques sur les preuves dans le tableau de la figure 8.3.

De la même manière, le modèle abstrait du contrôlé est présenté à l'annexe F et toutes les obligations de preuves générées sont vérifiées. Les statistiques sur les preuves sont données dans le tableau de la figure 8.2.

8.4.3 Premier raffinement

Dans le premier raffinement, nous raffinons le sous-système de fabrication. Ce sous-système est composé lui-même des trois sous-systèmes suivants :

- le sous-système de dosage : effectue le dosage,
- le sous-système de convoyeur : effectue l'approvisionnement en briques,
- le sous-système de malaxage : effectue le malaxage.

1. **Sous-système de dosage S_{dos} :**

   - Modèle du contrôleur :

   Le contrôleur du sous système de dosage est formé par les événements *Start_C*, *Obtenir_VA_C* et *Obtenir_VB_C*.Les événements *Obtenir_VA_C* et *Obtenir_VB_C*

permettent de commander respectivement le remplissage de la cuve par les liquides A et B.

- Modèle du contrôlé :

Le contrôlé du sous système de dosage contient les événements *Start_OP*, *Obtenir_VA_OP* et *Obtenir_VB_OP* qui sont les conjugués des événements *Start_C*, *Obtenir_VA_C* et *Obtenir_VB_C*.

- Modèle du sous système automatisé de dosage :

La spécification de composition du sous système de dosage (CS_d) est définie comme suit :

CS_d = {*(Start_C,Start_OP), (Obtenir_VA_C,Obtenir_VA_OP), (Obtenir_VB_C,Obtenir_VB_OP)*}

L'objectif de ce sous-système est d'obtenir dans une première étape de fabrication le produit *(ma↦0↦0↦0)* puis le produit *(ma↦mb↦0↦0)*.

2. **Sous-système de convoyeur** S_{conv} **:**

- Modèle du contrôleur :

Le contrôleur du sous-système de convoyeur contient l'événement *Obtenir_briques_C*. Cet événement commande la chute de n briques.

- Modèle du contrôlé :

Le contrôlé du sous-système de convoyeur contient l'événement *Obtenir_briques_OP*.

- Modèle du sous-système automatisé de convoyeur :

La spécification de composition du sous-système de convoyeur (CS_c) est définie comme suit :

CS_c = {*(Obtenir_briques_C,Obtenir_briques_OP)*}

Le rôle de ce sous-système est d'effectuer l'approvisionnement de n briques et obtenir le produit *(ma↦mb↦n↦0)*.

3. **Sous-système de malaxage** S_{mal} **:**

- Modèle du contrôleur :

Le contrôleur du sous système de malaxage contient l'événement *Fabriquer_C*. Cet événement modélise le malaxage du produit pendant *tm* unités de temps.

- Modèle du contrôlé :

Le contrôlé du sous système de malaxage contient l'événement *Fabriquer_OP*.

- Modèle du sous système automatisé de malaxage :

La spécification de composition du sous-système de malaxage (CS_m) est définie comme suit :

CS_m = {*(Fabriquer_C,Fabriquer_OP)*}

L'objectif de ce sous-système est d'effectuer le malaxage des liquides et des solides versés pendant tm unités de temps et obtenir le produit *(ma↦mb↦n↦tm)*.

4. **Composition séquentielle des sous systèmes automatisés :**

Le modèle du système automatisé du mixeur est une composition séquentielle de ses sous-systèmes de dosage, de convoyeur, de malaxage et de livraison : $S_{mix} = S_{fab}; S_{liv} = S_{dos}; S_{conv}; S_{mal}; S_{liv}$.

Chaque sous-système possède ses propres contrôleur et contrôlé (figure 8.9). L'interaction entre contrôleur et contrôlé du même sous-système est réalisée par des variables partagées

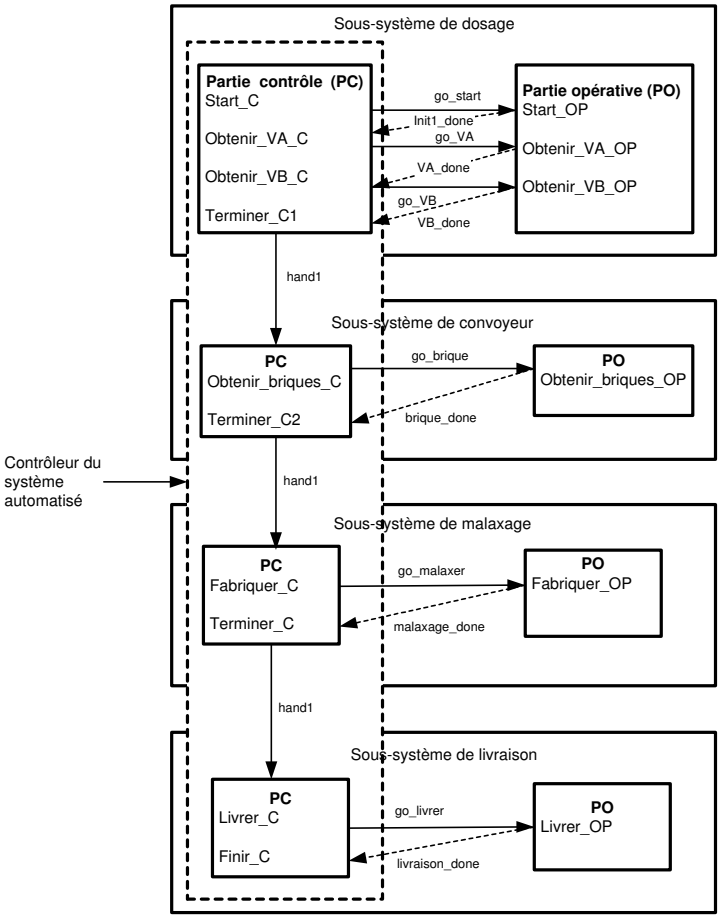


FIG. 8.9 – Les sous systèmes du premier raffinement.

(variables de commande et d'acquittement). La communication entre les sous-systèmes est effectuée par la variable *hand1* raffinant la variable *hand*.

Dans la figure 8.9, les événements *Terminer_C1* et *Terminer_C2* permettent de rendre compte de la terminaison de toutes les actions au niveau du contrôlé respectivement des sous-système de dosage et de convoyeur.

Les nouvelles variables de commande sont :

- *go_VA* : pour commander l'opération de remplissage de la cuve par le liquide A,
- *go_VB* : pour commander le remplissage de la cuve par le liquide B,
- *go_brique* : pour commander le versement des briques,
- *go_malaxer* : pour commander le malaxage pendant un temps t du mélange.

Les nouvelles variables d'acquittement sont :

- *VA_done* : permet d'informer le contrôleur de l'achèvement de l'exécution de l'action de remplissage de la cuve par le liquide A.
- *VB_done* : permet de signaler au contrôleur l'achèvement de l'opération de remplissage de la cuve par le liquide B,

- *brique_done* : permet de signaler au contrôleur la fin de l'opération de versement des briques,

- *malaxage_done* : permet de signaler au contrôleur la fin du malaxage.

Le modèle du contrôleur du système automatisé est une composition séquentielle des contrôleurs des sous-systèmes de dosage, de convoyeur, de malaxage et de livraison. Ce modèle est présenté à l'annexe F. Ce modèle est bien un raffinement du modèle précédent et toutes les obligations de preuve sont vérifiées d'une manière automatique et interactive. Les statistiques sur les preuves sont données dans le tableau de la figure 8.3.

Le modèle du contrôlé du système automatisé est donné à l'annexe F. Ce modèle est bien un raffinement du modèle précédent et toutes les obligations de preuve sont vérifiées d'une manière automatique et interactive. Les statistiques sur les preuves sont données dans le tableau de la figure 8.2.

8.4.4 Deuxième raffinement

A ce niveau de développement, nous raffinons les sous-systèmes de dosage et de livraison. Au niveau du premier, nous ajoutons l'événement *Transvaser* qui permet de vider la cuve et remplir le mixeur avec la quantité (*ma+mb*). Au niveau du deuxième sous-système, nous ajoutons l'événement *Basculer* permettant de basculer le mixeur de sa position verticale vers sa position horizontale.

Composition séquentielle des sous systèmes automatisés :

- Modèle du contrôleur :
Le modèle du contrôleur du système automatisé est obtenu par la composition séquentielle des contrôleurs de ses sous-systèmes. Ce modèle est donné par la figure 8.10.

Dans ce modèle, nous ajoutons les événements *Transvaser_C* et *Basculer_C* respectivement aux contrôleurs des sous-systèmes de dosage et de livraison. Nous ajoutons également les variables *go_transvaser*, *go_basculer*, *transvaser_done* et *basculer_done*. Les deux premières sont des variables de commande et les deux dernières sont des variables d'acquittement. La coordination entre les contrôleurs des différents sous-systèmes est effectuée par la variable *hand1*.

Le modèle du contrôleur est donné à l'annexe F. Ce modèle est un raffinement du modèle précédent et toutes les obligations de preuve générées sont vérifiées.

- Modèle du contrôlé :
Le modèle du contrôlé du système automatisé est aussi obtenu par la composition séquentielle des contrôlés de ses sous-systèmes.

Dans ce modèle, nous ajoutons de la même manière les événements *Transvaser_OP* et *Basculer_OP* respectivement aux contrôlés des sous-systèmes de dosage et de livraison. Ces événements reçoivent les ordres respectivement des événements *Transvaser_C* et *Basculer_C*.

Le modèle du contrôlé est présenté à l'annexe F et toutes les obligations de preuve sont vérifiées.

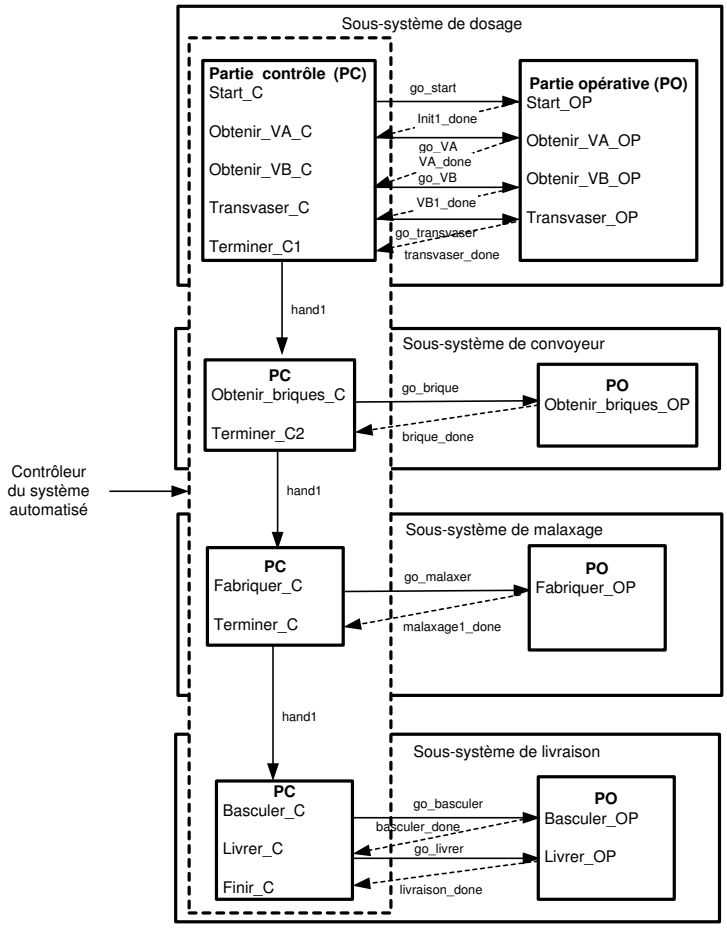


FIG. 8.10 – Les sous-systèmes du deuxième raffinement.

8.4.5 Troisième raffinement

Comme nous avons vu précédemment, dans ce raffinement, nous décrivons en détail chaque étape dans le processus de fabrication du produit.

Les différents sous systèmes sont donnés figure 8.11.

- Modèle du contrôleur :

Pour tout événement à réalisation différée, nous associons dans le modèle du contrôleur deux événements : l'un déclenche l'exécution du processus correspondant du contrôlé et l'autre le stoppe. Par exemple, l'événement *Obtenir_VA_C* est raffiné en deux événements : *Ouvrir_VA_C* et *Obtenir_VA_C*. Le premier commande l'ouverture de la vanne *va* et le deuxième commande sa fermeture une fois le volume souhaité détecté. Nous ajoutons également de nouvelles variables de commande et d'acquittement. La coordination entre les contrôleurs des sous-systèmes est mise en œuvre par la variable *hand1*.

Le modèle complet du contrôleur est donné à l'annexe F. Ce modèle est un raffinement du modèle précédent. Les statistiques sur les preuves sont données dans le tableau de la figure 8.3.

- Modèle du contrôlé :

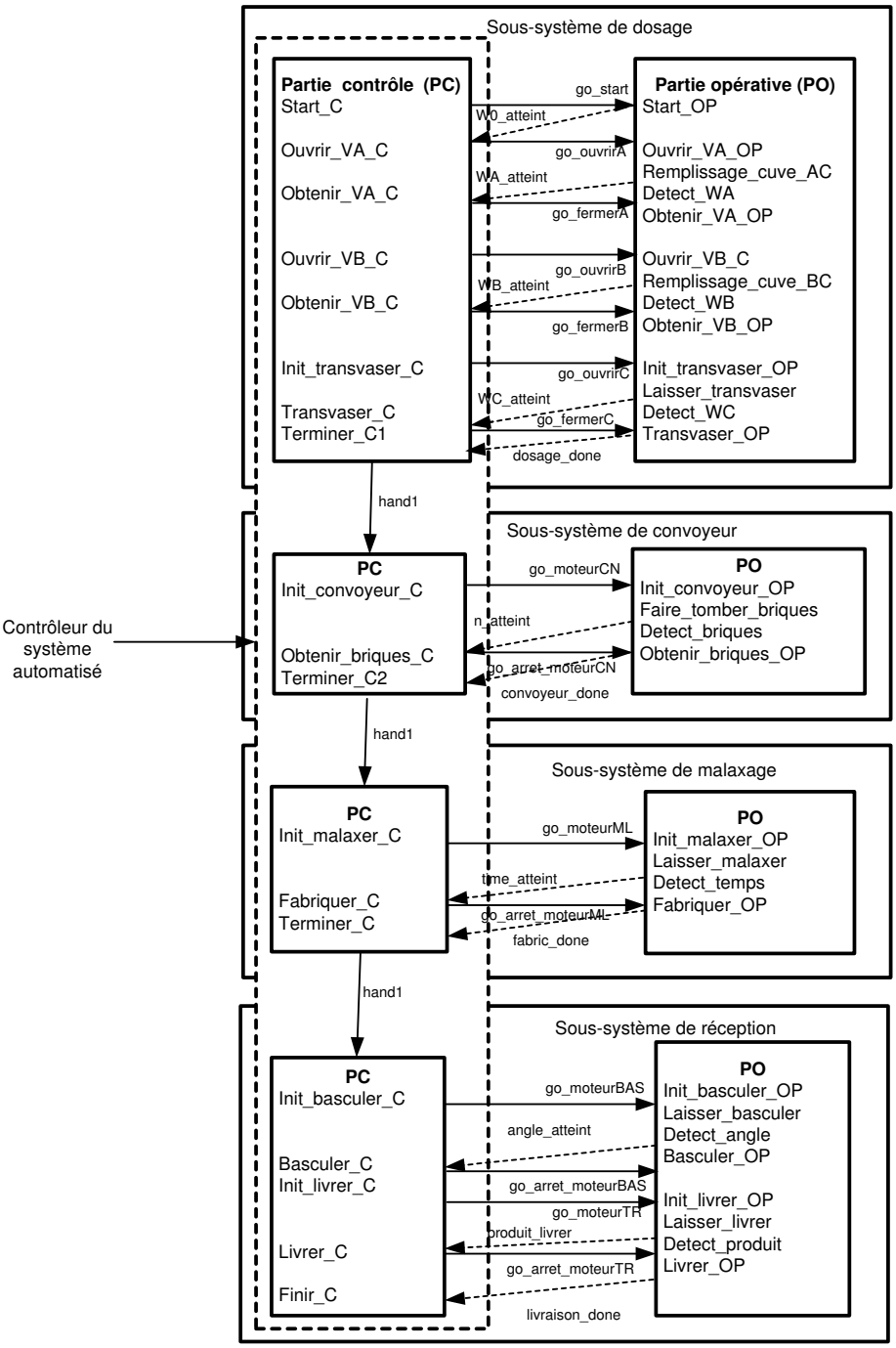


FIG. 8.11 – Les sous-systèmes du troisième raffinement.

Pour tout événement à réalisation différée, nous appliquons le patron de raffinement définit plus haut. Par exemple, l'événement *Obtenir_VA_OP* est raffiné par quatre événements : *Ouvrir_VA_OP, Remplissage_cuveAC_OP, Detect_WA et Obtenir_VA_OP*. Le modèle du contrôlé est présenté à l'annexe F. Ce modèle est un raffinement du modèle précédent. Les statistiques sur les preuves sont données dans le tableau de la figure 8.2.

8.4.6 Quatrième raffinement

Nous raffinons à ce niveau, le sous-système de convoyeur et plus particulièrement le contrôlé de ce sous-système. Nous ajoutons les deux nouveaux événements libres *Avancer_brique_OP* et *Detect_brique_OP*. Le premier modélise le déplacement de la brique une fois le convoyeur est mis en marche et le deuxième modélise la détection de la brique par un capteur ce qui engendre l'incrémentation du nombre des briques. Le modèle du contrôlé du système automatisé figure à l'annexe F.

8.4.7 Statistiques sur les preuves des modèles du contrôleur et contrôlé

Nous donnons dans les tableaux des figures 8.2 et 8.3 les statistiques sur les preuves des modèles du contrôleur et du contrôlé.

Raffinement	*Nombre d'obligations de preuve*	*Nombre de preuves automatiques*	*Nombre de preuves interactives*
0	8	8	0
1er	14	14	0
2ème	1	1	0
3ème	9	8	1
4ème	8	7	1

TAB. 8.2 – Statistiques sur les preuves des modèles du contrôlé.

Raffinement	*Nombre d'obligations de preuve*	*Nombre de preuves automatiques*	*Nombre de preuves interactives*
0	7	7	0
1er	30	21	9
2ème	19	5	4
3ème	254	228	26
4ème	1	1	0

TAB. 8.3 – Statistiques sur les preuves des modèles du contrôleur.

8.5 Conclusion

La contribution la plus intéressante de ce chapitre a été l'introduction d'un patron de raffinement pour modéliser une transition abstraite résultant de l'activation d'un processus continu pendant un temps suffisant pour que l'état final soit atteint. Le raffinement introduit différents événements de gestion du processus ainsi qu'un variant qui permet de prouver qu'au niveau raffiné la transition abstraite finit par se réaliser.

Un tel patron est utilisé dans différents systèmes qui contribuent à la fabrication de produits par un système de production manufacturière : remplissage et vidage de cuves, fourniture de briques de solides acheminées par une bande transporteuse, minuteur pour décompter une durée de mixage, versement du mixer.

Dans une première partie du chapitre, nous nous sommes placés au niveau du système automatisé, dans la seconde partie nous avons fait un double usage de la décomposition en modélisant le système de production comme un ensemble de sous-systèmes s'activant les uns après les autres et en décomposant chaque sous système en deux parties, le contrôleur et le contrôlé. Il est intéressant de noter ici, comment le contrôleur participe au raffinement quand on applique le patron de raffinement relatif à l'action du processus continu.

Pour mieux gérer la preuve dans les processus de conception et de validation, il est important de manipuler des patrons de raffinement fondés sur la preuve. De ce fait, un développement fondé sur la preuve contribue à réduire le coût du développement d'applications.

Conclusion et Perspectives

Dans ce qui suit, nous présentons une synthèse et un bilan du travail effectué puis nous terminons par des perspectives possibles à ce travail.

1 Synthèse et bilan

Le travail présenté dans cette thèse rentre dans le cadre de la spécification et de la vérification des systèmes automatisés. Ces systèmes sont constitués de deux composants : le contrôleur et le contrôlé qui interagissent.

Nous avons choisi le B événementiel pour modéliser les systèmes automatisés. Cette méthode permet de spécifier les systèmes par raffinements successifs d'un modèle initial. On commence par une spécification abstraite du système, on l'enrichit de plus en plus à chaque niveau de raffinement jusqu'à aboutir à l'implémentation. L'intérêt du raffinement est la maîtrise du processus de conception et de vérification de logiciels.

L'objectif initial de la thèse était de voir comment modéliser et vérifier des systèmes hybrides en B, pour ce faire nous sommes partis d'études de cas de systèmes où on trouve à la fois des comportements continus et discrets. Les études de cas étant des systèmes de contrôle-commande, nous avons été intéressés par la structure de ces systèmes, délaissant quelque peu le thème des systèmes hybrides. Ces systèmes, en effet, sont constitués d'un contrôleur (logiciel au comportement discret) qui observe le comportement (continu) du contrôlé en agissant sur un dispositif physique. C'est, bien sûr, le contrôleur qu'il faut développer ; le contrôlé étant une donnée du problème. La correction du contrôleur est établie indirectement par la preuve de correction du système automatisé.

Si dans un premier travail, nous avons développé le système automatisé par raffinements successifs, une contribution importante a consisté ici à proposer une méthode de développement partant de modèles abstraits du contrôleur et du contrôlé, à effectuer sur chacun d'eux un raffinement en les composant à chaque étape pour prouver le système automatisé ainsi produit. La nature des deux composants implique un type particulier de composition. Cette technique présente l'intérêt d'opérer de façon progressive et de récupérer facilement le modèle concret du contrôleur.

Les mécanismes de vérification associés au B événementiel nous ont permis de :

- vérifier la correction de chacun des modèles (contrôleur et contrôlé)
- vérifier la correction globale du système automatisé : (spécification du contrôleur + spécification du contrôlé $\Rightarrow$ spécification des objectifs).

Dans ce travail, nous avons montré comment combiner la composition de sous systèmes automatisés chacun d'eux ayant été obtenu par composition d'un contrôleur et d'un contrôlé. Ceci est applicable à des systèmes complexes.

Les études de cas nous ont montré qu'à un certain niveau d'abstraction, il est possible de tirer profit de l'environnement de B événementiel. Pour contourner l'absence de possibilité d'exprimer et de vérifier des propriétés de vivacité en B événementiel nous avons eu recours à TLA^+ pour proposer une extension à B permettant d'exprimer des propriétés de vivacité sous hypothèse d'équité. Nous avons construit un traducteur de B étendu en TLA^+ pour utiliser les outils de preuve et de model checking disponibles dans cet environnement.

La technique de model checking est également limitée surtout quand la vérification s'effectue sous hypothèses d'équité. Étant essentiellement fondée sur une énumération exhaustive des états atteignables d'un système, elle rencontre rapidement un problème d'explosion combinatoire dès que la taille du système devient trop importante. Pour remédier à cette situation, nous avons proposé l'utilisation des diagrammes de prédicats et son outil associé pour la modélisation et la vérification des propriétés de vivacité sur des systèmes à états infinis.

Dans le domaine des applications temps-réel, la logique temporelle est insuffisante pour prendre compte le caractère métrique du temps. Nous avons consacré moins de temps à développer ce point qui, pourtant, était un des objectifs initiaux. Nous avons, néanmoins, quelques propositions intéressantes. Dans cette voie, nous avons commencé à étudier la modélisation d'un *timeout* à partir de sa programmation en Esterel.

Nous avons aussi abordé le problème de la modélisation des processus continus en B. En effet, pour modéliser des comportements continus correspondant à l'activité des processus physiques, nous avons proposé un patron de conception qui encapsule la modélisation discrète d'un processus continu. Ce patron correspond à un ensemble d'événements modélisant l'écoulement du temps. Nous avons illustré notre méthode sur un cas industriel.

2 Perspectives

Dans le cadre de ces travaux, nous pouvons envisager plusieurs orientations pour ce travail :

- Automatiser le processus de composition en développant un logiciel qui effectue automatiquement la fusion des événements. Ce logiciel pourrait identifier automatiquement les événements à coupler, ce qui présenterait un avantage supplémentaire pour le développeur.
- Étudier la relation entre optimisation et raffinement des modèles B. Sur l'étude de cas du tri de paquets, nous avons commencé à traiter le tri de plusieurs paquets comme une optimisation du système qui n'accepte qu'un paquet à la fois dans le dispositif de tri. Un raffinement permettant de traiter plusieurs paquets à la fois.
- Étendre l'outil Rodin pour prendre en compte les règles de preuve des propriétés de vivacité.
- Tester l'approche de spécification et de vérification sur d'autres systèmes industriels de contrôle commande.
- A l'instar de l'extension temporelle, il pourrait être envisageable d'exprimer des propriétés temporisées en définissant une extension qui exprime le comportement continu des processus et une traduction dans un formalisme qui possède des outils de vérification. UPPALL pourrait être un bon candidat pour inspirer cette extension.

Bibliographie

[Abadi and Lamport, 1988] Abadi and Lamport. The existence of refinement mappings. In *LICS : IEEE Symposium on Logic in Computer Science*, 1988.

[Abadi and Lamport, 1995] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3) :507–534, May 1995.

[Abrial and Hallerstede, 2005] Jean-Raymond Abrial and Stefan Hallerstede. Decomposition, refinement and instantiation of generic models. In *ASM 05*, March 2005.

[Abrial and Hallerstede, 2006] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models : Application to Event-B. *Fundamenta Informaticae*, XXI, 2006.

[Abrial and Mussat, 1998] J-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 83–128, Montpellier, April 1998. Springer Verlag.

[Abrial *et al.*, 1980] J.-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs : An Advanced Course*, pages 343–410. Cambridge University Press, 1980.

[Abrial, 1996a] J-R. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abrial, 1996b] J-R. Abrial. Extending B without changing it (for developing distributed systems). In Henri Habrias, editor, *Proceedings of the 1st Conference on the B method*, pages 169–191, November 1996.

[Abrial, 2000] Jean-Raymond Abrial. Guidelines to formal system studies. MATISSE project, November 2000.

[Abrial, 2003] Jean-Raymond Abrial. B# : Toward a synthesis between Z and B. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 168–177, Turku, Finland, June 2003. Springer.

[Antsaklis and Koutsoukos, 2003] P. J. Antsaklis and X. D. Koutsoukos. Hybrid systems : review and recent progress. In T. Samad and IEEE press G. Balas, Eds, editors, *Chapter in Software-Enabled Control : Information Technologies for Dynamical Systems*, 2003.

[Arnold, 1992a] A. Arnold. An example of use of MEC : Verification of tromp's algorithm. Technical Report IRO-820, Dépt. d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1992.

[Arnold, 1992b] Andre Arnold. *Systèmes de Transitions Finis et Sémantique de Processus Communicants*. Masson, 1992.

[Back and K-Sere, 1989a] R-J. Back and K-Sere. Stepwise refinement of action systems. In *Mathematics of Program Construction.*, pages 115–138, Berlin - Heidelberg - New York, June 1989. Springer.

[Back and K-Sere, 1989b] R-J. Back and K-Sere. Stepwise refinement of action systems. In *Mathematics of Program Construction.*, pages 115–138, Berlin - Heidelberg - New York, June 1989. Springer.

[Back and Kurki-Suonio, 1983] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC*, pages 131–142, 1983.

[Back and v. Wright, 1998] R-J. Back and J v. Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Back and von Wright, 1998] Ralph Johan Back and Joakim von Wright. *Refinement Calculus : A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, Germany, 1998.

[Back, 1978] R. J. Back. *On the correctness of refinement in program development*. Ph.d. thesis, Department of Computer Science, University of Helsinki, Finland, 1978.

[Barradas and Bert, 2002] Hector Ruiz Barradas and Didier Bert. Specification and proof of liveness properties under fairness assumptions in B event systems. 2002.

[Barradas and Bert, 2005] Hector Ruiz Barradas and Didier Bert. Proof obligations for specification and refinement of liveness properties under weak fairness, 2005.

[Behm *et al.*, 1999] P. Behm, P. Benoit, and J. M. Meynadier. Meteor : A Successful Application of B in a Large Project. In *FM 99 — World Conference on Formal Methods in the Development of Computing Systems*, number 1708 in LNCS, pages 369–387. Springer Verlag, 1999.

[Bellegarde *et al.*, 2000a] F. Bellegarde, C. Darlot, J. Julliand, and O. Kouchnarenko. Reformulate dynamic properties during B refinement and forget variants and loop invariants. *Lecture Notes in Computer Science*, 1878 :230–245, 2000.

[Bellegarde *et al.*, 2000b] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Ready-simulation is not ready to express a modular refinement relation. In T. S. E. Maibaum, editor, *FASE*, volume 1783 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2000.

[Bellegarde *et al.*, 2001] Françoise Bellegarde, Christophe Darlot, Jacques Julliand, and Olga Kouchnarenko. Reformulation : A way to combine dynamic properties and B refinement. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 2001.

[Bellegarde *et al.*, 2002] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Synchronised parallel composition of events systems in B. In *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 436–457, Grenoble, France, January 2002. LSR-IMAG.

[Benveniste *et al.*, 1991] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations : The SIGNAL language and its semantics. *Science of Computer and Programming*, 16 :103–149, 1991.

[Berry and Cosserat, 1984] G. Berry and L. Cosserat. The esterel synchronous programming language and its mathematical semantics. In A. W. Roscoe S. D. Brookes and G. Winskel,

editors, *Proceedings of the Seminar on Concurrency*, volume 197 of *LNCS*, pages 389–448. Springer, July 1984.

[Berry and Gonthier,] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152.

[Berry *et al.*,] D. Berry, S. Moisan, and J. Rigault. Esterel : Towards a synchronous and semantically sound high level language for real-time applications. In *Proceedings of the 1983 IEEE Real-Time Systems Symposium*.

[Berry, 2000] G. Berry. *The Foundations of Esterel*. MIT Press, 2000. Editors : G. Plotkin, C. Stirling and M. Tofte.

[Bjorner and Jones, 1978] Dines Bjorner and Cliff B. Jones. The vienna development method : the meta-language. In *The vienna development method : the meta-language*, Lecture Notes in Computer Science. Springer, 1978.

[Blass *et al.*, 2006] A. Blass, Y. Gurevich, D. Rosenzweig, and B. Rossman. Interactive small-step algorithms 2 : abstract state machines and the characterization theorem. To appear in Logical methods in Computer Science. 2008. A preliminary version appeared as Microsoft resarch report MSR-TR-2006-171, 2006.

[Bodeveix *et al.*, 1999] Jean-Paul Bodeveix, Mamoun Filali, and César Munoz. A formalization of the B method in Coq and PVS. In *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*, pages 32–48. Springer-Verlag, 1999.

[Bodeveix *et al.*, 2004] Jean-Paul Bodeveix, Mamoun Filali, and Miloud Rached. Méthodes de spécification de systèmes temps réel en B. 2004.

[Bow,]

[Bozga *et al.*, 1998] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos : A model-checking tool for real-time systems. In *Proc. 10th International Computer Aided Verification Conference*, pages 546–550, 1998.

[Broadfoot and Roscoe, 2000] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and its applications. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, page 322. Springer, 2000.

[Butler, 1996] Michael J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2) :139–173, September 1996.

[Butler, 2000] Michael J. Butler. csp2B : A practical approach to combining CSP and B. *Formal Asp. Comput*, 12(3) :182–198, 2000.

[Cansell *et al.*, 2001a] Dominique Cansell, Dominique Méry, and Stephan Merz. Diagram refinements for the design of reactive systems. *Journal of Universal Computer Science*, 7(2) :159–174, 2001.

[Cansell *et al.*, 2001b] Dominique Cansell, Dominique Méry, and Stephan Merz. Formal analysis of a self-stabilizing algorithm using predicate diagrams. In Martin Wirsing, editor, *Workshop Integrating Diagrammatic and Formal Specification Techniques (GI-/ÖCG-Jahrestagung)*, volume 157/I of *books@ocg.at*, pages 39–45, Vienna, Austria, 2001.

[Caspi *et al.*, 1987] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[Casset, 2002] Ludovic Casset. Development of an embedded verifier for java card byte code using formal methods. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME*, volume 2391 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 2002.

[Chandy and Misra, 1988] K.M. Chandy and J. Misra. *Parallel Program Design : A Foundation*. Addison-Wesley, Reading, MA, 1988. University of Texas–Austin.

[Chombart, 1997] A. Chombart. Commande supervisée des systèmes hybrides. In *Thèse de doctorat. Institut Nationale Polytechnique de Grenoble. France.*, 1997.

[Clarke *et al.*, 1994] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, September 1994.

[ClearSy, 2002a] ClearSy. Atelier b. Technical Note Version 3.6, Aix-en-Provence(F), 2002.

[ClearSy, 2002b] ClearSy. Atelier b. Technical Note Version 3.6, Aix-en-Provence(F), 2002.

[Coq, 2002] The Coq. The coq proof assistant : Reference manual : Version 7.2, February 2002.

[de Groote and Salvati, 2004] Ph. de Groote and S. Salvati. Higher-order matching in the linear lambda-calculus with pairing. Computer Science Logic, 18th International Workshop, Volume 3210 of Lecture Notes in Computer Science, pages 220-234. Springer Verlag, 2004.

[de Groote, 1996] Ph. de Groote. Partially commutative linear logic : sequence calculus and phase semantics. In *Proofs and linguitic categories*. Cooperativa Libraria Universitaria Editrice Bologna. Italy, 1996.

[Dijkstra, 1975] Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM : Communications of the ACM*, 18, 1975.

[Dijkstra, 1976] E-W. Dijkstra. A discipline of programming. Technical report, Prentice-Hall, 1976.

[Fejoz *et al.*, 2005] Loïc Fejoz, Dominique Méry, and Stephan Merz. DIXIT : a graphical toolkit for predicate abstractions, 2005.

[Fowler, 1997] Martin Fowler. *Analysis Patterns : Reusable Objects Models*. Object Technology Series. Addison-Wesley, 1997.

[Hallerstede, 2003] Stefan Hallerstede. Parallel hardware design in B. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors, *ZB'2003 – Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 101–102, Turku, Finland, June 2003. Springer.

[Hammad *et al.*, 2003] Ahmed Hammad, Jacques Julliand, Hassan Mountassir, and D. Okalas Ossami. Expression en B et raffinement des sytèmes réactifs temps réel. In *Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 211–226, IRISA Rennes – France, January 2003. AFADL2003, IRISA, IRISA.

[Hammer *et al.*, 2005] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking, 2005.

[Harel and others, 1988] D. Harel et al. STATEMATE : a working environment for the development of complex reactive systems. In *Proc 10th IEEE Conf on Software Engineering*, 1988.

[Harel *et al.*, 1987] Harel, Pnueli, Schmidt, and Sherman. On the formal semantics of statecharts. In *LICS : IEEE Symposium on Logic in Computer Science*, 1987.

[Henzinger *et al.*, 1997] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH : A model checker for hybrid systems. *Lecture Notes in Computer Science*, 1254 :460–463, 1997.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall, 1985.

[Holzmann, 1997] Gerald J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, May 1997.

[Holzmann, 2003] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

[Hyt, 1997] Hytech : The hybrid technology tool, 1997.

[i-Logix, 1987] i-Logix. The languages of STATEMATE. Technical report, i-Logix, Burlington, Mass., 1987.

[Jones, l981] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Prgr.Res.Grp. 25, Oxford Univ., Comp. Lab., UK, June l981.

[Julliand *et al.*, 1999] Jacques Julliand, Pierre-Alain Masson, and Hassan Mountassir. Modular verification of dynamic properties for reactive systems. In *IFM*, pages 89–108, 1999.

[K-M. Chandy and J. Misra, 1989] K-M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, May 1989.

[Kro, 1998] The tool kronos, 1998.

[Kupferman and Vardi, 2000] Orna Kupferman and Moshe Y. Vardi. An automata-theortetic approach to modular model checking. *ACM Trans. Program. Lang. Syst*, 22(1) :87–128, 2000.

[Lamport and Yu, 2003] Leslie Lamport and Yuan Yu. Tlc-the tla+ model checker, 2003.

[Lamport, 1991] L. Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Systems Research Centre, December 1991.

[Lamport, 1994a] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, 1994.

[Lamport, 1994b] Leslie Lamport. Introduction to TLA. Technical Note 1994-001, Digital Systems Research Center, Palo Alto, CA, December 1994.

[Lamport, 1996] Leslie Lamport. Refinement in state-based formalisms. Technical Report SRC-TN-1996-001, Hewlett Packard Laboratories, December 23 1996.

[Lamport, 2002] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lano and Haughton, 1996] K. Lano and H. Haughton. *Specification in B : An Introduction Using the B Toolkit*. Imperial College Press, London, 1996.

[Lano *et al.*, 1996a] K. Lano, J. Bicarregui, and A. Sanchez. Using B to design and verify controllers for chemical processing. In Henri Habrias, editor, *Proceedingsof 1st Conference on the B method*, pages 237–270, November 1996.

[Lano *et al.*, 1996b] K. Lano, J. Fiadeiro, and J. Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[Lano *et al.*, 1996c] Kevin Lano, J. Fiadeiro, and Jeremy Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[Lano, 1996] K. Lano. *The B Language and Method : A guide to Practical Formal Development*. Springer Verlag London., 1996.

[Larsen *et al.*, 1997a] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, October 1997.

[Larsen *et al.*, 1997b] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL : Status & developments. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer, 1997.

[Leduc and Germeau, 2000] G. Leduc and F. Germeau. Verification of security protocols using lotos - method and application. In *Computer communications*, pages 1089–1103. Special issue on formal description techniques in practice, vol 23, 2000.

[Leuschel and Butler, 2003a] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[Leuschel and Butler, 2003b] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[Lewis, 2001] R. Lewis. Modelling control systems using iec 61499. In *Institution Of Engineering and Technology (UK), ISBN 0852967969*, 2001.

[Méry, 1986] Dominique Méry. A proof system to derive evantually properties under justice hypothesis. In *MFCS*, pages 536–544, 1986.

[Méry, 1987] Dominique Méry. Méthode axiomatique pour les propriétés de fatalité des programmes parallèles. *ITA*, 21(3) :287–322, 1987.

[Merz, 1999] S. Merz. An encoding of TLA in isabelle, January 25 1999.

[Merz, 2001] Stephan Merz. Model checking : A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[Merz, 2002] Stephan Merz. Model checking techniques for the analysis of reactive systems. *Synthese*, 133 :173–201, 2002. Special Issue *Foundations of the Formal Sciences*, B. Löwe and F. Rudolph (eds.).

[Merz, 2003] Stephan Merz. On the logic of tla$^+$. *Computers and Informatics*, 22 :351–379, 2003.

[Metayer *et al.*, 2005] Christophe Metayer, Jean-Raymond Abrial, and Laurent Voisin. Event-B language. RODIN Project Deliverable D7, May 2005.

[Michael Lemmon and Antsaklis, 2006] Peter Szymanski Michael Lemmon, Christopher Bett and Panos Antsaklis. Constructing hybrid control systems from robust linear control agents. In *Hybrid Systems II, Springer Berlin / Heidelberg*, 2006.

[Morgan, 1994] Carroll Morgan. *Programming from Specifications : Second Edition.* Prentice Hall International, Hempstead, UK, 1994.

[Morris, 1987] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3) :287–306, December 1987.

[Mosbahi and Jaray, 2004a] O. Mosbahi and J. Jaray. Représentation du temps en b événementiel pour la modélisation des systèmes temps réel. Rapport interne, LORIA, 2004.

[Mosbahi and Jaray, 2004b] O. Mosbahi and J. Jaray. Une démarche formelle de développement de systèmes de contrôle-commande. Rapport interne, LORIA, 2004.

[Mosbahi and Jaray, 2007] O. Mosbahi and J. Jaray. Specification and proof of liveness properties in b event systems. In *ICSOFT. 2th International Conference on Software and Data Technology, Barcelona*, 2007.

[Mosbahi and Jemni, 2006] O. Mosbahi and L. Jemni. Utilisation conjointe de la méthode b événementiel et de la logique temporelle tla+ pour la modélisation et la vérification des systèmes réactifs. In *MOSIM, Modélisation, Optimisation et Simulation des Systèmes, Rabat*, 2006.

[Mosbahi *et al.*, 2006a] O. Mosbahi, J. Jaray, and L. Jemni. A formal development approach of control systems using the event based b approch, case study : A parcel sorting device. In *ACS/IEEE, International Conference on Computer Systems and Applications, Dubai*, 2006.

[Mosbahi *et al.*, 2006b] O. Mosbahi, L. Jemni, and J. Jaray. A formal development approach of automated systems using the temporal logic of actions tla+. In *MOSIM, Modélisation, Optimisation et Simulation des Systèmes, Rabat*, 2006.

[Mosbahi *et al.*, 2007a] O. Mosbahi, J. Jaray, and S. Benahmed. Spécification et vérification des propriétés de vivacité en b événementiel. In *MSR. Modélisation des systèmes réactifs, Lyon*, 2007.

[Mosbahi *et al.*, 2007b] O. Mosbahi, L. Jemni, and J. Jaray. A formal approach of the development of automated systems. In *ICSOFT. 2th International Conference on Software and Data Technology, Barcelona*, 2007.

[Mountassir *et al.*, 2000] H. Mountassir, F. Bellegarde, Jacques Julliand, and P.-A. Masson. Coopération entre preuve et model-checking pour vérifier modulairement des propriétés LTL. In *AFADL'2000*, pages 127–141. AFADL2000, LSR/IMAG, LSR/IMAG, January 2000.

[Murata *et al.*, 2006] T. Murata, J. Yim, H. Yin, and O. Wolfson. Petri-net model and minimum cycle time for updating moving objects database. pages 211–217. International Journal of Computer Systems Science and Engineering, Vol.21, No.3, 2006.

[Nipkow, 2000] Tobias Nipkow. Isabelle HOL - the tutorial, May 10 2000.

[Nise, 2003] N. S. Nise. *Control systems engineering.* Wiley ; 4 Har/Cdr edition, 2003.

[Ostroff, 1989] J. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press Ltd., 1989.

[Owre *et al.*, 1992] S. Owre, J. M. Rushby, and N. Shankar. PVS : A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume June of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[Pnueli, 1988] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems. In *REX School-Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Noordwijkerhout, Netherlands, 1988.

[Pnueli, 1992] Pnueli. System specification and refinement in temporal logic. *FSTTCS : Foundations of Software Technology and Theoretical Computer Science*, 12, 1992.

[Potet, 2002] Marie-Laure Potet. *Spécifications et développements formels : Etude des aspects compositionnels dans la méthode B.* Thèse d'habilitation, LSR-IMAG, 2002.

[Pouzancre and Pitzalis, 2003] Guilhem Pouzancre and Jean-Philippe Pitzalis. Modélisation en B événementiel des fonctions mécaniques, électriques et informatiques d'un véhicule. *Technique et Science Informatiques*, 22(1) :119–128, 2003.

[Pouzancre, 2003] Guilhem Pouzancre. How to diagnose a modern car with a formal B model ? In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors, *ZB'2003 – Formal Specification and Development in Z and B,International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 98–100, Turku, Finland, June 2003. Springer.

[PVS, 1992] Pvs specification and verification system, 1992.

[Ramadge and Wonham, 1987] P. J. G. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control Optimization*, 25(1) :206–230, 1987.

[Ramadge and Wonham, 1989] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. of the IEEE*, 77(1) :81–98, January 1989.

[Rodin Partners, 2006] Rodin Partners. Rodin : Rigorous open development environment for complex systems. 2006.

[Roscoe, 1998] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, New York, 1998. Oxford.

[Rushby, 2000] J. Rushby. Theorem proving for verification. In *Proceedings of Summer School of Modelling and Verification of Parallel Processes : MOVEP 2000*, pages 39–57, June 2000.

[Schneider and Treharne, 2002] Steve Schneider and Helen Treharne. Communicating B machines. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 416–435. Springer, 2002.

[Shoenfield, 1978] J. R. Shoenfield. Axoims of set theory. In *hnbk*, chapter B.1, pages 321–344. 1978.

[Spivey, 1992] J. Spivey. *The Z Notation. A Reference Manual*. Prentice-Hall, 2 edition, 1992.

[Treharne *et al.*, 2003] Helen Treharne, Steve Schneider, and Marchia Bramble. Composing specifications using communication. In Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors, *ZB'2003 – Formal Specification and Development in Z and B,International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 58–78, Turku, Finland, June 2003. Springer.

[Welch, 2002] Peter Welch. Concurrent systems, CSP, and FDR, June 01 2002.

[Wonham, 2003] W.M. Wonham. Supervisory control theory : Models and methods. Workshop on Discrete Event Systems Control, 24th International Conference on Application Theory of Petri Nets (ATPN 2003), The Netherlands, pp.1-14, 2003.

[Worm, 1999] Torben Worm. Using metapatterns with SDL. In *SDL Forum*, pages 355–372, 1999.

[Zhang and Mackworth, 1995] Y. Zhang and A. K. Mackworth. Synthesis of hybrid constraint-based controllers. *Lecture Notes in Computer Science*, 999 :552 – 567, 1995.

[Zwiers, 1989] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *LNCS*. Springer-Verlag, 1989.

Annexe A
Prototype B2TLA$^+$

Nous présentons dans cet annexe le prototype B2TLA$^+$ que nous avons implanté afin de transformer un modèle B temporel vers un module TLA$^+$.

1 Prototype

L'interface du prototype présente quatre boutons :

Fig. 1 – Interface du prototype B2TLA$^+$

- Bouton Ouvrir : ce bouton est utilisé pour ouvrir un modèle B qu'on souhaite étendre pour prendre en compte des propriétés de vivacité.
- Bouton Convertir : ce bouton est utilisé pour transformer tout modèle B temporel vers un module TLA$^+$ afin de vérifier les propriétés de vivacité.
- Bouton Ajouter : ce bouton est utilisé pour ajouter des nouvelles clauses : Clauses FAIRNESS et EVENTUALITY et obtenir un modèle B temporel. La première clause permet d'ajouter des conditions d'équité sur l'exécution de certains événements. La deuxième clause ajoute des propriétés de fatalité.
- Bouton Quitter : ce bouton sert à quitter l'application.

2 Extension de la grammaire de B

Nous présentons dans ce qui suit l'extension de la grammaire de B par de nouvelles règles pour prendre en compte les propriétés d'équité et de fatalité.

clause_machine_abstraite : : = ...
| clause_eventuality
| clause_Fairness

clause_eventuality : : = "EVENTUALITY" predicat_eventuality ;

clause_fairness : : = "FAIRNESS" predicat_fairness ;

predicat_fairness : : = "WF" "(" Entete_operation ")"
| "SF" "(" Entete_operation ")"

predicat_eventuality : : = predicat_vivacite1 "Leadsto" predicat_vivacite1
| predicat "=>" predicat_vivacite2
|"[]" "<>" predicat

predicat_vivacite1 : : = predicat
| predicat_fairness

predicat_vivacite2 : : = "<>" predicat

3 Symboles en B et TLA$^+$

B événementiel	*Langage TLA$^+$*	*Mathématique*
&	$\wedge$	$\wedge$
or	$\vee$	$\vee$
=>	=>	$\Rightarrow$
:	$\in$	$\in$
/ :	$\notin$	$\notin$
/=	/=	$\neq$
<	<	<
>	>	>
<=	<=	$\leq$
>	>	$\geq$
!	\A	$\forall$
#	\E	$\exists$
*	$\times$	$\times$
« :	$\subset$	$\subset$
< :	$\subseteq$	$\subseteq$
pred	$\prec$	$\prec$
succ	$\succ$	$\succ$
|->	|->	$\longmapsto$
*	$\times$	$\times$
+	+	+
–>	–>	$\rightarrow$
[]	[]	$\Box$
<>	<>	$\Diamond$
~ >	~ >	$\rightsquigarrow$
Inter	$\cap$	$\cap$
Union	$\cup$	$\cup$
NATURAL	NATURAL	$\mathbb{N}$
NATURAL1	NATURAL1	$\mathbb{N}_1$

Tab. 1 – Symboles en B et TLA$^+$.

Annexe B
Les modèles B du système de tri des paquets postaux

Nous présentons dans cet annexe les modèles B correspondants à la modélisation du système de tri de paquets postaux au niveau abstrait ainsi que les trois raffinements.

1 Modèle abstrait

MODEL **Parcel_routing**
SETS
EPARCELS; *NODES*
CONSTANTS
PARCELS, BASKETS, adr
PROPERTIES
$PARCELS \subset EPARCELS \wedge BASKETS \subset NODES \wedge PARCELS \neq \{\} \wedge$
$BASKETS \neq \{\} \wedge adr \in PARCELS \rightarrow BASKETS$
VARIABLES
stored
INVARIANT
$stored \in PARCELS \nrightarrow BASKETS \wedge stored \subseteq adr$
INITIALISATION
$stored := \{\}$
EVENTS
Cross_sorting = **ANY** *p, b* **WHERE**
$p \in PARCELS \wedge$
$b \in BASKETS \wedge$
$p \notin dom(stored) \wedge$
$p \mapsto b \in adr$
THEN
$stored := stored \cup \{p \mapsto b\}$
END
END

2 Modèle du premier raffinement

REFINEMENT **Parcel_routing1**
REFINES **Parcel_routing**
VARIABLES
stored, *channel*, *next*, *current*, *sorted*, *ready_to_sort*, *adr_next*
INVARIANT
$next \in EPARCELS \wedge current \in$ EPARCELS $\wedge\ channel \in BASKETS \wedge$
$ready\_to\_sort \in BOOL \wedge sorted \subseteq PARCELS \wedge adr\_next \in BASKETS \wedge$
$(ready\_to\_sort = TRUE \Rightarrow channel = adr\_next) \wedge$
$(current \in PARCELS \Rightarrow channel = adr(current)) \wedge$
$(next \in PARCELS \Rightarrow adr\_next = adr(next)) \wedge$
$\forall p.(p \in PARCELS \wedge$ p $\in dom(stored) \Rightarrow stored(p) = adr(p))$
DEFINITIONS
$TO\_SORT == PARCELS$ - $sorted$;
$UNDEF == EPARCELS$ - $PARCELS$
INITIALISATION
$stored := \{\} || channel :\in BASKETS || current :\in UNDEF || next :\in UNDEF ||$
$sorted := \{\} || ready\_to\_sort := FALSE || adr\_next :\in BASKETS$
EVENTS
Select_parcel = **ANY** p **Where**
$p \in TO\_SORT \wedge current \in UNDEF \wedge$
$next \in UNDEF$ **THEN**
$adr\_next := adr(p) || next := p$
END ;
Set_channel = **SELECT**
$ready\_to\_sort = FALSE \wedge next \in PARCELS \wedge current \in UNDEF$
THEN
$channel := adr\_next || ready\_to\_sort := TRUE$
END ;
Release = **SELECT**
$next \in PARCELS \wedge current \in UNDEF \wedge ready\_to\_sort = TRUE$
THEN
$current := next || next :\in UNDEF || ready\_to\_sort := FALSE$
END ;
Cross_sorting = **SELECT**
$current \in PARCELS$
THEN
$stored(current) := channel \ || \ sorted := sorted \ \cup \{ current\} \ ||$
$current :\in UNDEF$
END
END

3 Modèle du deuxième raffinement

REFINEMENT Parcel_routing2
REFINES Parcel_routing1

CONSTANTS

rac, access, switches, right_n, left_n, depth, T

PROPERIES

rac $\in$ *NODES* $\wedge$ *rac* $\notin$ *BASKETS* $\wedge$ *switches* = *NODES - BASKETS* $\wedge$
BASKETS $\subset$ *NODES* $\wedge$ *T* $\in$ *NATURAL1* $\wedge$
depth $\in$ NODES $\rightarrow$ 0..T $\wedge$ depth(rac) = 0$\wedge$
access $\in$ *switches* $\rightarrow$ *POW1(BASKETS)* $\wedge$ *access(rac)* = *BASKETS* $\wedge$
right_n $\in$ *switches* $\rightarrow$ *NODES* $\wedge$ *left_n* $\in$ *switches* $\rightarrow$ *NODES* $\wedge$
$\forall$s.(s $\subseteq$ NODES $\wedge$ rac $\in$ s $\wedge$ (left_n $\cup$ right_n)[s] $\subseteq$ s $\Rightarrow$ NODES $\subseteq$ s)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ right_n(n) $\neq$ left_n(n))$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ right_n(n) $\neq$ n)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ left_n(n) $\neq$ n)$\wedge$
$\forall$n.(n $\in$ switches $\wedge$ right_n(n) $\in$ switches $\wedge$ left_n(n) $\in$ switches
$\Rightarrow$ access(right_n(n)) $\cap$ access(left_n(n)) = $\varnothing\wedge$
access(n) = access(right_n(n)) $\cup$ access(left_n(n))) $\wedge$
$\forall$n.(n $\in$ BASKETS $\Rightarrow$ depth(n) = T)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ depth(right_n(n)) = depth(n) + 1)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ depth(left_n(n)) = depth(n) + 1) $\wedge$
iterate(right_n, T)(rac) $\in$ *BASKETS* $\wedge$
iterate(left_n, T)(rac) $\in$ *BASKETS*

VARIABLES

stored, *channel*, *current*, *next*, *adr_next, sorted, ready_to_sort, exit, nd, lng*

INVARIANT

exit $\in$ *switches* $\nrightarrow$ *NODES* $\wedge$ *nd* $\in$ *NODES* $\wedge$ *lng* $\in$ *0..T* $\wedge$
/ Invariants de collage */*
(next $\in$ *PARCELS* $\wedge$ *current* $\notin$ *PARCELS* $\wedge$ *nd* $\neq$ *rac* $\wedge$ *nd* $\in$ *switches* $\wedge$
(right_n(nd) $\in$ *BASKETS* $\vee$ *left_n(nd)* $\in$ *BASKETS) $\Rightarrow$ channel = exit (nd)) $\wedge$*
(current $\in$ *PARCELS* $\wedge$ *nd* $\neq$ *rac* $\wedge$ *current* $\notin$ *dom(stored)* $\wedge$ *exit(nd)* $\in$ *BASKETS* $\Rightarrow$ *channel = exit(nd))*
(exit $\neq \varnothing \Rightarrow$ *iterate(exit,T)(rac)=channel)*

DEFINITIONS

TO_SORT == PARCELS - sorted;
UNDEF == EPARCELS - PARCELS

INITIALIZATION

stored := $\varnothing$|| *next* : $\in$ *UNDEF* || *current* : $\in$ *UNDEF* ||
adr_next : $\in$ *BASKETS* || *sorted* := $\varnothing$|| *ready_to_sort* := *FALSE* ||
nd := *rac* || *exit* := $\varnothing$|| *lng* :=*0*

EVENTS

Select_parcel = **ANY** *p* **Where**
p $\in$ *TO_SORT* $\wedge$ *next* $\in$ *UNDEF* $\wedge$
current $\in$ *UNDEF* $\wedge$ *ready_to_sort* = *FALSE* $\wedge$ *exit* = $\varnothing$
THEN
adr_next := *adr(p)* || *next* := *p* || *nd* := *rac* || *lng* := *0*
END ;

Set_switch $\triangleq$ **ANY** *node* **WHERE**
node $\in$ {right_n(nd), left_n(nd)}$\wedge$ *nd* $\in$ *switches* $\wedge$
node $\notin$ *BASKETS* $\wedge$ *next* $\in$ *PARCELS* $\wedge$ *lng* < *T-1* $\wedge$
current $\in$ *UNDEF* $\wedge$ *adr_next* $\in$ *access(node)* $\wedge$ *ready_to_sort* = *FALSE*

THEN
exit(nd) := node || nd := node || lng := lng+1
END ;
Set_channel ≙ **ANY** *node* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧ *nd* ∈ *switches* ∧
node ∈ *BASKETS* ∧ *adr_next* = *node* ∧ *next* ∈ *PARCELS* ∧
current ∈ *UNDEF* ∧ *ready_to_sort* = *FALSE* ∧ *lng* = *T-1*
THEN
exit(nd) := adr_next || nd := adr_next ||
ready_to_sort := TRUE || lng := lng+1
END ;
Release ≙ **SELECT**
next ∈ *PARCELS* ∧ *current* ∈ *UNDEF* ∧
ready_to_sort = *TRUE* ∧ *nd* ∈ *BASKETS*
THEN
current := next || next : ∈ *UNDEF || nd := rac ||*
ready_to_sort := FALSE || lng := 0
END ;
Cross_node ≙ **SELECT**
current ∈ *PARCELS* ∧ *exit(nd)* ∉ *BASKETS* ∧ *nd* ∈ *switches* ∧ *lng* < *T-1*
THEN
nd := exit(nd) || lng := lng+1
END ;
Cross_sorting ≙ **SELECT**
current ∈ *PARCELS* ∧ *exit(nd)* ∈ *BASKETS* ∧ *lng* = *T-1*
THEN
stored(current) := nd || sorted := sorted ∪ { *current*} ||
current : ∈ *UNDEF || lng := lng+1*
END
END

4 Modèle du troisième raffinement

REFINEMENT **Parcel_routing3**
REFINES **Parcel_routing2**
SETS *OUT* = {L, R, Indef}
VARIABLES
ready_to_sort, stored, current, next, sorted, adr_next,
exit, nd, lng, in, out_l , out_r , gate
INVARIANT
in ∈ *NODES* → *BOOL* ∧ *out_r* ∈ *switches* → *BOOL* ∧
out_l ∈ *switches* → *BOOL* ∧ *gate* ∈ *switches* → *OUT* ∧ *dom(in)* = *NODES* ∧
∀ *i* .(*i* ∈ *switches* ⇒ ¬ (*out_r(i)*=*TRUE* ∧ *out_l(i)*=*TRUE*))
INITIALIZATION
stored := ∅|| *ready_to_sort := FALSE || next* : ∈ *UNDEF ||*
current : ∈ *UNDEF || adr_next* : ∈ *BASKETS || sorted* := ∅||

lng := *0* || *nd* := *rac* || *exit* := ∅||
in := *NODES* * {FALSE}|| *out_r* := *switches* * {FALSE}||
out_l := *switches* * {FALSE}|| *gate* := *NODES* * {Indef}

EVENTS

Select_parcel ≙ **ANY** *p* **Where**
p ∈ *TO_SORT* ∧ *next* ∈ *UNDEF* ∧ *current* ∈ *UNDEF* ∧
ready_to_sort = *FALSE*
THEN
adr_next := *adr(p)* || *next* := *p* || *in(nd)* := *TRUE* || *nd* := *rac* || *lng* := *0*
END ;

Set_gate_right ≙ **SELECT**
next ∈ *PARCELS* ∧
((right_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(right_n(nd))) or*
(nd ∈ *switches* ∧ *right_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := *R*
END ;

Set_gate_left ≙ **SELECT**
next ∈ *PARCELS* ∧
((left_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(left_n(nd))) or*
(nd ∈ *switches* ∧ *left_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*
THEN
gate(nd) := *L*
END ;

Set_switch ≙ **ANY** *node, gate_nd* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧
nd ∈ *switches* ∧ *gate_nd* ∈ {R, L} ∧ *node* ∉ *BASKETS* ∧ *next* ∈ *PARCELS* ∧
current ∈ *UNDEF* ∧ *lng* < *T-1* ∧ *adr_next* ∈ *access(node)* ∧
ready_to_sort = *FALSE* ∧ *in(nd)* = *TRUE* ∧ *gate(nd)* = *gate_nd*
THEN
exit(nd) := *node* || *nd* := *node* || *lng* := *lng+1* || *in(node)* := *TRUE*
END ;

Set_channel ≙ **ANY** *node* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧ *nd* ∈ *switches* ∧
node ∈ *BASKETS* ∧ *adr_next* = *node* ∧ *next* ∈ *PARCELS* ∧ *lng* = *T-1* ∧
current ∈ *UNDEF* ∧ *ready_to_sort* = *FALSE* ∧ *gate(nd)* ≠ *Indef*
THEN
exit(nd) := *adr_next* || *nd* := *node* || *lng* := *lng+1* ||
ready_to_sort := *TRUE* || *in(node)* := *TRUE*
END ;

Release ≙ **SELECT**
next ∈ *PARCELS* ∧ *current* ∈ *UNDEF* ∧
ready_to_sort = *TRUE* ∧ *nd* ∈ *BASKETS*
THEN
current := *next* || *next* : ∈ *UNDEF* || *nd* := *rac* || *lng* := *0* //
ready_to_sort := *FALSE* || *in(nd)* := *TRUE*
END ;

Cross_right ≙ **SELECT**

current $\in$ *PARCELS* $\wedge$ *nd* $\in$ *switches* $\wedge$ *in(nd)* = *TRUE* $\wedge$ *gate(nd)*= *R*
THEN
out_l(nd) := *FALSE* || *out_r(nd)* := *TRUE* || *in(nd)* := *FALSE*
END ;

Cross_left $\triangleq$ **SELECT**
current $\in$ *PARCELS* $\wedge$ *nd* $\in$ *switches* $\wedge$ *in(nd)* = *TRUE* $\wedge$ *gate(nd)*= *L*
THEN
out_l(nd) := *TRUE* || *out_r(nd)* := *FALSE* || *in(nd)* := *FALSE*
END ;

Cross_node $\triangleq$ **SELECT**
current $\in$ *PARCELS* $\wedge$ *exit(nd)* $\notin$ *BASKETS* $\wedge$ *nd* $\in$ *switches* $\wedge$
(out_r(nd) = *TRUE* $\vee$ *out_l(nd)* = *TRUE)* $\wedge$ *lng* < *T-1*
THEN
nd := *exit(nd)* || *in(nd)* := *TRUE* || *lng* := *lng+1*
END ;

Cross_sorting $\triangleq$ **SELECT**
current $\in$ *PARCELS* $\wedge$ *exit(nd)* $\in$ *BASKETS* $\wedge$ *lng* = *T-1*
THEN
stored(current) := *exit(nd)* || *sorted* := *sorted* $\cup$ { *current*} ||
current : $\in$ *UNDEF* || *in(nd)* := *TRUE* || *lng* := *lng+1*
END

END

Annexe C
Les modules TLA$^+$ du système de tri des paquets postaux

Nous présentons dans cet annexe, les modules TLA$^+$ au niveau abstrait ainsi qu'aux niveaux des trois raffinements correspondants au système de tri de paquets postaux.

1 Model checker TLC

L'entrée à TLC correspond à un module TLA$^+$, un fichier de configuration et un fichier des instances finies. TLC vérifie si la spécification à la forme *INIT* $\wedge[\text{Next}]_{\text{vars}}\wedge$ *Temporal*. Le fichier de configuration indique à TLC le nom de la spécification et les propriétés à vérifier. Par exemple, le fichier de configuration *MCParcel_routing* du système de tri de paquets contient la déclaration SPECIFICATION *Spec* indiquant à TLC que *Spec* est la spécification. Les propriétés à vérifier sont spécifiées par la clause PROPERTIES. La clause INVARIANT spécifie un prédicat d'état (P). Pour prouver des propriétés d'invariance avec la clause PROPERTIES, la propriété doit avoir la forme P.

TLC génère les comportements qui satisfont la spécification. Pour ce faire, nous avons besoin d'un modèle de la spécification (fichier des instances). Dans ce modèle, nous assignons des valeurs aux constantes.

Il y a deux méthodes d'utilisation de TLC. La méthode par défaut est le *model checking* où il essaye de trouver tous les états accessibles. La deuxième méthode est le mode *simulation*, dans lequel il génère aléatoirement les comportements sans essayer de vérifier tous les états accessibles.

2 Module abstrait

MODULE Parcel_routing

EXTENDS Naturals

CONSTANTS EPARCELS, PARCELS, NODES, BASKETS, noBaskets, adr

VARIABLES stored

ASSUME adr $\in$ [PARCELS $\rightarrow$ BASKETS] $\wedge$ PARCELS $\neq$ {} $\wedge$ BASKETS $\neq$ {}
$\wedge$ PARCELS $\subseteq$ EPARCELS $\wedge$ PARCELS $\neq$ EPARCELS $\wedge$ noBaskets $\notin$ BASKETS
$\wedge$ BASKETS $\subseteq$ NODES $\wedge$ BASKETS $\neq$ NODES

TypeInvariant $\triangleq$ $\wedge$ stored $\in$ [PARCELS $\rightarrow$ BASKETS $\cup$ {noBaskets}]

```
Init ≜ ∧ stored = [p ∈ PARCELS ↦ noBaskets]
Cross_sorting ≜ ∧ ∃ << p, b >> ∈ PARCELS × BASKETS : p ∉ DOMAIN stored
                ∧ adr[p] = b
                ∧ stored′ = [stored EXCEPT ![p] = b]
Next ≜ ∨ Cross_sorting
```

```
trvars ≜ ⟨stored⟩
Fairness ≜ WF_trvars(cross_sorting)
Spec ≜ Init ∧ □[Next]_trvars ∧ Fairness
Invariant ≜ ∧ ∀ p ∈ PARCELS : p ∈ DOMAIN stored ⇒ stored[p] = adr[p]
```

```
NoParcelStored == stored = [p ∈ PARCELS ↦ noBaskets]
THEOREM Spec ⟹ □TypeInvariant
THEOREM Spec ⟹ □Invariant
```

Module de configuration :

```
MODULE MCParcel_routing
SPECIFICATION Spec
INVARIANTS TypeInvariant Invariant
```

Module des instances finies :

Ce module contient les instances de toutes les constantes déclarées dans le module Parcel_routing.

```
MODULE MCParcel_routing
EXTENDS Naturals
VARIABLES stored
PARCELS == 1..5
EPARCELS == 0..5
noBaskets == 20
NODES == 0..14
BASKETS == 7..14
adr == [p ∈ PARCELS ↦
        CASE p=1 → 7
        [ ] p=2 → 10
        [ ] p=3 → 12
        [ ] p=4 → 9
        [ ] p=5 → 14]
INSTANCE Parcel_routing
```

3 Module du premier raffinement

```
MODULE Parcel_routing1
EXTENDS Naturals, Parcel_routing
VARIABLES channel, next, current, sorted, adr_next, ready_to_sort
```

```
TypeInvariant1 ≜ ∧ channel ∈ BASKETS ∪ {noBaskets}
                 ∧ adr_next ∈ BASKETS ∪ {noBaskets}
                 ∧ next ∈ EPARCELS
                 ∧ current ∈ EPARCELS
                 ∧ stored ∈ [PARCELS → BASKETS ∪ {noBaskets}]
                 ∧ sorted ⊆ PARCELS
```

$\land$ ready_to_sort $\in$ BOOLEAN

TO_SORT $\triangleq$ PARCELS $\setminus$ sorted

UNDEF $\triangleq$ EPARCELS $\setminus$ PARCELS

Init1 $\triangleq$ $\land$ channel = noBaskets
$\land$ adr_next = noBaskets
$\land$ next $\in$ UNDEF
$\land$ current $\in$ UNDEF
$\land$ stored = [p $\in$ PARCELS $\mapsto$ noBaskets]
$\land$ sorted = {}
$\land$ ready_to_sort = FALSE

Select_parcel $\triangleq$
$\land$ next $\in$ UNDEF
$\land$ current $\in$ UNDEF
$\land \exists$ p $\in$ TO_SORT : $\land$ next$'$ = p $\land$ adr_next$'$ = adr[p]
$\land$ UNCHANGED $\langle$channel, stored, sorted, current, ready_to_sort$\rangle$

Set_channel $\triangleq$ $\land$ next $\in$ PARCELS
$\land$ current $\in$ UNDEF
$\land$ ready_to_sort = FALSE
$\land$ channel$'$ = adr_next
$\land$ ready_to_sort$'$ = TRUE
$\land$ UNCHANGED $\langle$stored, sorted, current, next, adr_next$\rangle$

Release $\triangleq$ $\land$ ready_to_sort = TRUE
$\land$ current $\in$ UNDEF
$\land$ next $\in$ PARCELS
$\land$ current$'$ = next
$\land$ next$'$ $\in$ UNDEF
$\land$ ready_to_sort$'$ = FALSE
$\land$ UNCHANGED $\langle$channel, stored, sorted, adr_next$\rangle$

Cross_sorting1 $\triangleq$ $\land$ current $\in$ PARCELS
$\land$ stored$'$ = [stored EXCEPT ![current] = channel]
$\land$ sorted$'$ = sorted $\cup$ {current}
$\land$ current$'$ $\in$ UNDEF
$\land$ UNCHANGED $\langle$channel, ready_to_sort, next, adr_next$\rangle$

Next1 $\triangleq$ $\lor$ Select_parcel
$\lor$ Set_channel
$\lor$ Release
$\lor$ Cross_sorting

---

tr1vars $\triangleq$ $\langle$channel, next, current, stored, sorted, ready_to_sort, adr_next$\rangle$

Tr1Fairness $\triangleq$ $\mathrm{WF}_{\mathrm{trvars}}$(Select_parcel) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Set_channel) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Release) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Cross_sorting)

Spec1 $\triangleq$ Init1 $\land \Box[\mathrm{Next1}]_{\mathrm{tr1vars}} \land$ Tr1Fairness

Invariant1 $\triangleq$
$\land$ ready_to_sort = TRUE $\implies$ channel = adr_next
$\land$ current $\in$ PARCELS $\implies$ channel = adr[current]
$\land$ next $\in$ PARCELS $\implies$ adr_next = adr[next]
$\land \forall$ p $\in$ PARCELS : p $\in$ DOMAIN stored $\implies$ stored[p] = adr[p]

Liveness1 $\triangleq$
$\land$ $\forall$ p $\in$ TO_SORT $\leadsto$ stored[p] = adr[p]
$\land$ $\forall$ p $\in$ TO_SORT $\leadsto$ stored[p] $\in$ BASKETS

THEOREM Spec1 $\implies \Box$Init1

THEOREM Spec1 $\implies \Box$TypeInvariant1

THEOREM Spec1 $\implies \Box$Invariant1

THEOREM Spec1 $\implies$ Liveness1

Module des instances finies :

```
MODULE MCParcel_routing1
EXTENDS Naturals
VARIABLES channel, next, current, stored, sorted, ready_to_sort, adr_next
PARCELS == 1..5
EPARCELS == 0..5
noBaskets == 20
NODES == 0..14
BASKETS == 7..14
adr == [p ∈ PARCELS ↦
        CASE p=1 → 7
        [ ] p=2 → 10
        [ ] p=3 → 12
        [ ] p=4 → 9
        [ ] p=5 → 14]
INSTANCE Parcel_routing1
```

Module de configuration :
La clause PROPERTIES informe TLC de vérifier que la spécification implique les deux propriétés Spec et Liveness1. La première exprime une relation de raffinement (Spec1 ⇒ Spec) et la deuxième exprime une propriété de vivacité à vérifier par la spécification (Spec1 ⇒ Liveness1).

```
MODULE MCParcel_routing1
SPECIFICATION Spec1
INVARIANTS TypeInvariant1 Invariant1
PROPERTIES Spec Liveness1
```

4 Module du deuxième raffinement

```
MODULE Parcel_routing2
EXTENDS Tri1, TLC
CONSTANTS rac, access, switches, right_n, left_n, depth, T, noNode
VARIABLES exit, nd, lng
ASSUME    rac ∈ NODES
        ∧ rac ∉ BASKETS
        ∧ BASKETS ⊆ NODES
        ∧ BASKETS ≠ NODES
        ∧ switches ⊆ NODES
        ∧ switches ≠ NODES
        ∧ switches ≠ {}
        ∧ access ∈ [switches → SUBSET BASKETS]
        ∧ right_n ∈ [switches → NODES]
        ∧ left_n ∈ [switches → NODES]
        ∧ noNode ∉ NODES
        ∧ switches = NODES \ BASKETS
        ∧ depth ∈ [NODES → (0 .. T)]
        ∧ depth[rac] = 0 ∧ ∀ s ⊆ NODES : rac ∈ NODES ∧ (left_n ∨ right_n)[s] ⊆ s ⟹ NODES ⊆ s
        ∧ ∀ n ∈ switches : right_n[n] ≠ left_n[n]
        ∧ ∀ m ∈ switches : right_n[m] ≠ m
        ∧ ∀ k ∈ switches : left_n[k] ≠ k
        ∧ ∀ q ∈ switches : right_n[q] ∈ switches ∧ left_n[q] ∈ switches ⟹ access[right_n[q]] ∧ access[left_n[q]] = {}
        ∧ ∀ r ∈ switches : right_n[r] ∈ switches ∧ left_n[r] ∈ switches ⟹ access[r] = access[right_n[r]] ∨ access[left_n[r]]
```

$\land \forall p \in BASKETS : depth[p] = T$
$\land \forall e \in switches : depth[right\_n[e]] = depth[e] + 1$
$\land \forall f \in switches : depth[left\_n[f]] = depth[f] + 1$

$TypeInvariant2 \triangleq \land TypeInvariant1$
$\land exit \in [switches \to NODES \cup \{noNode\}]$
$\land nd \in NODES$
$\land lng \in (0 \,..\, T)$

$Init2 \triangleq Init \land nd = rac$
$\land lng = 0$
$\land exit = [i \in switches \mapsto noNode]$

$Select\_parcel2 \triangleq \land Select\_parcel$
$\land nd' = rac$
$\land lng' = 0$
$\land \text{UNCHANGED } \langle exit \rangle$

$Set\_switch \triangleq \land nd \in switches$
$\land current \in UNDEF$
$\land next \in PARCELS$
$\land lng < T - 1$
$\land ready\_to\_sort = \text{FALSE}$
$\land \exists node \in \{right\_n[nd], left\_n[nd]\} : node \notin BASKETS \land adr\_next \in access[node]$
$\land exit' = [exit \text{ EXCEPT } ![nd] = node] \land nd' = node \land lng' = lng + 1$
$\land \text{UNCHANGED } \langle channel, ready\_to\_sort, current, next, stored, sorted, adr\_next \rangle$

$Set\_channel2 \triangleq \land Set\_channel$
$\land nd \in switches$
$\land lng = T - 1$
$\land \exists node \in \{right\_n[nd], left\_n[nd]\} : node \in BASKETS \land adr\_next = node$
$\land exit' = [exit \text{ EXCEPT } ![nd] = adr\_next] \land nd' = adr\_next \land lng' = lng + 1$
$\land \text{UNCHANGED } \langle \rangle$

$Release2 \triangleq \land Release$
$\land nd \in BASKETS$
$\land nd' = rac$
$\land lng' = 0$
$\land \text{UNCHANGED } \langle exit \rangle$

$Cross\_node \triangleq \land current \in PARCELS$
$\land nd \in switches$
$\land exit[nd] \notin BASKETS$
$\land nd' = exit[nd]$
$\land lng' = lng + 1$
$\land \text{UNCHANGED } \langle channel, current, next, stored, sorted, ready\_to\_sort, adr\_next, exit \rangle$

$Cross\_sorting2 \triangleq \land current \in PARCELS$
$\land exit[nd] \in BASKETS$
$\land lng = T - 1$
$\land stored' = [stored \text{ EXCEPT } ![current] = exit[nd]]$
$\land sorted' = sorted \cup \{current\}$
$\land current' \in UNDEF$
$\land lng' = lng + 1$
$\land nd' = exit[nd]$
$\land \text{UNCHANGED } \langle channel, ready\_to\_sort, next, adr\_next, exit \rangle$

$Next2 \triangleq \lor Select\_parcel2$
$\lor Set\_switch$
$\lor Set\_channel2$
$\lor Release2$
$\lor Cross\_node$
$\lor Cross\_sorting2$

$tr2vars \triangleq \langle channel, next, current, stored, sorted, ready\_to\_sort, adr\_next, nd, lng, exit \rangle$

$Tr2Fairness \triangleq WF_{tr2vars}(Select\_parcel2) \wedge WF_{tr2vars}(Set\_switch)$
$\wedge WF_{tr2vars}(Set\_channel2) \wedge WF_{tr2vars}(Release2)$
$\wedge WF_{tr2vars}(Cross\_node) \wedge WF_{tr2vars}(Cross\_sorting2)$

$Spec2 \triangleq Init2 \wedge \Box[Next2]_{tr2vars} \wedge Tr2Fairness$

$Invariant2 \triangleq$
$\wedge next \in PARCELS \wedge current \notin PARCELS \wedge nd \neq rac \wedge nd \in switches \wedge right\_n[nd] \in BASKETS \vee left\_n[nd] \in BASKETS$
$\Longrightarrow channel = exit[nd]$
$\wedge current \in PARCELS \wedge exit[nd] \in BASKETS \Longrightarrow channel = exit[nd]$

$Liveness2 \triangleq$
$\wedge \forall p \in TO\_SORT \rightsquigarrow stored[p] \in BASKETS$
$\wedge \forall p \in TO\_SORT \rightsquigarrow stored[p] = adr[p]$

THEOREM $Spec2 \Longrightarrow Spec1$
THEOREM $Spec2 \Longrightarrow \Box Init2$
THEOREM $Spec2 \Longrightarrow \Box TypeInvariant2$
THEOREM $Spec2 \Longrightarrow \Box Invariant2$
THEOREM $Spec2 \Longrightarrow Liveness2$

Module des instances finies :

```
MODULE MCParcel_routing2
EXTENDS Naturals
CONSTANTS R, L, Indef
VARIABLES channel, next, current, stored, sorted, ready_to_sort, adr_next, nd,
          pos, exit, gate
PARCELS == 1..5
EPARCELS == 0..5
noBaskets == 20
NODES == 0..14
BASKETS == 7..14
adr == [p ∈ PARCELS ↦
    CASE p=1 → 7
    [ ] p=2 → 10
    [ ] p=3 → 12
    [ ] p=4 → 9
    [ ] p=5 → 14]
rac == 0
switches == 0..6
noNode == 20
right_n == [ n ∈ switches ↦
        CASE n=0 → 1
        [] n=1 → 3
        [] n=2 → 5
        [] n=3 → 7
        [] n=4 → 9
        [] n=5 → 11
        [] n=6 → 13 ]
left_n == [ n ∈ switches ↦
```

CASE n=0 → 2
[] n=1 → 4
[] n=2 → 6
[] n=3 → 8
[] n=4 → 10
[] n=5 → 12
[] n=6 → 14]

access == [n ∈ switches ↦
CASE n=0 → 7..14
[] n=1 → 7..10
[] n=2 → 11..14
[] n=3 → 7..8
[] n=4 → 9..10
[] n=5 → 11..12
[] n=6 → 13..14]

INSTANCE Parcel_routing2

Module de configuration :

MODULE MCParcel_routing2
SPECIFICATION Spec2
INVARIANTS TypeInvariant2 Invariant2
PROPERTIES Spec2 Liveness2

5 Module du troisième raffinement

MODULE Parcel_routing3

EXTENDS Parcel_routing2, TLC

CONSTANTS L, R, Indef

VARIABLES in, out_l, out_r, gate

OUT $\triangleq$ {L, R, Indef}

TypeInvariant3 $\triangleq$ ∧ TypeInvariant2
∧ in ∈ [NODES → BOOLEAN]
∧ out_l ∈ [switches → BOOLEAN]
∧ out_r ∈ [switches → BOOLEAN]
∧ gate ∈ [switches → OUT]

Init3 $\triangleq$ Init2
∧ in = [i ∈ NODES ↦ FALSE]
∧ out_l = [i ∈ switches ↦ FALSE]
∧ out_r = [i ∈ switches ↦ FALSE]
∧ gate = [i ∈ switches ↦ Indef]

Select_parcel3 $\triangleq$ ∧ Select_parcel2
∧ in′ = [in EXCEPT ![nd] = TRUE]
∧ UNCHANGED ⟨out_r, out_l, gate⟩

Set_gate_right $\triangleq$ ∧ next ∈ PARCELS
∧ nd ∈ switches
∧ ((right_n[nd] ∈ switches ∧ adr_next ∈ access[right_n[nd]]) ∨ (right_n[nd] ∈ BASKETS ∧ adr_next ∈ access[nd]))
∧ gate′ = [gate EXCEPT ![nd] = R]
∧ UNCHANGED ⟨channel, next, current, stored, sorted, ready_to_sort, adr_next, nd, lng, exit, out_r, out_l, in⟩

Set_gate_left $\triangleq$ ∧ next ∈ PARCELS
∧ nd ∈ switches

$\wedge$ ((left_n[nd] $\in$ switches $\wedge$ adr_next $\in$ access[left_n[nd]]) $\vee$ (left_n[nd] $\in$ BASKETS $\wedge$ adr_next $\in$ access[nd])) $\wedge$ g

$\wedge$ UNCHANGED $\langle$channel, next, current, stored, sorted, ready_to_sort, adr_next, nd, lng, exit, out_r, out_l, in$\rangle$

Set_switch3 $\triangleq$ $\wedge$ Set_switch

$\wedge \exists$ gate_nd $\in$ {R, L} : gate[nd] = gate_nd

$\wedge$ in$'$ = [in EXCEPT ![nd] = TRUE]

$\wedge$ UNCHANGED $\langle$out_r, out_l, gate$\rangle$

Set_channel3 $\triangleq$ $\wedge$ Set_channel2

$\wedge$ gate[nd] $\neq$ Indef

$\wedge \exists$ node $\in$ {right_n[nd], left_n[nd]} : node $\in$ BASKETS $\wedge$ adr_next = node $\wedge$ in$'$ = [in EXCEPT ![node] = TRUE]

$\wedge$ UNCHANGED $\langle$gate, out_r, out_l$\rangle$

Release3 $\triangleq$ $\wedge$ Release2

$\wedge$ in$'$ = [in EXCEPT ![nd] = TRUE]

$\wedge$ UNCHANGED $\langle$out_r, out_l, gate$\rangle$

Cross_right $\triangleq$ $\wedge$ current $\in$ PARCELS

$\wedge$ nd $\in$ switches

$\wedge$ in[nd] = TRUE

$\wedge$ gate[nd] = R

$\wedge$ out_l$'$ = [out_l EXCEPT ![nd] = FALSE]

$\wedge$ out_r$'$ = [out_r EXCEPT ![nd] = TRUE]

$\wedge$ in$'$ = [in EXCEPT ![nd] = FALSE]

$\wedge$ UNCHANGED $\langle$channel, next, current, stored, sorted, ready_to_sort, adr_next, nd, lng, exit, gate$\rangle$

Cross_left $\triangleq$ $\wedge$ current $\in$ PARCELS

$\wedge$ nd $\in$ switches

$\wedge$ in[nd] = TRUE

$\wedge$ gate[nd] = L

$\wedge$ out_l$'$ = [out_l EXCEPT ![nd] = TRUE]

$\wedge$ out_r$'$ = [out_r EXCEPT ![nd] = FALSE]

$\wedge$ in$'$ = [in EXCEPT ![nd] = FALSE]

$\wedge$ UNCHANGED $\langle$channel, next, current, stored, sorted, ready_to_sort, adr_next, nd, lng, exit, gate$\rangle$

Cross_node3 $\triangleq$ $\wedge$ Cross_node

$\wedge$ (out_r[nd] = TRUE $\vee$ out_l[nd] = TRUE)

$\wedge$ in$'$ = [in EXCEPT ![nd] = TRUE]

$\wedge$ UNCHANGED $\langle$out_r, out_l, gate$\rangle$

Cross_sorting3 $\triangleq$ $\wedge$ Cross_sorting2

$\wedge$ in$'$ = [in EXCEPT ![nd] = TRUE]

$\wedge$ UNCHANGED $\langle$out_r, out_l, gate$\rangle$

Next3 $\triangleq$ $\vee$ Select_parcel3

$\vee$ Set_gate_right

$\vee$ Set_gate_left

$\vee$ Set_switch3

$\vee$ Set_channel3

$\vee$ Release3

$\vee$ Cross_right

$\vee$ Cross_left

$\vee$ Cross_node3

$\vee$ Cross_sorting3

---

tr3vars $\triangleq$ $\langle$channel, next, current, stored, sorted, ready_to_sort, adr_next, nd, lng, exit, in, out_r, out_l, gate$\rangle$

Tr3Fairness $\triangleq$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Select_parcel3) $\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Set_gate_right)

$\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Set_gate_left) $\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Set_switch3)

$\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Set_channel3) $\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Release3)

$\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Cross_right) $\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Cross_left)

$\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Cross_node3) $\wedge$ $\mathrm{WF}_{\mathrm{tr3vars}}$(Cross_sorting3)

Spec3 $\triangleq$ Init3 $\wedge$ $\Box[\mathrm{Next3}]_{\mathrm{tr3vars}}$ $\wedge$ Tr3Fairness

Invariant3 $\triangleq$ $\wedge \forall$ i $\in$ switches : $\neg$(out_l[i] $\wedge$ out_r[i])

THEOREM Spec3 $\Longrightarrow$ Spec2

THEOREM Spec3 $\Longrightarrow$ $\Box$Init3

THEOREM Spec3 $\Longrightarrow$ $\Box$TypeInvariant3

THEOREM Spec3 $\Longrightarrow$ $\Box$Invariant3

THEOREM Spec3 $\Longrightarrow$ Liveness2

Module des instances finies :

```
MODULE MCParcel_routing3
EXTENDS Naturals
CONSTANTS R, L, Indef
VARIABLES channel, next, current, stored, sorted, ready_to_sort, adr_next, nd,
          pos, exit, gate, in, out_r, out_l
PARCELS == 1..5
EPARCELS == 0..5
noBaskets == 20
NODES == 0..14
BASKETS == 7..14
adr == [p ∈ PARCELS ↦
    CASE p=1 → 7
    [ ] p=2 → 10
    [ ] p=3 → 12
    [ ] p=4 → 9
    [ ] p=5 → 14]
rac == 0
switches == 0..6
noNode == 20
right_n == [ n ∈ switches ↦
          CASE n=0 → 1
          [] n=1 → 3
          [] n=2 → 5
          [] n=3 → 7
          [] n=4 → 9
          [] n=5 → 11
          [] n=6 → 13 ]
left_n == [ n ∈ switches ↦
          CASE n=0 → 2
          [] n=1 → 4
          [] n=2 → 6
          [] n=3 → 8
          [] n=4 → 10
          [] n=5 → 12
          [] n=6 → 14 ]
access == [ n ∈ switches ↦
         CASE n=0 → 7..14
         [] n=1 → 7..10
         [] n=2 → 11..14
         [] n=3 → 7..8
         [] n=4 → 9..10
```

```
        [] n=5 → 11..12
        [] n=6 → 13..14 ]
    INSTANCE Parcel_routing3
```

Module de configuration :

```
MODULE MCParcel_routing3
CONSTANTS
    R = R
    L = L
SPECIFICATION Spec3
INVARIANTS TypeInvariant3 Invariant3
PROPERTIES Spec2 Liveness2
```

Annexe D
Les modèles B des contrôleur et contrôlé du système de tri des paquets postaux

Nous présentons dans cet annexe, les modèles des contrôleur et contrôlé au niveau abstrait et au niveau des raffinements.

1 Niveau abstrait

1.1 Modèle du contrôlé

MODEL M_{op}
SETS
NODES ; EPARCELS
CONSTANTS
PARCELS, adr, Baskets
PROPERIES
$PARCELS \subset EPARCELS \wedge Baskets \subset NODES \wedge$
$adr \in \text{PARCELS} \rightarrow Baskets \wedge PARCELS \neq \varnothing \wedge Baskets \neq \varnothing$
VARIABLES
next, current, channel, stored, sorted, adr_next, go_set, go_release, sel_channel
INVARIANT
$next \in EPARCELS \wedge current \in EPARCELS \wedge stored \in PARCELS \nrightarrow Baskets \wedge$
$sorted \subseteq PARCELS \wedge channel \in Baskets \wedge adr\_next \in Baskets \wedge$
$go\_set \in BOOL \wedge go\_release \in BOOL \wedge sel\_channel \in Baskets$
DEFINITIONS
$TO\_SORT \triangleq PARCELS - sorted ;$
$UNDEF \triangleq EPARCELS - PARCELS$
INITIALISATION
$stored := \varnothing \;||\; current :\in UNDEF \;||\; next :\in UNDEF \;||$
$sorted := \varnothing \;||\; channel :\in Baskets \;||\; adr\_next :\in Baskets \;||$
$go\_set := FALSE \;||\; go\_release := FALSE \;||\; sel\_channel :\in Baskets$
EVENTS
Select_parcel $\triangleq$ **ANY** p **Where**

p ∈ *TO_SORT* ∧ *next* ∈ *UNDEF* ∧ *current* ∈ *UNDEF*
THEN
adr_next := *adr(p)*|| *next* := *p*
END ;
Set_channel_exe ≙ **SELECT**
go_set = *TRUE*
THEN
channel := *sel_channel* || *go_set* := *FALSE*
END ;
Release_exe ≙ **SELECT**
go_release = *TRUE*
THEN
current := *next* || *next* :∈ *UNDEF* || *go_release* := *FALSE*
END ;
Cross_sorting ≙ **SELECT**
current ∈ *PARCELS*
THEN
stored(current) := *channel* || *sorted* := *sorted* ∪ { *current*} ||
current :∈ *UNDEF*
END
END

1.2 Modèle du contrôleur

MODEL M_c
SETS
NODES ; EPARCELS
CONSTANTS
PARCELS, adr, Baskets
PROPERIES
PARCELS ⊂ *EPARCELS* ∧ *Baskets* ⊂ *NODES* ∧ *adr* ∈ PARCELS→ *Baskets* ∧
PARCELS ≠ ∅∧ *Baskets* ≠ ∅
VARIABLES
adr_next, ready_to_sort, current, next, go_set, go_release, sel_channel
INVARIANT
adr_next ∈ *Baskets* ∧ *ready_to_sort* ∈ *BOOL* ∧ *current* ∈ *EPARCELS* ∧
next ∈ *EPARCELS* ∧ *go_set* ∈ *BOOL* ∧ *go_release* ∈ *BOOL* ∧ *sel_channel* ∈ *Baskets*
DEFINITIONS
TO_SORT ≙ *PARCELS - sorted ;*
UNDEF ≙ *EPARCELS - PARCELS*
INITIALISATION
adr_next : ∈ *Baskets* || *current* : ∈ *UNDEF* || *next* : ∈ *UNDEF* ||
ready_to_sort := *FALSE* || *go_set* := *FALSE* ||
go_release := *FALSE* || *sel_channel* : ∈ *Baskets*
EVENTS
Set_channel_cmd ≙ **SELECT**
ready_to_sort = *FALSE* ∧ *next* ∈ *PARCELS* ∧

current $\in$ *UNDEF* $\wedge$ *go_set* = *FALSE*
THEN
sel_channel := *adr_next* || *ready_to_sort* := *TRUE* || *go_set* := *TRUE*
END ;
Release_cmd $\triangleq$ **SELECT**
next $\in$ *PARCELS* $\wedge$ *current* $\in$ *UNDEF* $\wedge$ *ready_to_sort* = *TRUE*
THEN
ready_to_sort := *FALSE* || *go_release* := *TRUE*
END
END

2 Premier raffinement

2.1 Modèle du contrôlé

REFINEMENT M^1_{op}
REFINES M_{op}
CONSTANTS
rac, access, switches, right_n, left_n, depth, T
PROPERIES
rac $\in$ *NODES* $\wedge$ *switches* = *NODES - Baskets* $\wedge$ *Baskets* $\subset$ *NODES* $\wedge$
access $\in$ *switches* $\rightarrow$ *POW1(Baskets)* $\wedge$ *access(rac)* = *Baskets* $\wedge$
right_n $\in$ *switches* $\rightarrow$ *NODES* $\wedge$ *left_n* $\in$ *switches* $\rightarrow$ *NODES* $\wedge$ *T* $\in$ *NATURAL1* $\wedge$
depth $\in$ NODES $\rightarrow$ 0..T $\wedge$ depth(rac) = 0$\wedge$
$\forall$s.(s $\subseteq$ NODES $\wedge$ rac $\in$ s $\wedge$ (left_n $\cup$ right_n)[s] $\subseteq$ s $\Rightarrow$ NODES $\subseteq$ s)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ right_n(n) $\neq$ left_n(n))$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ right_n(n) $\neq$ n)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ left_n(n) $\neq$ n)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ access(right_n(n)) $\cap$ access(left_n(n)) = $\varnothing\wedge$
access(n) = access(right_n(n)) $\cup$ access(left_n(n)))
$\forall$n.(n $\in$ BASKETS $\Rightarrow$ depth(n) = T)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ depth(right_n(n)) = depth(n) + 1)$\wedge$
$\forall$n.(n $\in$ switches $\Rightarrow$ depth(left_n(n)) = depth(n) + 1) $\wedge$
iterate(right_n, T)(rac) $\in$ *BASKETS* $\wedge$
iterate(left_n, T)(rac) $\in$ *BASKETS*
VARIABLES
stored, channel, current, next, adr_next, exit, nd, sorted, lng
go_set, go_release, sel_channel, go_switch, sel_exit
INVARIANT
exit $\in$ *switches* $\nrightarrow$ *NODES* $\wedge$ *nd* $\in$ *NODES* $\wedge$
go_switch $\in$ *BOOL* $\wedge$ *sel_exit* $\in$ *NODES* $\wedge$
(next $\in$ *PARCELS* $\wedge$ *current* $\notin$ *PARCELS* $\wedge$ *nd* $\neq$ *rac* $\wedge$ *nd* $\in$ *switches* $\wedge$
(right_n(nd) $\in$ *BASKETS* $\vee$ *left_n(nd)* $\in$ *BASKETS)* $\Rightarrow$ *channel* = *exit (nd))* $\wedge$
(current $\in$ PARCELS$\wedge$current $\notin$ dom(stored)$\wedge$nd $\neq$ rac$\wedge$exit(nd) $\in$ Baskets $\Rightarrow$ channel = exit(nd))
(exit $\neq \varnothing \Rightarrow$ *iterate(exit,T)(rac)=channel)*

INITIALIZATION

channel : ∈ *Baskets* || *stored* := ∅ || *next* : ∈ *UNDEF* ||
current : ∈ *UNDEF* || *adr_next* : ∈ *Baskets* || *sorted* := ∅ ||
go_set := *FALSE* || *go_release* := *FALSE* || *sel_channel* : ∈ *Baskets* ||
nd : ∈ *NODES* || *exit* : ∈ *switches* → *NODES* ||
go_switch := *FALSE* || *sel_exit* : ∈ *NODES*

EVENTS

Select_parcel = ANY *p* **Where**
p ∈ *TO_SORT* ∧ *next* ∈ *UNDEF* ∧
current ∈ *UNDEF*
THEN
adr_next := *adr(p)* || *next* := *p* || *nd* := *rac* || *lng* := *0*
END ;

Set_switch_exe = SELECT
go_switch = *TRUE*
THEN
exit(nd) := *sel_exit* || *nd* := *sel_exit* || *go_switch* := *FALSE* || *lng* := *lng* + *1*
END ;

Set_channel_exe ≙ SELECT
go_set = *TRUE*
THEN
exit(nd) := *sel_exit* || *go_set* := *FALSE*
END ;

Release_exe = SELECT
go_release = *TRUE*
THEN
current := *next* || *next* : ∈ *UNDEF* || *nd* := *rac* ||
go_release := *FALSE* || *lng* := *0* || *ready_to_sort* := *FALSE*
END ;

Cross_node ≙ SELECT
current ∈ *PARCELS* ∧ *exit(nd)* ∉ *BASKETS* ∧ *nd* ∈ *switches* ∧ *lng* < *T-1*
THEN
nd := *exit(nd)* || *lng* := *lng* + *1*
END ;

Cross_sorting ≙ SELECT
current ∈ *PARCELS* ∧ *exit(nd)* ∈ *Baskets* ∧ *lng* = *T-1*
THEN
stored(current) := *nd* || *sorted* := *sorted* ∪ {*current*} ||
current : ∈ *UNDEF* || *lng* := *lng* + *1*
END

END

2.2 Modèle du contrôleur

REFINEMENT M_c^1
REFINES M_c
CONSTANTS

rac, access, switches, right_n, left_n, depth, T

PROPERIES

rac ∈ *NODES* ∧ *switches* = *NODES - Baskets* ∧ *Baskets* ⊂ *NODES* ∧
access ∈ *switches* → *POW1(Baskets)* ∧ *access(rac)* = *Baskets* ∧
right_n ∈ *switches* → *NODES* ∧ *left_n* ∈ *switches* → *NODES* ∧ *T* ∈ *NATURAL1* ∧
depth ∈ NODES → 0..T ∧ depth(rac) = 0∧
∀s.(s ⊆ NODES ∧ rac ∈ s ∧ (left_n ∪ right_n)[s] ⊆ s ⇒ NODES ⊆ s)∧
∀n.(n ∈ switches ⇒ right_n(n) ≠ left_n(n))∧
∀n.(n ∈ switches ⇒ right_n(n) ≠ n)∧
∀n.(n ∈ switches ⇒ left_n(n) ≠ n)∧
∀n.(n ∈ switches ⇒ access(right_n(n)) ∩ access(left_n(n)) = ∅∧
access(n) = access(right_n(n)) ∪ access(left_n(n)))
∀n.(n ∈ BASKETS ⇒ depth(n) = T)∧
∀n.(n ∈ switches ⇒ depth(right_n(n)) = depth(n) + 1)∧
∀n.(n ∈ switches ⇒ depth(left_n(n)) = depth(n) + 1) ∧
iterate(right_n, T)(rac) ∈ *BASKETS* ∧
iterate(left_n, T)(rac) ∈ *BASKETS*

VARIABLES

ready_to_sort, current, next, adr_next, go_set, lng
go_release, sel_channel, nd, go_switch, sel_exit

INVARIANT

nd ∈ *NODES* ∧ *go_switch* ∈ *BOOL* ∧ *sel_exit* ∈ *NODES*

INITIALIZATION

next :∈ *UNDEF* || *current* :∈ *UNDEF* || *adr_next* :∈ *Baskets* ||
ready_to_sort := *FALSE* || *go_set* := *FALSE* || *go_release* := *FALSE* ||
sel_channel :∈ *Baskets* || *nd* :∈ *NODES* || *go_switch* := *FALSE* || *sel_exit* :∈ *NODES*

EVENTS

Set_switch_cmd = **ANY** *node* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧ *nd* ∈ *switches* ∧
node ∉ *BASKETS* ∧ *next* ∈ *PARCELS* ∧ *lng* < *T-1* ∧
current ∈ *UNDEF* ∧ *adr_next* ∈ *access(node)* ∧ *ready_to_sort* = *FALSE*
go_switch = *FALSE*
THEN
sel_exit := *node* || *go_switch* := *TRUE*
END ;

Set_channel_cmd ≙ **ANY** *node* **WHERE**
node ∈ {right_n(nd), left_n(nd)}∧ *nd* ∈ *switches* ∧
node ∈ *BASKETS* ∧ *adr_next* = *node* ∧ *next* ∈ *PARCELS* ∧
current ∈ *UNDEF* ∧ *ready_to_sort* = *FALSE* ∧ *lng* = *T-1* ∧ *go_set* = *FALSE*
THEN
sel_exit := *adr_next* || *ready_to_sort* := *TRUE* || *go_set* := *TRUE*
END

END

3 Deuxième Raffinement

3.1 Modèle du contrôlé

REFINEMENT M^2_{op}
REFINES M^1_{op}
SETS $OUT = \{L, R, Indef\}$
VARIABLES
stored, current, next, sorted, adr_next, go_set,
go_release, exit, nd, go_switch, sel_exit, in,
out_l , out_r , gate, go_gate_r, go_gate_l, sel_gate
INVARIANT
$in \in Node \rightarrow BOOL \wedge out\_r \in switches \rightarrow BOOL \wedge$
$out\_l \in switches \rightarrow BOOL \wedge$
$go\_gate\_r \in BOOL \wedge go\_gate\_l \in BOOL \wedge sel\_gate \in OUT \wedge$
$\forall i .(i \in switches \Rightarrow \neg (out\_r(i)=TRUE \wedge out\_l(i)=TRUE))$
INITIALIZATION
$stored := \varnothing \parallel next :\in UNDEF \parallel current :\in UNDEF \parallel$
$adr\_next :\in Baskets \parallel sorted := \varnothing \parallel go\_set := FALSE \parallel$
$go\_release := FALSE \parallel nd :\in NODES \parallel exit :\in switches \rightarrow NODES \parallel$
$go\_switch := FALSE \parallel sel\_exit :\in NODES \parallel in := NODES * \{FALSE\} \parallel$
$out\_r := switches * \{FALSE\} \parallel out\_l := switches * \{FALSE\} \parallel$
$gate := NODES * \{Indef\} \parallel go\_gate\_r := FALSE \parallel$
$go\_gate\_l := FALSE \parallel sel\_gate := Indef$
EVENTS
Set_gate_right_exe $\triangleq$ **SELECT**
$go\_gate\_r = TRUE$
THEN
$gate(nd) := sel\_gate \parallel go\_gate\_r := FALSE$
END ;
Set_gate_left_exe $\triangleq$ **SELECT**
$go\_gate\_l = TRUE$
THEN
$gate(nd) := sel\_gate \parallel go\_gate\_l := FALSE$
END ;
Cross_right $\triangleq$ **SELECT**
$nd \in switches \wedge in(nd) = TRUE \wedge gate(nd)= R$
THEN
$out\_l(nd) := FALSE \parallel out\_r(nd) := TRUE \parallel in(nd) := FALSE$
END ;
Cross_left $\triangleq$ **SELECT**
$nd \in switches \wedge in(nd) = TRUE \wedge gate(nd)= L$
THEN
$out\_l(nd) := TRUE \parallel out\_r(nd) := FALSE \parallel in(nd) := FALSE$
END ;
Cross_node $\triangleq$ **SELECT**
$current \in PARCELS \wedge exit(nd) \notin Baskets \wedge nd \in switches \wedge$

(out_l(nd)=TRUE or out_r(nd)=TRUE) $\wedge$ *lng < T-1*
THEN
nd := exit(nd) || *in(nd) := TRUE* || *lng := lng+1*
END ;
Cross_sorting $\triangleq$ **SELECT**
current $\in$ *PARCELS* $\wedge$ *exit(nd)* $\in$ *Baskets* $\wedge$ *lng = T-1*
THEN
stored(current) := nd || *sorted := sorted* $\cup$ *{current}* ||
current : $\in$ *UNDEF* || *lng := lng + 1* || *in(nd) := TRUE*
END ;
Release_exe $\triangleq$ **SELECT**
go_release = TRUE
THEN
current := next || *next :* $\in$ *UNDEF* || *nd := rac* || *lng := 0* ||
in(nd) := TRUE || *ready_to_sort := FALSE* || *go_release := FALSE*
END
END

3.2 Modèle du contrôleur

REFINEMENT $\mathrm{M}_{\mathrm{c}}^{2}$
REFINES $\mathrm{M}_{\mathrm{c}}^{1}$
SETS $OUT = \{\mathrm{L}, \mathrm{R}, \mathrm{Indef}\}$
VARIABLES
ready_to_sort, current, next, adr_next, go_set,
go_release, nd, go_switch, sel_exit, in,
out_l , out_r , gate, go_gate_r, go_gate_l, sel_gate
INVARIANT
in $\in$ *Node* $\rightarrow$ *BOOL* $\wedge$ *out_r* $\in$ *switches*$\rightarrow$ *BOOL* $\wedge$
out_l $\in$ *switches* $\rightarrow$ *BOOL* $\wedge$ *go_gate_r* $\in$ *BOOL* $\wedge$
go_gate_l $\in$ *BOOL* $\wedge$ *sel_gate* $\in$ *OUT* $\wedge$
$\forall$ *i .(i* $\in$ *switches* $\Rightarrow$ $\neg$ *(out_r(i)=TRUE* $\wedge$ *out_l(i)=TRUE))*
INITIALIZATION
ready_to_sort : $\in$ *BOOL* || *next :* $\in$ *UNDEF* || *current :* $\in$ *UNDEF* ||
adr_next : $\in$ *Baskets* || *go_set := FALSE* || *go_release := FALSE* ||
nd : $\in$ *NODES* || *go_switch := FALSE* || *sel_exit :* $\in$ *NODES* ||
*in := NODES ** {FALSE} || *out_r := switches ** {FALSE} ||
*out_l := switches ** {FALSE} || *gate := NODES ** {Indef} ||
go_gate_r := FALSE || *go_gate_l := FALSE* || *sel_gate := Indef*
EVENTS
Set_gate_right_cmd $\triangleq$ **SELECT**
next $\in$ *PARCELS* $\wedge$
((right_n(nd) $\in$ *switches* $\wedge$ *adr_next* $\in$ *access(right_n(nd))) or*
(nd $\in$ *switches* $\wedge$ *right_n(nd)* $\in$ *BASKETS* $\wedge$ *adr_next* $\in$ *access(nd))))* $\wedge$
go_gate_r = FALSE
THEN
sel_gate := R || *go_gate_r := TRUE*

END ;

Set_gate_left_cmd ≙ **SELECT**

next ∈ *PARCELS* ∧

((left_n(nd) ∈ *switches* ∧ *adr_next* ∈ *access(left_n(nd))) or*

(nd ∈ *switches* ∧ *left_n(nd)* ∈ *BASKETS* ∧ *adr_next* ∈ *access(nd)))*

go_gate_l = FALSE

THEN

sel_gate := L || go_gate_l := TRUE

END ;

Set_switch_cmd ≙ **ANY** *node, gate_nd* **WHERE**

node ∈ {right_n(nd), left_n(nd)}∧

nd ∈ *switches* ∧ *gate_nd* ∈ {R, L} ∧ *node* ∉ *BASKETS* ∧

current ∈ *UNDEF* ∧ *lng < T-1* ∧ *adr_next* ∈ *access(node)* ∧

ready_to_sort = FALSE ∧ *in(nd) = TRUE* ∧ *next* ∈ *PARCELS* ∧

in(nd) = TRUE ∧ *gate(nd) = gate_nd* ∧*in(nd) = TRUE*

THEN

sel_exit := node || go_switch := TRUE

END

END

Annexe E
Les modèles B du système de production manufacturière

Nous présentons dans cet annexe les modèles B correspondants au système de production manufacturière.

1 Les modèles B

1.1 Modèle du premier raffinement

REFINEMENT Mixeur1
REFINES Mixeur
SETS
ETAPES = {commencer, ouvrir_VA, verser_VA, ouvrir_VB, verser_VB, ouvrir_VC, transvaser, verser_brique, malaxer, fabriquer, livrer, basculer, terminer, marche_moteur_conv, marche_moteur_mal, marche_moteur_liv, marche_moteur_bas }
VARIABLES
loc1_produit, prod, step
INVARIANT
prod ∈ (0..ma)(0..mb)*(0..n)*(0..ts) ∧ step ∈ ETAPES ∧ loc1_produit ∈ LIEUX ∧*
/ Invariants de collage*/*
(en_marche = FALSE ⇒ step = commencer) ∧
(step = commencer ⇒ en_marche = FALSE) ∧
(step ∈ {malaxer,livrer} ⇒ en_marche = TRUE) ∧
(loc_produit = no_where ⇒ loc1_produit ∈ {no_where, in}) ∧
(loc1_produit = no_where ⇒ loc_produit = no_where) ∧
(loc_produit = in ⇒ loc1_produit = in) ∧
(loc1_produit = in ⇒ loc_produit ∈ {no_where,in}) ∧
(loc_produit = out ⇒ loc1_produit = out) ∧
(loc1_produit = out ⇒ loc_produit = out) ∧
(step ∈ {verser_VA,verser_VB,verser_brique,malaxer} ⇒ loc_produit = no_where) ∧
(step = livrer ∧ loc1_produit = in ⇒ loc_produit = in)
INITIALISATION

loc1_produit := no_where || prod := (0↦0↦0↦0) || step := commencer

EVENTS

Start = SELECT
step = commencer
THEN
step := verser_VA || prod := (0↦0↦0↦0) ||
loc1_produit := no_where
END ;

Obtenir_VA = SELECT
step = verser_VA ∧ loc1_produit = no_where
THEN
prod := (ma↦0↦0↦0) || step := verser_VB || loc1_produit := in
END ;

Obtenir_VB = SELECT
step = verser_VB ∧ loc1_produit = in
THEN
prod := (ma↦mb↦0↦0) || step := verser_brique
END ;

Obtenir_briques = SELECT
step = verser_brique ∧ loc1_produit = in
THEN
prod := (ma↦mb↦n↦0) || step := malaxer
END ;

Fabriquer = SELECT
step = malaxer ∧ loc1_produit = in
THEN
prod := (ma↦mb↦n↦tm) || step := livrer
END ;

Livrer = SELECT
step = livrer ∧ loc1_produit = in
THEN
loc_produit := out
END

Finir = SELECT
loc1_produit = out
THEN
step := commencer
END

END

1.2 Modèle du deuxième raffinement

REFINEMENT Mixeur2
REFINES Mixeur1
SETS
POSITION_MIX = {verticale,horizontale}
VARIABLES

prod, loc_prod, step2, position

INVARIANT

loc_prod ∈ LIEUX ∧ step2 ∈ ETAPES ∧ position ∈ POSITION_MIX ∧
/ Invariants de collage */*
(step2 ∈ {commencer, verser_VA, malaxer, fabriquer, livrer} ∧
(loc_prod = mixeur ⇒ step2 ∈ {verser_brique, malaxer, livrer})
(loc1_produit = no_where ⇒ loc_prod = no_where) ∧
(loc_prod = no_where ⇒ loc1_produit = no_where) ∧
(loc1_produit = in ⇒ loc_prod ∈ {cuve, mixeur}) ∧
(loc_prod ∈ {cuve, mixeur} ⇒ loc1_produit = in) ∧
(loc1_produit = out ⇒ loc_prod = out) ∧
(loc_prod = out ⇒ loc1_produit = out) ∧
(step2 = basculer ⇒ step = livrer) ∧
(step2 : verser_brique,transvaser ⇒ step = verser_brique) ∧
(step2 = basculer ⇒ step = livrer)

INITIALISATION

prod := (0↦0↦0↦0) || loc_prod := sortie || step2 := commencer || position := verticale

EVENTS

Start = SELECT *step2 = commencer*
THEN
step2 := verser_VA || prod := (0↦0↦0↦0) ||
loc_prod := no_where
END ;

Obtenir_VA = SELECT
step2 = verser_VA ∧ loc_prod = no_where
THEN
prod := (ma↦0↦0↦0) || step2 := verser_VB ||
loc_prod := cuve
END ;

Obtenir_VB = SELECT
step2 = verser_VB ∧ loc_prod= cuve
THEN
prod := (ma↦mb↦0↦0) || step2 := transvaser
END ;

Transvaser = SELECT
step2 = transvaser ∧ loc_prod = cuve
THEN
loc_prod := mixeur || step2 := verser_brique
END ;

Obtenir_briques = SELECT
step2 = verser_brique ∧ loc_prod = mixeur
THEN
prod := (ma↦mb↦n↦0) || step2 := malaxer
END ;

Fabriquer = SELECT
step2 = malaxer ∧ loc_prod = mixeur
THEN
prod := (ma↦mb↦n↦tm) || step2 := basculer

END ;
Basculer = SELECT
step2 = basculer ∧ position = verticale
THEN
position := horizontale || step2 := livrer
END ;
Livrer = SELECT
step2 = livrer ∧ loc_prod = mixeur
THEN
loc_prod := out
END
Finir = SELECT
loc_prod = out
THEN
step2 := commencer
END
END

1.3 Modèle du troisième raffinement

REFINEMENT Mixeur3
REFINES Mixeur2
SETS
VAL = {close, open} ; MOTEUR = {on, off}
VARIABLES
va, vb, vc, step3, moteur_convoyer, moteur_malaxer, moteur_livrer, moteur_tourner, angle, contenu, position, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint
INVARIANT
contenu ∈ LIEUX ⇸ (0..ma)(0..mb)*(0..n)*(0..ts) ∧*
moteur_convoyer ∈ MOTEUR ∧ moteur_malaxer ∈ MOTEUR ∧
moteur_tourner ∈ MOTEUR ∧ moteur_livrer ∈ MOTEUR ∧
va ∈ VAL ∧ vb ∈ VAL ∧ vc ∈ VAL ∧
step3 ∈ ETAPES ∧ angle ∈ 0..90 ∧
/ Invariants de liaison */*
(step3 ∈ {commencer,fabriquer} ⇒ step2 = step3) ∧
(step2 ∈ {commencer,fabriquer} ⇒ step3 = step2) ∧
(step3 ∈ {ouvrir_VA,verser_VA} ⇒ step2=verser_VA) ∧
(step2 = verser_VA ⇒ step3 ∈ {ouvrir_VA,verser_VA}) ∧
(step3 ∈ {ouvrir_VB,verser_VB} ⇒ step2=verser_VB) ∧
(step2 = verser_VB ⇒ step3 ∈ {ouvrir_VB,verser_VB}) ∧
(step3 ∈ {ouvrir_VC,transvaser} ⇒ step2=transvaser) ∧
(step2 = transvaser ⇒ step3 ∈ {ouvrir_VC,transvaser}) ∧
(step3 ∈ marche_moteur_conv,verser_brique ⇒ step2=verser_brique) ∧
(step2=verser_brique ⇒ step3 ∈ marche_moteur_conv,verser_brique) ∧
(step3 ∈ marche_moteur_mal, malaxer ⇒ step2=malaxer) ∧
(step2=malaxer ⇒ step3 ∈ marche_moteur_mal, malaxer) ∧

$(step3 \in marche\_moteur\_bas, basculer \Rightarrow step2=basculer) \wedge$
$(step2=basculer \Rightarrow step3 \in marche\_moteur\_bas, basculer) \wedge$
$(step3 \in marche\_moteur\_liv, livrer, terminer \Rightarrow step2=livrer) \wedge$
$(step2=livrer \Rightarrow step3 \in marche\_moteur\_liv, livrer, terminer) \wedge$
$(step3 \in ouvrir\_VA, verser\_VA \Rightarrow loc\_prod = no\_where) \wedge$
$(step3 = commencer \Rightarrow loc\_prod \in no\_where, out) \wedge$
$(step3 \in ouvrir\_VB, verser\_VB, ouvrir\_VC, transvaser \Rightarrow loc\_prod = cuve) \wedge$
$(loc\_prod = cuve \Rightarrow step3 \in ouvrir\_VB, verser\_VB, ouvrir\_VC, transvaser) \wedge$
$(step3 \in verser\_brique, marche\_moteur\_mal, malaxer, marche\_moteur\_bas,$
$marche\_moteur\_conv, marche\_moteur\_liv, basculer, livrer \Rightarrow loc\_prod = mixeur) \wedge$
$(step3 = terminer \Rightarrow loc\_prod = out) \wedge$
$/* Propriétés de comportement */$
$\forall(x,y).(x \in 0..ma \wedge y \in 0..mb \wedge vc=close \wedge vb=open \wedge$
$contenu(cuve) = (x \mapsto y \mapsto 0 \mapsto 0) \wedge y>0 \Rightarrow x=ma) \wedge$
$\forall(x,y,z).(x \in 0..ma \wedge y \in 0..mb \wedge z \in 0..n \wedge$
$contenu(mixeur) = (x \mapsto y \mapsto z \mapsto 0) \wedge z>0 \Rightarrow x=ma \wedge y=mb) \wedge$
$\forall(x,y,z,t).(x \in 0..ma \wedge y \in 0..mb \wedge z \in 0..n \wedge t \in 0..ts \wedge$
$contenu(mixeur) = (x \mapsto y \mapsto z \mapsto t) \wedge t>0 \Rightarrow x=ma \wedge y=mb \wedge z=n) \wedge$
$/* Propriétés de sûreté */$
$\neg(vc=open \wedge angle=0) \wedge \neg(moteur\_convoyer=on \wedge angle=0) \wedge$
$\neg(moteur\_malaxer=on \wedge angle=0) \wedge \neg(vc=open \wedge moteur\_convoyer=on) \wedge$
$\neg(vc=open \wedge moteur\_malaxer=on) \wedge \neg(va=open \wedge vb=open) \wedge$
$\neg(va=open \wedge vc=open) \wedge \neg(vb=open \wedge vc=open)$

INITIALISATION

$va := close \parallel vb := close \parallel vc := close \parallel angle := 90 \parallel step3 := commencer \parallel$
$contenu := LIEUX\ *(0 \mapsto 0 \mapsto 0 \mapsto 0) \parallel moteur\_convoyer := off \parallel$
$moteur\_malaxer := off \parallel moteur\_tourner := off \parallel moteur\_livrer := off \parallel$
$position := verticale \parallel W0\_atteint := FALSE \parallel WA\_atteint := FALSE \parallel$
$WB\_atteint := FALSE \parallel WC\_atteint := FALSE \parallel n\_atteint := FALSE \parallel$
$time\_atteint := FALSE \parallel angle\_atteint := FALSE \parallel produit\_livrer := FALSE$

EVENTS

Start = SELECT
$step3 = commencer$
THEN
$step3 := ouvrir\_VA \parallel contenu := LIEUX\ *(0 \mapsto 0 \mapsto 0 \mapsto 0) \parallel$
$angle := 90 \parallel va := close \parallel vb := close \parallel vc := close \parallel$
$moteur\_convoyer := off \parallel moteur\_malaxer := off \parallel moteur\_tourner := off$
END ;

Ouvrir_VA = SELECT
$step3 = ouvrir\_VA \wedge va = close \wedge vb = close \wedge vc = close \wedge$
$contenu(cuve) = (0 \mapsto 0 \mapsto 0 \mapsto 0)$
THEN
$va := open \parallel step3 := verser\_VA$
END ;

Remplissage_cuveAC = ANY x **WHERE**
$x \in 0..ma\text{-}1 \wedge contenu(cuve) = (x \mapsto 0 \mapsto 0 \mapsto 0) \wedge va=open$
THEN
$contenu(cuve) := (x+1 \mapsto 0 \mapsto 0 \mapsto 0)$

END ;
Detect_WA = SELECT
contenu(cuve) = (ma↦0↦0↦0)
THEN
WA_atteint := TRUE
END ;
Obtenir_VA = SELECT
step3 = verser_VA ∧ WA_atteint = TRUE ∧ va = open
THEN
step3 := ouvrir_VB || va := close
END ;
Ouvrir_VB = SELECT
step3 = ouvrir_VB ∧ va = close ∧ vb = close ∧ vc = close ∧
contenu(cuve) = (ma↦0↦0↦0)
THEN
vb := open || step3 := verser_VB
END ;
Remplissage_cuveBC = ANY *y* WHERE
y ∈ 0..mb-1 ∧ vb=open
contenu(cuve) = (ma↦y↦0↦0)
THEN
contenu(cuve) := (ma↦y+1↦0↦0)
END ;
Detect_WB = SELECT
contenu(cuve) = (ma↦mb↦0↦0)
THEN
WB_atteint := TRUE
END ;
Obtenir_VB = SELECT
step3 = verser_VB ∧ WB_atteint = TRUE ∧ vb = open
THEN
step3 := ouvrir_VC || vb := close
END ;
Init_transvaser = SELECT
step3 = ouvrir_VC ∧ contenu(mixeur)= (0↦0↦0↦0) ∧
contenu(cuve) = (ma↦mb↦0↦0) ∧
vc = close ∧ va = close ∧ vb = close ∧
angle = 90 ∧ moteur_convoyer = off ∧ moteur_malaxer = off
THEN
vc := open || step3 := transvaser
END ;
Laisser_transvaser = ANY *x,y,u,v,a,b,c,d* WHERE
x ∈ 0..ma ∧ y ∈ 0..mb ∧ u ∈ 0..ma ∧
v ∈ 0..mb ∧ x+y ∈ 1..ma+mb ∧ u+v ∈ 0..ma+mb-1 ∧
x+y+u+v = ma+mb ∧ x+u = ma ∧ y+v = mb ∧
a ∈ 0..ma ∧ b ∈ 0..mb ∧ c ∈ 0..ma ∧
d ∈ 0..mb ∧ a+c=ma ∧ b+d=mb ∧
a+b ∈ 1..ma+mb ∧ c+d ∈ 0..ma+mb-1 ∧ a+b+c+d = ma+mb ∧ vc = open ∧

$contenu(cuve) = (x \mapsto y \mapsto 0 \mapsto 0) \wedge$
$contenu(mixeur) = (u \mapsto v \mapsto 0 \mapsto 0) \wedge angle = 90$
THEN
$contenu \in (contenu : LIEUX \nrightarrow (0..ma)*(0..mb)*(0..n)*(0..ts) \wedge$
$contenu(cuve)=(a \mapsto b \mapsto 0 \mapsto 0) \wedge a+b=x+y\text{-}1 \wedge$
$contenu(mixeur)=(c \mapsto d \mapsto 0 \mapsto 0) \wedge c+d=u+v+1)$
END ;

Detect_WC = SELECT
$contenu(mixeur) = (ma \mapsto mb \mapsto 0 \mapsto 0)$
THEN
$WC\_atteint := TRUE$
END ;

Transvaser = SELECT
$step3 = transvaser \wedge WC\_atteint = TRUE \wedge vc = open$
THEN
$step3 := marche\_moteur\_conv \parallel vc := close$
END ;

Init_convoyer = SELECT
$step3 = marche\_moteur\_conv \wedge$
$moteur\_convoyer = off \wedge angle=90 \wedge vc=close$
THEN
$moteur\_convoyer := on \parallel step3 := verser\_brique$
END ;

Laisser_tomber_briques = ANY *nb* WHERE
$nb : 0..n\text{-}1 \wedge moteur\_convoyer = on$
$contenu(mixeur) = (ma \mapsto mb \mapsto nb \mapsto 0)$
THEN
$contenu(mixeur) := (ma \mapsto mb \mapsto nb+1 \mapsto 0)$
END ;

Detect_briques = SELECT
$contenu(mixeur) = (ma \mapsto mb \mapsto nb \mapsto 0)$
THEN
$n\_atteint := TRUE$
END ;

Obtenir_briques = SELECT
$step3 = verser\_brique \wedge moteur\_convoyer = on \wedge n\_atteint = TRUE$
THEN
$step3 := marche\_moteur\_mal \parallel moteur\_convoyer := off$
END ;

Init_malaxer = SELECT
$step3 = marche\_moteur\_mal \wedge$
$moteur\_malaxer = off \wedge angle = 90 \wedge vc=close$
THEN
$moteur\_malaxer := on \parallel step3 := malaxer$
END ;

Laisser_malaxer = ANYt **WHERE**
$t \in 0..ts\text{-}1 \wedge moteur\_malaxer = on \wedge$
$contenu(mixeur) = (ma \mapsto mb \mapsto n \mapsto t)$

THEN
contenu(mixeur) := (ma↦mb↦n↦t+1)
END ;
Detect_temps = SELECT
contenu(mixeur) = (ma↦mb↦n↦tm)
THEN
time_atteint := TRUE
END ;
Fabriquer = SELECT
step3 = malaxer ∧ moteur_malaxer = on ∧ time_atteint = TRUE
THEN
step3 := marche_moteur_bas || moteur_malaxer := off
END ;
Init_basculer = SELECT
step3 = marche_moteur_bas ∧ moteur_tourner = off
THEN
moteur_tourner := on || step3 := basculer
END ;
Laisser_basculer = SELECT
moteur_tourner = on ∧ angle : 1..90 ∧
moteur_convoyer = off ∧ moteur_malaxer = off ∧ vc=close
THEN
angle := angle -1
END ;
Detect_angle = SELECT
angle = 0
THEN
angle_atteint := TRUE
END ;
Basculer = SELECT
moteur_tourner = on ∧ step3 = basculer ∧
angle_atteint = TRUE ∧ position = verticale
THEN
moteur_tourner := off || position := horizontale || step3 := marche_moteur_liv
END ;
Init_livrer = SELECT
step3 = marche_moteur_liv ∧ moteur_livrer = off
THEN
moteur_livrer := on || step3 := livrer
END ;
Laisser_livrer = SELECT
moteur_livrer = on ∧
THEN
contenu ∈ (contenu ∈ LIEUX+->(0..ma)(0..mb)*(0..n)*(0..ts) ∧*
contenu(mixeur)=(0↦0↦0↦0) ∧
contenu(cuve)= (0↦0↦0↦0)
contenu(out)=(ma↦mb↦n↦tm))
END ;

Detect_produit = SELECT
contenu(out)=(ma↦mb↦n↦tm)
THEN
produit_livrer := TRUE
END ;
Livrer = SELECT
step3 = livrer ∧ moteur_livrer = on ∧ produit_livrer = TRUE
THEN
moteur_livrer := off || step3 := terminer
END ;
Finir = SELECT
step3 = terminer
THEN
step3 := commencer
END
END

1.4 Modèle du quatrième raffinement

REFINEMENT Mixeur4
REFINES Mixeur3
CONSTANTS *d_max*
PROPERTIES *d_max ∈ NATURAL1*
VARIABLES *.....d, detect*
INVARIANT
detect ∈ BOOL ∧ d ∈ 0..d_max
INITIALISATION
.....d := 0 || detect := FALSE
EVENTS
.....
Avancer_brique = SELECT
moteur_convoyeur = on ∧ d < d_max ∧ detect = FALSE
THEN *d := d+1*
END ;
Detect_brique = SELECT
moteur_convoyeur = on ∧ d = d_max ∧ detect = FALSE
THEN
detect := TRUE || d := 0
END ;
Laisser_tomber_briques = ANY *nb* WHERE
nb : 0..n-1 ∧ moteur_convoyer = on ∧ detect = TRUE
contenu(mixeur) = (ma↦mb↦nb↦0)
THEN
contenu(mixeur) := (ma↦mb↦nb+1↦0) || detect := FALSE
END ;
.....
END

Annexe F
Les modèles B des contrôleurs et contrôlés du système de production manufacturière

Nous présentons dans cet annexe, les modules B des contrôleurs et contrôlés du système de production manufacturière au niveau abstrait ainsi qu'aux niveaux des quatre raffinements.

1 Niveau abstrait du système

1.1 Modèle du contrôlé

MODEL OP0
SETS
LIEUX = {no_where,in,out,cuve,mixeur}
CONSTANTS
ma, mb, n, tm, ts
PROPERTIES
$ma \in NATURAL1 \wedge mb \in NATURAL1 \wedge n \in NATURAL1 \wedge$
$tm \in NATURAL1 \wedge ts \in 0..tm$
VARIABLES
en_marche, loc_produit, produit,
go_start, go_fabriquer, go_livrer, /* Variables de commande */
init_done, fabrication_done, livraison_done, /* variables d'acquittement */
qa, qb, nb, t /* variables d'échange de valeur */
INVARIANT
$produit \in (0..ma)*(0..mb)*(0..n)*(0..tm) \wedge en\_marche \in BOOL \wedge$
$loc\_produit : LIEUX \wedge go\_start \in BOOL \wedge go\_fabriquer \in BOOL \wedge go\_livrer \in BOOL \wedge$
$init\_done \in BOOL \wedge fabrication\_done \in BOOL \wedge livraison\_done \in BOOL \wedge$
$qa \in 0..ma \wedge qb \in 0..mb \wedge nb \in 0..n \wedge t \in 0..tm$
INITIALISATION
$produit := (0 \mapsto 0 \mapsto 0 \mapsto 0) \parallel en\_marche := FALSE \parallel loc\_produit := no\_where \parallel$
$go\_start := FALSE \parallel go\_fabriquer := FALSE \parallel go\_livrer := FALSE \parallel$
$init\_done := FALSE \parallel fabrication\_done := FALSE \parallel livraison\_done := FALSE \parallel$
$qa := 0 \parallel qb := 0 \parallel nb := 0 \parallel t := 0$

EVENTS
/* Sous système de fabrication */
Init_OP = SELECT *go_start = TRUE*
THEN
produit := (0↦0↦0↦0) || loc_produit := no_where ||
init_done := TRUE || go_start := FALSE
END ;
Fabriquer_OP = SELECT *go_fabriquer = TRUE*
THEN
produit := (qa↦qb↦n↦ts) || loc_produit := in ||
fabrication_done := TRUE || go_fabriquer := FALSE
END ;
/* Sous système de livraison */
Livrer_OP = SELECT *go_livrer = TRUE*
THEN
loc_produit := out || livraison_done := TRUE || go_livrer := FALSE
END
END

1.2 Modèle du contrôleur

MODEL C0
SETS
LIEUX = {no_where,in,out,cuve,mixeur}
CONSTANTS
ma, mb, n, tm, ts
PROPERTIES
ma ∈ NATURAL1 ∧ mb ∈ NATURAL1 ∧ n ∈ NATURAL1 ∧
tm ∈ NATURAL1 ∧ ts ∈ 0..tm
VARIABLES
en_marche, loc_produit, produit, hand
go_start, go_fabriquer, go_livrer, / Variables de commande */*
init_done, fabrication_done, livraison_done, / variables d'acquittement */*
qa, qb, nb, t / variables d'échange de valeur */*
INVARIANT
produit ∈ (0..ma)(0..mb)*(0..n)*(0..tm) ∧ en_marche ∈ BOOL ∧ loc_produit : LIEUX ∧*
go_start ∈ BOOL ∧ go_fabriquer ∈ BOOL ∧ go_livrer ∈ BOOL ∧
init_done ∈ BOOL ∧ fabrication_done ∈ BOOL ∧ livraison_done ∈ BOOL ∧
qa ∈ 0..ma ∧ qb ∈ 0..mb ∧ nb ∈ 0..n ∧ t ∈ 0..tm ∧ hand ∈ NATURAL
INITIALISATION
produit := (0↦0↦0↦0) || en_marche := FALSE || loc_produit := no_where ||
go_start := FALSE || go_fabriquer := FALSE || go_livrer := FALSE ||
init_done := FALSE || fabrication_done := FALSE || livraison_done := FALSE ||
qa := 0 || qb := 0 || nb := 0 || t := 0 || hand := 1
EVENTS
/* Sous système de fabrication */
Start_C = SELECT

en_marche = FALSE ∧ go_start = FALSE ∧ hand = 1
THEN
en_marche := TRUE || go_start := TRUE
END ;
Fabriquer_C = SELECT
en_marche = TRUE ∧ go_fabriquer = FALSE ∧ hand = 1 ∧
init_done = TRUE ∧ loc_produit = no_where
THEN
qa :=ma || qb := mb || nb := n || t := ts ||
go_fabriquer := TRUE || init_done := FALSE
END ;
Terminer_C = SELECT
hand = 1 ∧ fabrication_done = TRUE
THEN
fabrication_done := FALSE || hand := 2
END ;
/* Sous système de livraison */
Livrer_C = SELECT
go_livrer = FALSE ∧ hand = 2 ∧ loc_produit = in ∧ hand = 2
THEN
go_livrer := TRUE **END ;**
Finir = SELECT
livraison_done = TRUE ∧ loc_produit = out
THEN
en_marche := FALSE || livraison_done := FALSE
END
END

2 Premier raffinement

2.1 Modèle du contrôlé

REFINEMENT OP1
REFINES OP0
VARIABLES
prod, loc1_produit,
/ Variables de commande */*
go_start, go_livrer, go_VA, go_VB, go_brique, go_malaxer,
/ Variables d'échange de valeur */*
qr_a, qr_b, nbr, tr,
/ Variables d'acquittement */*
init1_done, livraison_done, VA_done, VB_done, brique_done, malaxage_done
INVARIANT
prod ∈ (0..ma)(0..mb)*(0..n)*(0..ts) ∧ init1_done ∈ BOOL ∧ loc1_produit ∈ LIEUX ∧*
go_VA ∈ BOOL ∧ go_VB ∈ BOOL ∧ go_brique ∈ BOOL ∧ go_malaxer ∈ BOOL ∧
qr_a ∈ 0..ma ∧ qr_b ∈ 0..mb ∧ nbr ∈ 0..n ∧ tr ∈ 0..tm ∧

VA_done $\in$ *BOOL* $\wedge$ *VB_done* $\in$ *BOOL* $\wedge$ *brique_done* $\in$ *BOOL* $\wedge$
malaxage_done $\in$ *BOOL* $\wedge$
/ Invariants de collage */*
(go_malaxer = TRUE $\Rightarrow$ *go_fabriquer = TRUE)*

INITIALISATION

prod := (0$\mapsto$*0*$\mapsto$*0*$\mapsto$*0) || step := commencer || loc1_produit := no_where ||*
go_start := FALSE || go_malaxer := FALSE || go_livrer := FALSE ||
go_VA := FALSE || go_VB := FALSE || go_brique := FALSE ||
init_done := FALSE || fabrication_done := FALSE || livraison_done := FALSE ||
qr_a := 0 || qr_b := 0 || nbr := 0 || tr := 0 ||
init1_done := FALSE || livraison_done := FALSE ||
VA_done := FALSE || VB_done := FALSE ||
brique_done := FALSE || malaxage_done := FALSE

EVENTS

*/** **Sous système de dosage** **/*

Init_OP = SELECT
go_start = TRUE
THEN
prod := (0$\mapsto$*0*$\mapsto$*0*$\mapsto$*0) || loc1_produit := no_where ||*
init1_done := TRUE || go_start := FALSE
END ;

Obtenir_VA_OP = SELECT
go_VA = TRUE
THEN
prod := (qr_a$\mapsto$*0*$\mapsto$*0*$\mapsto$*0) ||*
loc1_produit := in || VA_done := TRUE || go_VA := FALSE
END ;

Obtenir_VB_OP = SELECT
go_VB=TRUE
THEN
prod := (qr_a$\mapsto$*qr_b*$\mapsto$*0*$\mapsto$*0) ||*
VB_done := TRUE || go_VB := FALSE
END ;

/* Sous système de convoyeur */

Obtenir_briques_OP = SELECT
go_brique=TRUE
THEN
prod := (qr_a$\mapsto$*qr_b*$\mapsto$*nbr*$\mapsto$*0) ||*
brique_done := TRUE || go_brique := FALSE
END ;

/* Sous système de malaxage*/

Fabriquer_OP = SELECT
go_malaxer = TRUE
THEN
prod := (qr_a$\mapsto$*qr_b*$\mapsto$*nbr*$\mapsto$*tr) ||*
malaxage_done := TRUE || go_malaxer := FALSE
END ;

/* Sous système de livraison */

Livrer_OP = SELECT
go_livrer = TRUE
THEN
loc1_produit := out || livraison_done := TRUE || go_livrer := FALSE
END
END

2.2 Modèle du contrôleur

REFINEMENT C1
REFINES C0
SETS
ETAPES = {commencer, ouvrir_VA, verser_VA, ouvrir_VB, verser_VB, ouvrir_VC, transvaser, marche_moteur_conv, verser_brique, marche_moteur_mal, malaxer, fabriquer, marche_moteur_liv, marche_moteur_bas,basculer,livrer,terminer}
VARIABLES
step, loc1_produit, hand1
/ Variables de commande */*
go_start, go_livrer, go_VA, go_VB, go_brique, go_malaxer,
/ Variables d'échange de valeur */*
qr_a, qr_b, nbr, tr,
/ Variables d'acquittement */*
init1_done, livraison_done, VA_done, VB_done, brique_done, malaxage_done
INVARIANT
step ∈ ETAPES ∧ init1_done ∈ BOOL ∧ loc1_produit ∈ LIEUX ∧
go_VA ∈ BOOL ∧ go_VB ∈ BOOL ∧ go_brique ∈ BOOL ∧ go_malaxer ∈ BOOL ∧
qr_a ∈ 0..ma ∧ qr_b ∈ 0..mb ∧ nbr ∈ 0..n ∧ tr ∈ 0..tm ∧ hand1 ∈ NATURAL
VA_done ∈ BOOL ∧ VB_done ∈ BOOL ∧ brique_done ∈ BOOL ∧
malaxage_done ∈ BOOL ∧
/ invariants de collage */*
(step = commencer ⇒ en_marche = FALSE) ∧
(en_marche = FALSE ⇒ step = commencer) ∧
(step ∈ {malaxer,livrer} ⇒ en_marche = TRUE) ∧
(loc_produit = no_where ⇒ loc1_produit ∈ {no_where,in}) ∧
(loc1_produit = no_where ⇒ loc_produit = no_where) ∧
(loc_produit = in ⇒ loc1_produit = in) ∧
(loc1_produit = in ⇒ loc_produit ∈ {no_where,in}) ∧
(loc_produit = out ⇒ loc1_produit = out) ∧
(loc1_produit = out ⇒ loc_produit = out) ∧
(brique_done = TRUE ⇒ init_done = TRUE) ∧
(malaxage_done = TRUE ⇒ fabrication_done = TRUE) ∧
(go_malaxer = FALSE ⇒ go_fabriquer = FALSE) ∧
(step ∈ {verser_VA,verser_VB,verser_brique} ⇒ loc_produit = no_where)
(hand1 ∈ {1,2,3} ⇒ hand = 1) ∧ (hand1 = 4 ⇒ hand = 2)
INITIALISATION
step := commencer || loc1_produit := no_where || hand1 := 1
go_start := FALSE || go_malaxer := FALSE || go_livrer := FALSE ||

go_VA := FALSE || go_VB := FALSE || go_brique := FALSE ||
qr_a := 0 || qr_b := 0 || nbr := 0 || tr := 0 ||
init1_done := FALSE || livraison_done := FALSE ||
VA_done := FALSE || VB_done := FALSE || brique_done := FALSE ||
malaxage_done := FALSE

EVENTS

/ ***Sous système de dosage*** */*

Start_C = SELECT
step = commencer ∧ go_start = FALSE ∧ hand1 = 1
THEN
step := verser_VA || go_start := TRUE
END ;

Obtenir_VA_C = SELECT
step = verser_VA ∧ go_VA = FALSE ∧ init1_done = TRUE ∧
loc1_produit = no_where ∧ hand1 = 1
THEN
go_VA := TRUE || qr_a := ma || init1_done := FALSE || step := verser_VB
END ;

Obtenir_VB_C = SELECT
step = verser_VB ∧ go_VB = FALSE ∧ VA_done = TRUE ∧
loc1_produit = in ∧ hand1 = 1
THEN
qr_b := mb || VA_done := FALSE || go_VB := TRUE
END ;

Terminer_C1 = SELECT
hand1 = 1 ∧ VB_done = TRUE
THEN
VB_done := FALSE || hand1 := 2 || step := verser_brique
END ;

/ ***Sous système de convoyeur*** */*

Obtenir_briques_C = SELECT
step = verser_brique ∧ go_brique = FALSE ∧
loc1_produit = in ∧ hand1 = 2
THEN
go_brique := TRUE || nbr := n **END ;**

Terminer_C2 = SELECT
hand1 = 2 ∧ brique_done = TRUE
THEN
brique_done := FALSE || hand1 := 3 || step := malaxer
END ;

/ ***Sous système de malaxage*** */*

Fabriquer_C = SELECT
step = malaxer ∧ go_malaxer = FALSE ∧ hand1 = 3 loc1_produit = in
THEN
go_malaxer := TRUE || tr := tm **END ;**

Terminer_C = SELECT
hand1 = 3 ∧ malaxage_done = TRUE
THEN

malaxage_done := FALSE || hand1 := 4 || step := livrer
END ;
/ **Sous système de livraison** */*
Livrer_C = SELECT
step = livrer ∧ go_livrer = FALSE ∧ hand1 = 4 ∧ loc1_produit = in
THEN
go_livrer := TRUE || **END ;**
Finir = SELECT
livraison_done = TRUE ∧ loc1_produit = out ∧ hand1 = 4
THEN
step := commencer || livraison_done := FALSE
END
END

3 Deuxième raffinement

3.1 Modèle du contrôlé

REFINEMENT OP2
REFINES OP1
SETS
POSITION_MIX = {verticale, horizontale}
VARIABLES
prod, loc_prod, position,
/ Variables de commande */*
go_start, go_livrer, go_VA, go_VB, go_brique, go_malaxer, go_transvaser, go_basculer,
/ Variables d'échange de valeur */*
qr_a, qr_b, nbr, tr,
/ Variables d'acquittement */*
init1_done, livraison_done, VA_done, VB1_done, brique_done,
malaxage1_done, transvaser_done, basculer_done
INVARIANT
loc_prod ∈ LIEUX ∧ VB1_done ∈ BOOL ∧ malaxage1_done ∈ BOOL
transvaser_done ∈ BOOL ∧ go_transvaser ∈ BOOL
basculer_done ∈ BOOL ∧ go_basculer ∈ BOOL
INITIALISATION
prod := (0↦0↦0↦0) || loc_prod := no_where || go_start := FALSE ||
go_malaxer := FALSE || go_livrer := FALSE || go_transvaser := FALSE ||
go_VA := FALSE || go_VB := FALSE || go_brique := FALSE || go_basculer := FALSE
qr_a := 0 || qr_b := 0 || nbr := 0 || tr := 0 || position := verticale ||
init1_done := FALSE || livraison_done := FALSE ||
VA_done := FALSE || VB1_done := FALSE || brique_done := FALSE ||
malaxage1_done := FALSE || transvaser_done := FALSE || basculer_done := FALSE
EVENTS
/ **Sous système de dosage** */*
Init_OP = SELECT

go_start = TRUE
THEN
prod := (0↦0↦0↦0) || loc_prod := no_where || position := verticale ||
init1_done := TRUE || go_start := FALSE
END ;
Obtenir_VA_OP = SELECT
go_VA = TRUE
THEN
prod := (qr_a↦0↦0↦0) || loc_prod := cuve || VA_done := TRUE || go_VA := FALSE
END ;
Obtenir_VB_OP = SELECT
go_VB=TRUE
THEN
prod := (qr_a↦qr_b↦0↦0) || VB1_done := TRUE || go_VB := FALSE
END ;
Transvaser_OP = SELECT
go_transvaser = TRUE
THEN
go_transvaser := FALSE || transvaser_done := TRUE || loc_prod := mixeur
END ;
/* Sous système de convoyeur */
Obtenir_briques_OP = SELECT
go_brique=TRUE
THEN
prod := (qr_a↦qr_b↦nbr↦0) || brique_done := TRUE || go_brique := FALSE
END ;
/* Sous système de malaxage*/
Fabriquer_OP = SELECT
go_malaxer = TRUE
THEN
prod := (qr_a↦qr_b↦nbr↦tr) || malaxage1_done := TRUE || go_malaxer := FALSE
END ;
/* Sous système de livraison */
Basculer_OP = SELECT *go_basculer = TRUE*
THEN
go_livrer := FALSE || loc_prod := out || livraison_done := TRUE
END ;
Livrer_OP = SELECT
go_livrer = TRUE
THEN
loc_prod := out || livraison_done := TRUE || go_livrer := FALSE
END
END

3.2 Modèle du contrôleur

REFINEMENT C2
REFINES C1
VARIABLES
step2, loc_prod, hand1,
/ Variables de commande */*
go_start, go_livrer, go_VA, go_VB, go_brique, go_malaxer, go_transvaser,
/ Variables d'échange de valeur */*
qr_a, qr_b, nbr, tr,
/ Variables d'acquittement */*
init1_done, livraison_done, VA_done, VB1_done, brique_done, malaxage_done,
transvaser_done
INVARIANT
$step2 \in ETAPES \wedge loc\_prod \in LIEUX \wedge VB1\_done \in BOOL \wedge$
$go\_transvaser \in BOOL \wedge transvaser\_done \in BOOL \wedge$
/ Invariants de collage */*
$(step2 \in \{commencer, verser\_VA, verser\_VB, malaxer, fabriquer, livrer\} \Rightarrow step = step2\ ) \wedge$
$(loc\_prod = mixeur \Rightarrow step2 \in \{malaxer, basculer, livrer\}) \wedge$
$(loc1\_produit = no\_where \Rightarrow loc\_prod = no\_where) \wedge$
$(loc\_prod = no\_where \Rightarrow loc1\_produit = no\_where) \wedge$
$(loc1\_produit = in \Rightarrow loc\_prod : \{cuve, mixeur\}) \wedge$
$(loc\_prod : \{cuve, mixeur\} \Rightarrow loc1\_produit = in) \wedge$
$(loc1\_produit = out \Rightarrow loc\_prod = out) \wedge$
$(loc\_prod = out \Rightarrow loc1\_produit = out) \wedge$
$(step2 = basculer \Rightarrow step = livrer) \wedge$
$(transvaser\_done = TRUE \Rightarrow VB\_done = TRUE\ ) \wedge$
$(malaxage1\_done = TRUE \Rightarrow malaxage\_done = TRUE) \wedge$
$(step2 = transvaser \Rightarrow step = verser\_VB) \wedge$
$(step2 = verser\_brique \Rightarrow step = verser\_brique) \wedge$
INITIALISATION
$step2 := commencer \parallel loc\_prod := no\_where \parallel go\_start := FALSE \parallel hand1 :=$
$go\_malaxer := FALSE \parallel go\_livrer := FALSE \parallel go\_transvaser := FALSE \parallel$
$go\_VA := FALSE \parallel go\_VB := FALSE \parallel go\_brique := FALSE \parallel$
$qr\_a := 0 \parallel qr\_b := 0 \parallel nbr := 0 \parallel tr := 0 \parallel hand1 := 1$
$init1\_done := FALSE \parallel livraison\_done := FALSE \parallel$
$VA\_done := FALSE \parallel VB1\_done := FALSE \parallel brique\_done := FALSE \parallel$
$malaxage\_done := FALSE \parallel transvaser\_done := FALSE$
EVENTS
/ **Sous système de dosage** */*
Start_C = SELECT
$step2 = commencer \wedge go\_start = FALSE \wedge hand1 = 1$
THEN
$step2 := verser\_VA \parallel go\_start := TRUE$
END ;
Obtenir_VA_C = SELECT
$step2 = verser\_VA \wedge go\_VA = FALSE \wedge init1\_done = TRUE \wedge$
$hand1 = 1 \wedge loc\_prod = no\_where$

THEN
go_VA := TRUE || qr_a := ma || init1_done := FALSE || step2 := verser_VB
END ;
Obtenir_VB_C = SELECT
step2 = verser_VB ∧ go_VB = FALSE ∧ VA_done = TRUE ∧
hand1 = 1 ∧ loc_prod = cuve
THEN
step2 := verser_brique || qr_b := mb || VA_done := FALSE || go_VB := TRUE
END ;
Transvaser_C = SELECT
step2 = transvaser ∧ go_transvaser = FALSE ∧
loc_prod = cuve ∧ VB1_done = TRUE ∧ hand1 = 1
THEN
VB1_done := FALSE || go_transvaser := TRUE
END ;
Terminer_C1 = SELECT
hand1 = 1 ∧ transvaser_done = TRUE ∧ step2 = transvaser
THEN
transvaser_done := FALSE || hand1 := 2 || step2 := verser_brique
END ;
/ **Sous système de convoyeur** */*
Obtenir_briques_C = SELECT
step2 = verser_brique ∧ go_brique = FALSE ∧ loc_prod = mixeur ∧ hand1 = 2
THEN
go_brique := TRUE || nbr := n
END ;
Terminer_C2 = SELECT
hand1 = 2 ∧ brique_done = TRUE ∧ step2 = verser_brique
THEN
brique_done := FALSE || hand1 := 3 || step2 := malaxer
END ;
/ **Sous système de malaxage** */*
Fabriquer_C = SELECT
step2 = malaxer ∧ go_malaxer = FALSE ∧
loc_prod = mixeur ∧ hand1 = 3
THEN
go_malaxer := TRUE || tr := tm
END ;
Terminer_C = SELECT
hand1 = 3 ∧ malaxage1_done = TRUE ∧ step2 = malaxer
THEN
malaxage1_done := FALSE || hand1 := 4 || step2 := basculer
END ;
/ **Sous système de livraison** */*
Basculer_C = SELECT
step2 = basculer ∧ loc_prod = mixeur ∧ go_basculer = FALSE ∧ hand1 = 4
THEN
go_basculer := TRUE || step2 := livrer

END ;
Livrer_C = SELECT
step2 = livrer ∧ go_livrer = FALSE ∧ basculer_done = TRUE ∧
loc_prod = mixeur ∧ hand1 = 4
THEN
go_livrer := TRUE || basculer_done := FALSE
END ;
Finir = SELECT
livraison_done = TRUE ∧ loc_prod = out ∧ hand1 = 4
THEN
step2 := commencer || livraison_done := FALSE
END
END

4 Troisième raffinement

4.1 Modèle du contrôlé

REFINEMENT OP3
REFINES OP2
SETS
VAL = {close, open} ;
MOTEUR = {on, off}
VARIABLES
va, vb, vc, angle, contenu, moteur_convoyer,
moteur_malaxer, moteur_tourner, moteur_livrer
/ Variables de commande */*
go_start, go_ouvrirA, go_ouvrirB, go_ouvrirC , go_moteurCN, go_moteurML,
go_moteurBAS, go_moteurTR, go_livrer, go_VA, go_VB, go_brique,
go_malaxer, go_transvaser,
/ Variables d'acquittement */*
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint,
time_atteint, fabric_done, convoyeur_done, dosage_done,
produit_livre, angle_atteint, livraison_done,
INVARIANT
contenu ∈ LIEUX ⇸ (0..ma)(0..mb)*(0..n)*(0..tm) ∧*
moteur_convoyer ∈ MOTEUR ∧ moteur_malaxer ∈ MOTEUR ∧
moteur_tourner ∈ MOTEUR ∧ moteur_livrer ∈ MOTEUR ∧
va ∈ VAL ∧ vb ∈ VAL ∧ vc ∈ VAL ∧ angle ∈ 0..90 ∧
W0_atteint ∈ BOOL ∧ WA_atteint ∈ BOOL ∧ WB_atteint ∈ BOOL ∧
WC_atteint ∈ BOOL ∧ n_atteint ∈ BOOL ∧ time_atteint ∈ BOOL ∧
angle_atteint ∈ BOOL ∧ convoyeur_done ∈ BOOL ∧ dosage_done ∈ BOOL ∧
fabric_done ∈ BOOL ∧ produit_livre ∈ BOOL ∧ go_ouvrirA : BOOL ∧
go_ouvrirB ∈ BOOL ∧ go_ouvrirC ∈ BOOL ∧ go_moteurCN ∈ BOOL ∧
go_moteurML ∈ BOOL ∧ go_moteurBAS ∈ BOOL ∧ go_moteurTR ∈ BOOL ∧
INITIALISATION

va := close || vb := close || vc := close ||
*contenu := LIEUX * {(0↦0↦0↦0)} || loc_prod := no_where ||*
moteur_convoyer := off || moteur_malaxer := off || moteur_tourner := off ||
moteur_livrer := off || angle := 90 || W0_atteint := FALSE //
WA_atteint := FALSE || WB_atteint := FALSE || WC_atteint := FALSE ||
n_atteint := FALSE || time_atteint := FALSE || angle_atteint := FALSE ||
dosage_done := FALSE || convoyeur_done := FALSE || produit_livre := FALSE ||
fabric_done := FALSE || livraison_done := FALSE || go_start := FALSE ||
go_ouvrirA := FALSE || go_ouvrirB := FALSE || go_ouvrirC := FALSE ||
go_moteurCN := FALSE || go_moteurML := FALSE || go_transvaser := FALSE ||
go_moteurBAS := FALSE || go_moteurTR := FALSE || go_VA := FALSE ||
go_VB := FALSE || go_brique := FALSE || go_malaxer := FALSE

EVENTS

/ **Sous système de dosage** */*

Start_OP = **SELECT**
go_start = TRUE
THEN
*contenu := LIEUX *(0↦0↦0↦0) || go_start := FALSE || W0_atteint := TRUE ||*
va := close || vb := close || vc := close ||
moteur_convoyer := off || moteur_malaxer := off || moteur_tourner := off ||
moteur_livrer := off || angle := 90
END ;

Ouvrir_VA_OP = **SELECT**
go_ouvrirA = TRUE
THEN
va := open || go_ouvrirA := FALSE
END ;

Remplissage_cuveAC_OP = **ANY x WHERE**
x : 0..ma-1 ∧ contenu(cuve) = (x↦0↦0↦0)
THEN
contenu(cuve) := (x+1↦0↦0↦0)
END ;

Detect_WA = SELECT
contenu(cuve) = (ma↦0↦0↦0)
THEN
WA_atteint := TRUE
END ;

Obtenir_VA_OP = **SELECT**
go_VA=TRUE
THEN
va := close || go_VA := FALSE
END ;

Ouvrir_VB_OP = **SELECT**
go_ouvrirB = TRUE
THEN
vb := open || go_ouvrirB := FALSE
END ;

Remplissage_cuveBC_OP = **ANY y WHERE**

$y : 0..mb\text{-}1 \wedge contenu(cuve) = (ma \mapsto y \mapsto 0 \mapsto 0)$
THEN
$contenu(cuve) := (ma \mapsto y+1 \mapsto 0 \mapsto 0)$
END ;

Detect_WB = SELECT
$contenu(cuve) = (ma \mapsto mb \mapsto 0 \mapsto 0)$
THEN
$WB\_atteint := TRUE$
END ;

Obtenir_VB_OP = SELECT
$go\_VB=TRUE$
THEN
$vb := close \parallel go\_VB := FALSE$
END ;

Init_transvaser_OP = SELECT
$go\_ouvrirC = TRUE$
THEN
$vc := open \parallel go\_ouvrirC := FALSE$
END ;

Laisser_transvaser_OP = ANY x,y,u,v,a,b,c,d WHERE
$x \in 0..ma \wedge y \in 0..mb \wedge u \in 0..ma \wedge v \in 0..mb \wedge$
$x+y \in 1..ma+mb \wedge u+v \in 0..ma+mb\text{-}1 \wedge x+y+u+v = ma+mb \wedge x+u = ma \wedge$
$y+v = mb \wedge a \in 0..ma \wedge b \in 0..mb \wedge c \in 0..ma \wedge$
$d \in 0..mb \wedge a+c=ma \wedge b+d=mb \wedge a+b \in 1..ma+mb \wedge c+d \in 0..ma+mb\text{-}1 \wedge$
$a+b+c+d = ma+mb \wedge contenu(cuve) = (x \mapsto y \mapsto 0 \mapsto 0) \wedge$
$contenu(mixeur) = (u \mapsto v \mapsto 0 \mapsto 0) \wedge$
THEN
$contenu \in (contenu \in LIEUX \nrightarrow (0..ma)*(0..mb)*(0..n)*(0..tm) \wedge$
$contenu(cuve)=(a \mapsto b \mapsto 0 \mapsto 0) \wedge a+b=x+y\text{-}1 \wedge contenu(mixeur)=(c \mapsto d \mapsto 0 \mapsto 0) \wedge c+d=u+v+1)$
END ;

Detect_WC = SELECT
$contenu(mixeur) = (ma \mapsto mb \mapsto 0 \mapsto 0)$
THEN
$WC\_atteint := TRUE$
END ;

Transvaser_OP = SELECT
$go\_transvaser = TRUE$
THEN
$vc := close \parallel go\_transvaser := FALSE \parallel dosage\_done := TRUE$
END ;

/* ***Sous système de convoyeur*** */

***Init_convoyer_OP* = SELECT**
$go\_moteurCN = TRUE$
THEN
$moteur\_convoyer := on \parallel go\_moteurCN := FALSE$
END ;

Laisser_tomber_briques_OP = ANY nb WHERE
$nb \in 0..n\text{-}1 \wedge contenu(mixeur) = (ma \mapsto mb \mapsto nb \mapsto 0)$

THEN
contenu(mixeur) := (ma↦mb↦nb+1↦0)
END ;
Detect_briques = SELECT
contenu(mixeur) = (ma↦mb↦n↦0)
THEN
n_atteint := TRUE
END ;
Obtenir_briques_OP = SELECT
go_brique = TRUE
THEN
go_brique := FALSE ||
moteur_convoyer := off || convoyeur_done := TRUE
END ;
/ **Sous système de malaxage** */*
***Init_malaxer_OP* = SELECT**
go_moteur_ML = TRUE
THEN
moteur_malaxer := on || go_moteur_ML := FALSE
END ;
Laisser_malaxer_OP = ANY tt WHERE
tt : 0..tm-1 ∧ contenu(mixeur) = (ma↦mb↦n↦tt)
THEN
contenu(mixeur) := (ma↦mb↦n↦tt+1)
END ;
Detect_temps = SELECT
contenu(mixeur) = (ma↦mb↦n↦tm)
THEN
time_atteint := TRUE
END ;
Fabriquer_OP = SELECT
go_malaxer = TRUE
THEN
moteur_malaxer := off || go_malaxer := FALSE || fabric_done := TRUE
END ;
/ **Sous système de livraison** */*
***Init_basculer_OP* = SELECT**
go_moteurBAS = TRUE
THEN
moteur_tourner := on || go_moteurBAS := FALSE
END ;
Laisser_basculer = SELECT
moteur_tourner = on ∧ angle : 1..90
THEN
angle := angle -1
END ;
Detect_angle = SELECT
angle=0

THEN
angle_atteint := TRUE
END ;
Basculer_OP = SELECT
go_basculer = TRUE
THEN
moteur_tourner := off || go_basculer := FALSE
END ;
Init_livrer_OP = SELECT
go_moteur_TR = TRUE
THEN
moteur_tourner := on || go_moteur_TR := FALSE
END ;
Laisser_livrer_OP = SELECT
moteur_livrer = on
THEN
contenu ∈ (contenu ∈ LIEUX ⇸ (0..ma)(0..mb)*(0..n)*(0..tm) ∧*
contenu(out)= (ma↦mb↦n↦tm) ∧ contenu(mixeur) = (0↦0↦0↦0))
END ;
Detect_produit = SELECT
contenu(out) = (ma↦mb↦n↦tm)
THEN
produit_livre := TRUE
END ;
Livrer_OP = SELECT
go_livrer = TRUE ∧ moteur_livrer = on
THEN
moteur_livrer := off || go_livrer := FALSE || livraison_done := TRUE
END
END

4.2 Modèle du contrôleur

REFINEMENT C3
REFINES C2
VARIABLES
step3, hand1,
/ Variables de commande */*
go_start, go_ouvrirA, go_ouvrirB, go_ouvrirC , go_moteurCN, go_moteurML,
go_livrer, go_VA, go_VB, go_brique, go_malaxer, go_transvaser, go_basculer
/ Variables d'acquittement */*
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint,
time_atteint, fabric_done, convoyeur_done, dosage_done,
produit_livre, angle_atteint, livraison_done
INVARIANT
step3 ∈ ETAPES ∧ W0_atteint ∈ BOOL ∧ WA_atteint ∈ BOOL ∧
WB_atteint ∈ BOOL ∧ WC_atteint ∈ BOOL ∧ n_atteint ∈ BOOL ∧

time_atteint $\in$ *BOOL* $\wedge$ *angle_atteint* $\in$ *BOOL* $\wedge$
dosage_done $\in$ *BOOL* $\wedge$ *convoyeur_done* $\in$ *BOOL* $\wedge$
fabric_done $\in$ *BOOL* $\wedge$ *produit_livre* $\in$ *BOOL* $\wedge$ *go_ouvrirA* $\in$ *BOOL* $\wedge$
go_ouvrirB $\in$ *BOOL* $\wedge$ *go_ouvrirC* $\in$ *BOOL* $\wedge$ *go_moteurCN* $\in$ *BOOL* $\wedge$
go_moteurML $\in$ *BOOL* $\wedge$ *go_moteurBAS* $\in$ *BOOL* $\wedge$ *go_moteurTR* $\in$ *BOOL* $\wedge$
/ Invariants de liaison */*
(step3 $\in$ *{commencer,fabriquer}* $\Rightarrow$ *step2 = step3)* $\wedge$
(step2 $\in$ *{commencer,fabriquer}* $\Rightarrow$ *step3 = step2)* $\wedge$
(step3 $\in$ *{ouvrir_VA,verser_VA}* $\Rightarrow$ *step2=verser_VA)* $\wedge$
(step2=verser_VA $\Rightarrow$ *step3* $\in$ *{ouvrir_VA,verser_VA})* $\wedge$
(step3 $\in$ *{ouvrir_VB,verser_VB}* $\Rightarrow$ *step2=verser_VB)* $\wedge$
(step2=verser_VB $\Rightarrow$ *step3* $\in$ *{ouvrir_VB,verser_VB})* $\wedge$
(step3 $\in$ *{ouvrir_VC,transvaser}* $\Rightarrow$ *step2=transvaser)* $\wedge$
(step2=transvaser $\Rightarrow$ *step3* $\in$ *{ouvrir_VC,transvaser})* $\wedge$
(step3 $\in$ *{marche_moteur_conv,verser_brique}* $\Rightarrow$ *step2=verser_brique)* $\wedge$
(step2=verser_brique $\Rightarrow$ *step3* $\in$ *{marche_moteur_conv,verser_brique})* $\wedge$
(step3 $\in$ *{marche_moteur_mal, malaxer}* $\Rightarrow$ *step2=malaxer)* $\wedge$
(step2=malaxer $\Rightarrow$ *step3* $\in$ *{marche_moteur_mal, malaxer})* $\wedge$
(step3 $\in$ *{marche_moteur_bas,basculer}* $\Rightarrow$ *step2=basculer)* $\wedge$
(step2=basculer $\Rightarrow$ *step3* $\in$ *{marche_moteur_bas,basculer}* $\wedge$
(step2=livrer $\Rightarrow$ *step3* $\in$ *{marche_moteur_liv,livrer})* $\wedge$
(step3 $\in$ *{marche_moteur_liv,livrer}* $\Rightarrow$ *step2=livrer)*
(step3 $\in$ *{ouvrir_VA, verser_VA,ouvrir_VB}* $\Rightarrow$ *loc_prod = no_where)* $\wedge$
(step3 = commencer $\Rightarrow$ *loc_prod* $\in$ *{no_where,out})* $\wedge$
(step3 $\in$ *{verser_brique,marche_moteur_mal,malaxer,marche_moteur_bas,basculer,*
marche_moteur_conv} $\Rightarrow$ *loc_prod = mixeur)* $\wedge$
(WA_atteint = init1_done) $\wedge$ *(WB_atteint = VA_done)* $\wedge$ *(WC_atteint = VB1_done)*
(WB_atteint = TRUE $\Rightarrow$ *loc_prod = cuve)* $\wedge$
(time_atteint = TRUE $\Rightarrow$ *loc_prod = mixeur)* $\wedge$
(n_atteint = TRUE $\Rightarrow$ *loc_prod = mixeur)* $\wedge$
(produit_livre = TRUE $\Rightarrow$ *loc_prod = mixeur)* $\wedge$
(WC_atteint = TRUE $\Rightarrow$ *loc_prod = cuve)* $\wedge$
(fabric_done = TRUE $\Rightarrow$ *malaxage1_done = TRUE)* $\wedge$
(convoyeur_done = TRUE $\Rightarrow$ *brique_done = TRUE)* $\wedge$
(dosage_done = TRUE $\Rightarrow$ *transvaser_done = TRUE)* $\wedge$

INITIALISATION

step3 := commencer || hand1 := 1 || W0_atteint := FALSE //
WA_atteint := FALSE || WB_atteint := FALSE || WC_atteint := FALSE ||
n_atteint := FALSE || time_atteint := FALSE || angle_atteint := FALSE ||
dosage_done := FALSE || convoyeur_done := FALSE || produit_livre := FALSE ||
fabric_done := FALSE || livraison_done := FALSE || go_start := FALSE ||
go_ouvrirA := FALSE || go_ouvrirB := FALSE || go_ouvrirC := FALSE ||
go_moteurCN := FALSE || go_moteurML := FALSE || go_transvaser := FALSE ||
go_moteurBAS := FALSE || go_moteurTR := FALSE || go_VA := FALSE ||
go_VB := FALSE || go_brique := FALSE || go_malaxer := FALSE
go_basculer := FALSE || go_livrer := FALSE

EVENTS

/ **Sous système de dosage** */*

Start_C = **SELECT**
step3 = commencer ∧ go_start = FALSE ∧ hand1 = 1
THEN
step3 := ouvrir_VA || go_start := TRUE
END ;
Ouvrir_VA_C = SELECT
step3 = ouvrir_VA ∧ W0_atteint = TRUE ∧ go_ouvrirA = FALSE ∧ hand1 = 1
THEN
step3 := verser_VA || go_ouvrirA := TRUE // W0_atteint := FALSE
END ;
Obtenir_VA_C = SELECT
step3 = verser_VA ∧ WA_atteint = TRUE ∧ go_VA = FALSE ∧ hand1 = 1
THEN
step3 := ouvrir_VB || go_VA := TRUE || WA_atteint := FALSE
END ;
Ouvrir_VB_C = SELECT
step3 = ouvrir_VB ∧ go_ouvrirB = FALSE ∧ hand1 = 1
THEN
go_ouvrirB := TRUE || step3 := verser_VB
END ;
Obtenir_VB_C = SELECT
step3 = verser_VB ∧ WB_atteint = TRUE ∧ go_VB = FALSE ∧ hand1 = 1
THEN
step3 := ouvrir_VC || go_VB := TRUE || WB_atteint := FALSE
END ;
Init_transvaser_C = SELECT
step3 = ouvrir_VC ∧ go_ouvrirC = FALSE ∧ hand1 = 1
THEN
go_ouvrirC := TRUE || step3 := transvaser
END ;
Transvaser_C = SELECT
step3 = transvaser ∧ WC_atteint = TRUE ∧ go_transvaser = FALSE ∧ hand1 = 1
THEN
go_transvaser := TRUE || WC_atteint := FALSE
END ;
Terminer_C1 = SELECT
hand1 = 1 ∧ dosage_done = TRUE ∧ step3 = transvaser
THEN
dosage_done := FALSE || hand1 := 2 || step3 := marche_moteur_conv
END ;
/ **Sous système de convoyeur** */*
Init_convoyeur_C = **SELECT**
step3 = marche_moteur_conv ∧ go_moteurCN = FALSE ∧ hand1 = 2
THEN
go_moteurCN := TRUE || step3 := verser_brique
END ;
Obtenir_briques_C = SELECT
step3 = verser_brique ∧ n_atteint = TRUE ∧ go_brique = FALSE ∧ hand1 = 2

THEN
go_brique := TRUE || n_atteint := FALSE
END ;
Terminer_C2 = SELECT
hand1 = 2 ∧ convoyeur_done = TRUE ∧ step3 = verser_brique
THEN
convoyeur_done := FALSE || hand1 := 3 || step3 := marche_moteur_mal
END ;
/ **Sous système de malaxage** */*
***Init_malaxer_C* = SELECT**
step3 = marche_moteur_mal ∧ go_moteurML = FALSE ∧ hand1 = 3
THEN
go_moteurML := TRUE || step3 := malaxer
END ;
Fabriquer_C = SELECT
step3 = malaxer ∧ time_atteint = TRUE ∧ go_malaxer = FALSE ∧ hand1 = 3
THEN
go_malaxer := TRUE || time_atteint := FALSE
END ;
Terminer_C = SELECT
hand1 = 3 ∧ fabric_done = TRUE ∧ step3 = malaxer
THEN
fabric_done := FALSE || hand1 := 4 || step3 := marche_moteur_bas
END ;
/ **Sous système de livraison** */*
***Init_basculer_C* = SELECT**
step3 = marche_moteur_bas ∧ go_moteurBAS = FALSE ∧ hand1 = 4
THEN
go_moteurTR := TRUE || step3 := basculer
END ;
Basculer_C = SELECT
step3 = basculer ∧ angle_atteint = TRUE ∧ go_basculer = FALSE ∧ hand1 = 4
THEN
go_basculer := TRUE || step3 := marche_moteur_liv || angle_atteint := FALSE
END ;
Init_livrer_C = SELECT
step3 = marche_moteur_liv ∧ go_moteurTR = FALSE ∧ hand1 = 4
THEN
go_moteurTR := TRUE || step3 := livrer
END ;
Livrer_C =
SELECT step3 = livrer ∧ go_livrer = FALSE ∧ produit_livre = TRUE
THEN
go_livrer := TRUE || produit_livre := FALSE
END ;
Finir = SELECT
livraison_done = TRUE ∧ step3 = terminer ∧ hand1 = 4
THEN

step3 := commencer || livraison_done := FALSE
END
END

5 Quatrième raffinement

5.1 Modèle du contrôlé

REFINEMENT OP4
REFINES OP3
CONSTANTS *d_max*
PROPERTIES *d_max ∈ NATURAL1*
VARIABLES
.....d, detect, go_avance_brique
INVARIANT
detect ∈ BOOL ∧ d ∈ 0..d_max ∧ go_avance_brique ∈ BOOL
INITIALISATION
.....d := 0 || detect := FALSE || go_avance_brique := FALSE
EVENTS
.....
Avancer_brique = SELECT
moteur_convoyeur = on ∧ d < d_max ∧ detect = FALSE
THEN d := d+1
END ;
Detect_brique = SELECT
moteur_convoyeur = on ∧ d = d_max ∧ detect = FALSE
THEN
detect := TRUE || d := 0
END ;
Laisser_tomber_briques = ANY *nb* WHERE
nb : 0..n-1 ∧ moteur_convoyer = on ∧ detect = TRUE
contenu(mixeur) = (ma↦mb↦nb↦0)
THEN
contenu(mixeur) := (ma↦mb↦nb+1↦0) || detect := FALSE
END ;
.....
END

Annexe G
Les modules TLA$^+$ du système de production manufacturière

Nous présentons dans cet annexe les modules TLA$^+$ au niveau abstrait ainsi qu'aux niveaux des quatre raffinements correspondants au système de production manufacturière.

1 Module abstrait

MODULE mix

```
EXTENDS Naturals, TLC
CONSTANTS ma, mb, n, tm, ts, no_where, in, out, mixeur, cuve
VARIABLES produit, en_marche, loc_produit
ASSUME    ma ∈ Nat
          ∧  ma ≠ 0
          ∧  mb ∈ Nat
          ∧  mb ≠ 0
          ∧  n ∈  Nat
          ∧  n ≠ 0
          ∧  tm ∈ Nat
          ∧  tm ≠ 0
          ∧  ts  ∈ (0 . . tm)
LIEUX ≜ {no_where, in, out, mixeur, cuve}
```

```
TypeInvariant ≜  ∧ produit ∈ (0 . . ma) × (0 . . mb) × (0 . . n) × (0 . . tm)
                 ∧ en_marche ∈ BOOLEAN
                 ∧ loc_produit ∈ LIEUX
```

```
Init  ≜  ∧ produit = ⟨0, 0, 0, 0⟩
         ∧ en_marche = FALSE
         ∧ loc_produit = no_where
Start ≜
         ∧ en_marche = FALSE
         ∧ en_marche′ = TRUE
         ∧ produit′ = ⟨0, 0, 0, 0⟩
         ∧ loc_produit′ = no_where
         ∧ UNCHANGED ⟨⟩
Fabriquer ≜  ∧ en_marche = TRUE
             ∧ produit = ⟨0, 0, 0, 0⟩
             ∧ loc_produit = no_where
             ∧ produit′ = ⟨ma, mb, n, tm⟩
```

$\wedge$ loc_produit$'$ = in
$\wedge$ UNCHANGED ⟨en_marche⟩

Livrer $\triangleq$ $\wedge$ loc_produit = in
$\wedge$ loc_produit$'$ = out
$\wedge$ UNCHANGED ⟨produit, en_marche⟩

Finir $\triangleq$ $\wedge$ loc_produit = out
$\wedge$ en_marche = TRUE
$\wedge$ en_marche$'$ = FALSE
$\wedge$ UNCHANGED ⟨produit, loc_produit⟩

Next $\triangleq$ $\vee$ Start
$\vee$ Fabriquer
$\vee$ Livrer
$\vee$ Finir

---

trvars $\triangleq$ ⟨produit, en_marche, loc_produit⟩

TrFairness $\triangleq$ $\mathrm{WF}_{\mathrm{trvars}}$(Fabriquer) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Livrer)

Spec $\triangleq$ Init $\wedge$ $\Box[\mathrm{Next}]_{\mathrm{trvars}}$ $\wedge$ TrFairness

Liveness $\triangleq$ $\wedge$ en_marche = TRUE $\wedge$ produit = ⟨0, 0, 0, 0⟩ $\rightsquigarrow$ loc_produit = out $\wedge$ produit = ⟨ma, mb, n, tm⟩

THEOREM Spec $\Longrightarrow$ $\Box$Init

THEOREM Spec $\Longrightarrow$ $\Box$TypeInvariant

THEOREM Spec $\Longrightarrow$ Liveness

2 Module du premier raffinement

MODULE mix1

EXTENDS mix, Naturals, TLC

CONSTANTS commencer, ouvrir_VA, verser_VA, ouvrir_VB, verser_VB,
ouvrir_VC, transvaser, verser_brique, malaxer,
fabriquer, livrer, basculer, terminer,
marche_moteur_conv, marche_moteur_mal,
marche_moteur_liv, marche_moteur_bas

VARIABLES prod, step, loc1_produit

ETAPES $\triangleq$ {commencer, ouvrir_VA, verser_VA, ouvrir_VB, verser_VB,
ouvrir_VC, transvaser, verser_brique, malaxer,
fabriquer, livrer, basculer, terminer,
marche_moteur_conv, marche_moteur_mal,
marche_moteur_liv, marche_moteur_bas}

TypeInvariant1 $\triangleq$ $\wedge$ TypeInvariant
$\wedge$ prod $\in$ (0 . . ma) $\times$ (0 . . mb) $\times$ (0 . . n) $\times$ (0 . . tm)
$\wedge$ step $\in$ ETAPES
$\wedge$ loc1_produit $\in$ LIEUX

Init1 $\triangleq$ $\wedge$ Init
$\wedge$ prod = ⟨0, 0, 0, 0⟩
$\wedge$ step = commencer
$\wedge$ loc1_produit = no_where

Start1 $\triangleq$ $\wedge$ Start
$\wedge$ step = commencer
$\wedge$ step$'$ = verser_VA
$\wedge$ prod$'$ = ⟨0, 0, 0, 0⟩

$\land$ loc1_produit$'$ = no_where

Obtenir_VA $\triangleq$ $\land$ step = verser_VA
$\land$ loc1_produit = no_where
$\land$ prod$'$ = ⟨ma, 0, 0, 0⟩
$\land$ step$'$ = verser_VB
$\land$ loc1_produit$'$ = in
$\land$ UNCHANGED ⟨loc_produit, en_marche, produit⟩

Obtenir_VB $\triangleq$ $\land$ step = verser_VB
$\land$ loc1_produit = in
$\land$ prod$'$ = ⟨ma, mb, 0, 0⟩
$\land$ step$'$ = verser_brique
$\land$ UNCHANGED ⟨loc1_produit, loc_produit, en_marche, produit⟩

Obtenir_briques $\triangleq$ $\land$ step = verser_brique
$\land$ loc1_produit = in
$\land$ prod$'$ = ⟨ma, mb, n, 0⟩
$\land$ step$'$ = malaxer
$\land$ UNCHANGED ⟨loc1_produit, loc_produit, en_marche, produit⟩

Fabriquer1 $\triangleq$ $\land$ Fabriquer
$\land$ step = malaxer
$\land$ loc1_produit = in
$\land$ prod$'$ = ⟨ma, mb, n, tm⟩
$\land$ step$'$ = livrer
$\land$ UNCHANGED ⟨loc1_produit, en_marche⟩

Livrer1 $\triangleq$ $\land$ Livrer
$\land$ step = livrer
$\land$ loc1_produit = in
$\land$ loc1_produit$'$ = out
$\land$ UNCHANGED ⟨prod, step⟩

Finir1 $\triangleq$ $\land$ Finir
$\land$ loc1_produit = out
$\land$ step$'$ = commencer
$\land$ UNCHANGED ⟨prod, loc1_produit⟩

Next1 $\triangleq$ $\lor$ Start1
$\lor$ Obtenir_VA
$\lor$ Obtenir_VB
$\lor$ Obtenir_briques
$\lor$ Fabriquer1
$\lor$ Livrer1
$\lor$ Finir1

---

trvars1 $\triangleq$ ⟨prod, step, loc1_produit⟩

TrFairness1 $\triangleq$ WF_{trvars}(Obtenir_VA)
$\land$ WF_{trvars}(Obtenir_VB) $\land$ WF_{trvars}(Obtenir_briques)
$\land$ WF_{trvars}(Fabriquer1) $\land$ WF_{trvars}(Livrer1)

Spec1 $\triangleq$ Init1 $\land$ $\Box[\text{Next1}]_{trvars1}$ $\land$ TrFairness1

Invariant1 $\triangleq$ $\land$ en_marche = FALSE $\implies$ step = commencer
$\land$ step = commencer $\implies$ en_marche = FALSE
$\land$ step $\in$ {malaxer, livrer} $\implies$ en_marche = TRUE
$\land$ loc_produit = no_where $\implies$ loc1_produit $\in$ {no_where, in}
$\land$ loc1_produit = no_where $\implies$ loc_produit = no_where
$\land$ loc_produit = in $\implies$ loc1_produit = in
$\land$ loc1_produit = in $\implies$ loc_produit $\in$ {no_where, in}
$\land$ loc_produit = out $\implies$ loc1_produit = out
$\land$ loc1_produit = out $\implies$ loc_produit = out
$\land$ step $\in$ {verser_VA, verser_VB, verser_brique, malaxer} $\implies$ loc_produit = no_where
$\land$ step = livrer $\land$ loc1_produit = in $\implies$ loc_produit = in

```
Liveness1 ≜  ∧ step = commencer  ∧ prod = ⟨0, 0, 0, 0⟩ ⇝ loc1_produit = out ∧ prod = ⟨ma, mb, n, tm⟩
```

```
THEOREM  Spec1 ⟹ Spec
THEOREM  Spec1 ⟹ □Init1
THEOREM  Spec1 ⟹ □TypeInvariant1
THEOREM  Spec1 ⟹ □Invariant1
THEOREM  Spec1 ⟹ Liveness1
```

3 Module du deuxième raffinement

```
MODULE mix2
EXTENDS mix1, Naturals, TLC
CONSTANTS verticale, horizontale
VARIABLES  loc_prod, step2, position
POSITION_MIX ≜ {verticale, horizontale}
```

```
TypeInvariant2 ≜  ∧ TypeInvariant1
                  ∧ loc_prod ∈ LIEUX
                  ∧ step2 ∈ ETAPES
                  ∧ position ∈ POSITION_MIX
```

```
Init2  ≜  ∧ Init1
          ∧ loc_prod = no_where
          ∧ step2 = commencer
          ∧ position = verticale
Start2 ≜  ∧ Start1
          ∧ step2 = commencer
          ∧ step2′ = verser_VA
          ∧ loc_prod′ = no_where
          ∧ UNCHANGED ⟨position⟩
Obtenir_VA2 ≜  ∧ Obtenir_VA
               ∧ step2 = verser_VA
               ∧ loc_prod = no_where
               ∧ step2′ = verser_VB
               ∧ loc_prod′ = cuve
               ∧ UNCHANGED ⟨position⟩
Obtenir_VB2 ≜  ∧ Obtenir_VB
               ∧ step2 = verser_VB
               ∧ loc_prod = cuve
               ∧ step2′ = transvaser
               ∧ UNCHANGED ⟨loc_prod, position⟩
Transvaser ≜  ∧ step2 = transvaser
              ∧ loc_prod = cuve
              ∧ step2′ = verser_brique
              ∧ loc_prod′ = mixeur
              ∧ UNCHANGED ⟨loc_produit, en_marche, produit, step, prod, position, loc1_produit⟩
Obtenir_briques2 ≜  ∧ Obtenir_briques
                    ∧ step2 = verser_brique
                    ∧ loc_prod = mixeur
                    ∧ step2′ = malaxer
                    ∧ UNCHANGED ⟨loc_prod, position⟩
Fabriquer2  ≜  ∧ Fabriquer1
```

$\wedge$ step2 = malaxer
$\wedge$ loc_prod = mixeur
$\wedge$ step2′ = basculer
$\wedge$ UNCHANGED ⟨loc_prod, position⟩

Basculer $\triangleq$ $\wedge$ step2 = basculer
$\wedge$ position = verticale
$\wedge$ position′ = horizontale
$\wedge$ step2′ = livrer
$\wedge$ UNCHANGED ⟨loc_prod, prod, produit, loc1_produit, loc_produit, step, en_marche⟩

Livrer2 $\triangleq$ $\wedge$ Livrer1
$\wedge$ step2 = livrer
$\wedge$ loc_prod = mixeur
$\wedge$ loc_prod′ = out
$\wedge$ UNCHANGED ⟨prod, step, position, step2⟩

Finir2 $\triangleq$ $\wedge$ Finir1
$\wedge$ loc_prod = out
$\wedge$ step2′ = commencer
$\wedge$ UNCHANGED ⟨loc_prod, position⟩

Next2 $\triangleq$ $\vee$ Start2
$\vee$ Obtenir_VA2
$\vee$ Obtenir_VB2
$\vee$ Transvaser
$\vee$ Obtenir_briques2
$\vee$ Fabriquer2
$\vee$ Basculer
$\vee$ Livrer2
$\vee$ Finir2

---

trvars2 $\triangleq$ ⟨loc_prod, step2, position⟩

TrFairness2 $\triangleq$ $\mathrm{WF}_{\mathrm{trvars2}}$(Obtenir_VA2)
$\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Obtenir_VB2) $\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Transvaser)
$\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Obtenir_briques2)
$\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Fabriquer2) $\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Basculer)
$\wedge$ $\mathrm{WF}_{\mathrm{trvars2}}$(Livrer2)

Spec2 $\triangleq$ Init2 $\wedge$ $\Box[\mathrm{Next2}]_{\mathrm{trvars2}}$ $\wedge$ TrFairness2

Invariant2 $\triangleq$ $\wedge$ step2 $\in$ {commencer, verser_VA, verser_VB, malaxer, livrer} $\Longrightarrow$ step = step2
$\wedge$ loc_prod = mixeur $\Longrightarrow$ step2 $\in$ {verser_brique, malaxer, basculer, livrer}
$\wedge$ loc1_produit = no_where $\Longrightarrow$ loc_prod = no_where
$\wedge$ loc_prod = no_where $\Longrightarrow$ loc1_produit = no_where
$\wedge$ loc1_produit = in $\Longrightarrow$ loc_prod $\in$ {cuve, mixeur}
$\wedge$ loc_prod $\in$ {cuve, mixeur} $\Longrightarrow$ loc1_produit = in
$\wedge$ loc1_produit = out $\Longrightarrow$ loc_prod = out
$\wedge$ loc_prod = out $\Longrightarrow$ loc1_produit = out
$\wedge$ step2 = basculer $\Longrightarrow$ step = livrer
$\wedge$ step2 $\in$ {verser_brique, transvaser} $\Longrightarrow$ step = verser_brique
$\wedge$ step2 = basculer $\Longrightarrow$ step = livrer

Liveness2 $\triangleq$ step2 = commencer $\rightsquigarrow$ loc_prod = out $\wedge$ prod = ⟨ma, mb, n, tm⟩

THEOREM Spec2 $\Longrightarrow$ Spec1
THEOREM Spec2 $\Longrightarrow$ $\Box$Init2
THEOREM Spec2 $\Longrightarrow$ $\Box$TypeInvariant2
THEOREM Spec2 $\Longrightarrow$ $\Box$Invariant2
THEOREM Spec2 $\Longrightarrow$ Liveness2

4 Module du troisième raffinement

MODULE mix3

EXTENDS mix2, Naturals, TLC

CONSTANTS close, open, on, off

VARIABLES va, vb, vc, step3, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, contenu,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint

$VAL \triangleq \{close, open\}$

$MOTEUR \triangleq \{on, off\}$

$$
\begin{aligned}
TypeInvariant3 \triangleq\ & \wedge\ TypeInvariant2 \\
& \wedge\ contenu \in [LIEUX \to (0\,..\,ma) \times (0\,..\,mb) \times (0\,..\,n) \times (0\,..\,tm)] \\
& \wedge\ moteur\_convoyer \in MOTEUR \\
& \wedge\ moteur\_malaxer \in MOTEUR \\
& \wedge\ moteur\_tourner \in MOTEUR \\
& \wedge\ moteur\_livrer \in MOTEUR \\
& \wedge\ va \in VAL \\
& \wedge\ vb \in VAL \\
& \wedge\ vc \in VAL \\
& \wedge\ step3 \in ETAPES \\
& \wedge\ angle \in (0\,..\,90) \\
& \wedge\ W0\_atteint \in BOOLEAN \\
& \wedge\ WA\_atteint \in BOOLEAN \\
& \wedge\ WB\_atteint \in BOOLEAN \\
& \wedge\ WC\_atteint \in BOOLEAN \\
& \wedge\ n\_atteint \in BOOLEAN \\
& \wedge\ time\_atteint \in BOOLEAN \\
& \wedge\ angle\_atteint \in BOOLEAN \\
& \wedge\ produit\_livrer \in BOOLEAN
\end{aligned}
$$

$$
\begin{aligned}
Init3 \triangleq\ & \wedge\ Init2 \\
& \wedge\ contenu = [p \in LIEUX \mapsto \langle 0, 0, 0, 0\rangle] \\
& \wedge\ step3 = commencer \\
& \wedge\ moteur\_convoyer = off \\
& \wedge\ moteur\_malaxer = off \\
& \wedge\ moteur\_tourner = off \\
& \wedge\ moteur\_livrer = off \\
& \wedge\ angle = 90 \\
& \wedge\ va = close \\
& \wedge\ vb = close \\
& \wedge\ vc = close \\
& \wedge\ W0\_atteint = FALSE \\
& \wedge\ WA\_atteint = FALSE \\
& \wedge\ WB\_atteint = FALSE \\
& \wedge\ WC\_atteint = FALSE \\
& \wedge\ n\_atteint = FALSE \\
& \wedge\ time\_atteint = FALSE \\
& \wedge\ angle\_atteint = FALSE \\
& \wedge\ produit\_livrer = FALSE
\end{aligned}
$$

$$
\begin{aligned}
Start3 \triangleq\ & \wedge\ Start2 \\
& \wedge\ step3 = commencer \\
& \wedge\ step3' = ouvrir\_VA \\
& \wedge\ contenu' = [p \in LIEUX \mapsto \langle 0, 0, 0, 0\rangle] \\
& \wedge\ angle' = 90 \\
& \wedge\ va' = close \\
& \wedge\ vb' = close
\end{aligned}
$$

$\wedge$ vc′ = close
$\wedge$ moteur_convoyer′ = off
$\wedge$ moteur_malaxer′ = off
$\wedge$ moteur_tourner′ = off
$\wedge$ moteur_livrer′ = off
$\wedge$ W0_atteint′ = FALSE
$\wedge$ WA_atteint′ = FALSE
$\wedge$ WB_atteint′ = FALSE
$\wedge$ WC_atteint′ = FALSE
$\wedge$ n_atteint′ = FALSE
$\wedge$ time_atteint′ = FALSE
$\wedge$ angle_atteint′ = FALSE
$\wedge$ produit_livrer′ = FALSE

Ouvrir_VA $\triangleq$
$\wedge$ step3 = ouvrir_VA
$\wedge$ va = close
$\wedge$ vb = close
$\wedge$ vc = close
$\wedge$ contenu[cuve] = ⟨0, 0, 0, 0⟩
$\wedge$ va′ = open
$\wedge$ step3′ = verser_VA
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod,
vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer,
angle, position, contenu, W0_atteint, WA_atteint, WB_atteint,
WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Remplissage_cuveAC $\triangleq$
$\wedge$ va = open
$\wedge \exists$ x $\in$ 0 . . ma − 1 : $\wedge$ contenu[cuve] = ⟨x, 0, 0, 0⟩ $\wedge$ contenu′ = [contenu EXCEPT ![cuve] = ⟨x + 1, 0, 0, 0⟩]
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod,
va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer,
angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint,
n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Detect_WA $\triangleq$
$\wedge$ contenu[cuve] = ⟨ma, 0, 0, 0⟩
$\wedge$ WA_atteint′ = TRUE
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod,
va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer,
angle, position, step3, W0_atteint, contenu, WB_atteint, WC_atteint,
n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Obtenir_VA3 $\triangleq$
$\wedge$ Obtenir_VA2
$\wedge$ step3 = verser_VA
$\wedge$ va = open
$\wedge$ WA_atteint = TRUE
$\wedge$ step3′ = ouvrir_VB
$\wedge$ va′ = close
$\wedge$ UNCHANGED ⟨contenu, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner,
moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint ,
n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Ouvrir_VB $\triangleq$
$\wedge$ step3 = ouvrir_VB
$\wedge$ va = close
$\wedge$ vb = close
$\wedge$ vc = close
$\wedge$ contenu[cuve] = ⟨ma, 0, 0, 0⟩
$\wedge$ vb′ = open
$\wedge$ step3′ = verser_VB

```
        ∧  UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod,
                      va, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu,
                      W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Remplissage_cuveBC ≜
        ∧ vb = open
        ∧ ∃ y ∈ 0 .. mb − 1 : ∧ contenu[cuve] = ⟨ma, y, 0, 0⟩  ∧ contenu′ = [contenu EXCEPT ![cuve] = ⟨ma, y + 1, 0, 0⟩]
        ∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod,
                      va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
                      W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Detect_WB ≜
        ∧  contenu[cuve] = ⟨ma, mb, 0, 0⟩
        ∧  WB_atteint′ = TRUE
        ∧  UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
                      moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position,
                      step3, W0_atteint, contenu, WA_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Obtenir_VB3 ≜
        ∧ Obtenir_VB2
        ∧ step3 = verser_VB
        ∧ vb = open
        ∧ WB_atteint = TRUE
        ∧ step3′ = ouvrir_VC
        ∧ vb′ = close
        ∧ UNCHANGED ⟨contenu, va, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle,
                      W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Init_transvaser ≜
        ∧ step3 = ouvrir_VC
        ∧ contenu[mixeur] = ⟨0, 0, 0, 0⟩
        ∧ contenu[cuve] = ⟨ma, mb, 0, 0⟩
        ∧ vc = close
        ∧ va = close
        ∧ vb = close
        ∧ angle = 90
        ∧ moteur_convoyer = off
        ∧ moteur_malaxer = off
        ∧ vc′ = open
        ∧ step3′ = transvaser
        ∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb,
                      moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint,
                      WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Laisser_transvaser ≜
        ∧ vc = open
        ∧ angle = 90
        ∧ ∃ ⟨x, y, u, v, a, b, c, d⟩  ∈ (0 .. ma) ×  (0 .. mb) × (0 .. ma) × (0 .. mb) × (0 .. ma) ×  (0 .. mb) × (0 .. ma) × (0 .. mb) :
        ∧ x + y ∈ 1 .. ma + mb ∧ u + v ∈ 0 .. ma + mb − 1  ∧  x + y + u + v = ma + mb ∧ x + u = ma ∧ y + v = mb
        ∧ a + b ∈ 1 .. ma + mb ∧ c + d ∈ 0 .. ma + mb − 1 ∧ a + c = ma ∧ b + d = mb ∧ a + b + c + d = ma + mb
        ∧ contenu[cuve] = ⟨x, y, 0, 0⟩ ∧ contenu[mixeur] = ⟨u, v, 0, 0⟩  ∧ contenu′ = (mixeur :> ⟨c, d, 0, 0⟩ @@
       cuve :> ⟨a, b, 0, 0⟩) ∧ a + b = x + y − 1 ∧ c + d = u + v + 1
        ∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
                     moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
                     W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
Detect_WC ≜
        ∧  contenu[mixeur] = ⟨ma, mb, 0, 0⟩
        ∧  WC_atteint′ = TRUE
        ∧  UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
                      moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
                      W0_atteint, contenu, WA_atteint, WB_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩
```

Transvaser3 $\triangleq$

$\wedge$ Transvaser

$\wedge$ angle = 90

$\wedge$ WC_atteint = TRUE

$\wedge$ step3 = transvaser

$\wedge$ vc = open

$\wedge$ step3′ = marche_moteur_conv

$\wedge$ vc′ = close

$\wedge$ UNCHANGED ⟨contenu, va, vb, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Init_convoyer $\triangleq$

$\wedge$ step3 = marche_moteur_conv

$\wedge$ moteur_convoyer = off

$\wedge$ angle = 90

$\wedge$ vc = close

$\wedge$ moteur_convoyer′ = on

$\wedge$ step3′ = verser_brique

$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint,
WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Laisser_tomber_briques $\triangleq$

$\wedge$ moteur_convoyer = on

$\wedge \exists$ nb $\in 0 \,..\, n-1$: contenu[mixeur] = ⟨ma, mb, nb, 0⟩ $\wedge$ contenu′ = [contenu EXCEPT ![mixeur] = ⟨ma, mb, nb + 1, 0⟩]

$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Detect_briques $\triangleq$

$\wedge$ contenu[mixeur] = ⟨ma, mb, n, 0⟩

$\wedge$ n_atteint′ = TRUE

$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, time_atteint, produit_livrer, angle_atteint⟩

Obtenir_briques3 $\triangleq$

$\wedge$ Obtenir_briques2

$\wedge$ step3 = verser_brique

$\wedge$ moteur_convoyer = on

$\wedge$ n_atteint = TRUE

$\wedge$ step3′ = marche_moteur_mal

$\wedge$ moteur_convoyer′ = off

$\wedge$ UNCHANGED ⟨contenu, va, vb, vc, moteur_malaxer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint,
WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Init_malaxer $\triangleq$

$\wedge$ step3 = marche_moteur_mal

$\wedge$ moteur_malaxer = off

$\wedge$ angle = 90

$\wedge$ vc = close

$\wedge$ moteur_malaxer′ = on

$\wedge$ step3′ = malaxer

$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint,
WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Laisser_malaxer $\triangleq$

$\wedge$ moteur_malaxer = on

$\wedge\ \exists\, tt \in 0\,..\,tm-1 :\ \wedge\ contenu[mixeur] = \langle ma,\ mb,\ n,\ tt\rangle\ \wedge\ contenu' = [contenu\ \text{EXCEPT}\ ![mixeur] = \langle ma,\ mb,\ n,\ tt+1\rangle]$

$\wedge\ \text{UNCHANGED}\ \langle loc\_produit,\ loc1\_produit,\ loc\_prod,\ en\_marche,\ produit,\ step,\ step2,\ prod,\ va,\ vb,\ vc,\ moteur\_convoyer,$
$moteur\_malaxer,\ moteur\_tourner,\ moteur\_livrer,\ angle,\ position,\ step3,\ W0\_atteint,\ WA\_atteint,$
$WB\_atteint,\ WC\_atteint,\ n\_atteint,\ time\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Detect\_temps \triangleq$

$\wedge\ contenu[mixeur] = \langle ma,\ mb,\ n,\ tm\rangle$

$\wedge\ time\_atteint' = \text{TRUE}$

$\wedge\ \text{UNCHANGED}\ \langle loc\_produit,\ loc1\_produit,\ loc\_prod,\ en\_marche,\ produit,\ step,\ step2,\ prod,\ va,\ vb,\ vc,$
$moteur\_convoyer,\ moteur\_malaxer,\ moteur\_tourner,\ moteur\_livrer,\ angle,\ position,\ step3,$
$W0\_atteint,\ contenu,\ WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ n\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Fabriquer3 \triangleq$

$\wedge\ Fabriquer2$

$\wedge\ step3 = malaxer$

$\wedge\ moteur\_malaxer = on$

$\wedge\ time\_atteint = \text{TRUE}$

$\wedge\ step3' = marche\_moteur\_bas$

$\wedge\ moteur\_malaxer' = off$

$\wedge\ \text{UNCHANGED}\ \langle contenu,\ va,\ vb,\ vc,\ moteur\_convoyer,\ moteur\_tourner,\ moteur\_livrer,\ angle,\ W0\_atteint,$
$WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ n\_atteint,\ time\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Init\_basculer \triangleq$

$\wedge\ step3 = marche\_moteur\_bas$

$\wedge\ moteur\_tourner = off$

$\wedge\ moteur\_tourner' = on$

$\wedge\ step3' = basculer$

$\wedge\ \text{UNCHANGED}\ \langle loc\_produit,\ loc1\_produit,\ loc\_prod,\ en\_marche,\ produit,\ step,\ step2,\ prod,\ va,\ vb,\ vc,$
$moteur\_convoyer,\ moteur\_malaxer,\ moteur\_livrer,\ angle,\ position,\ contenu,\ W0\_atteint,$
$WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ n\_atteint,\ time\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Laisser\_basculer \triangleq$

$\wedge\ moteur\_tourner = on$

$\wedge\ angle \in (1\,..\,90)$

$\wedge\ moteur\_convoyer = off$

$\wedge\ moteur\_malaxer = off$

$\wedge\ vc = close$

$\wedge\ angle' = angle - 1$

$\wedge\ \text{UNCHANGED}\ \langle loc\_produit,\ loc1\_produit,\ loc\_prod,\ en\_marche,\ produit,\ step,\ step2,\ prod,\ va,\ vb,\ vc,$
$moteur\_convoyer,\ moteur\_malaxer,\ moteur\_tourner,\ moteur\_livrer,\ contenu,\ position,\ step3,$
$W0\_atteint,\ WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ n\_atteint,\ time\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Detect\_angle \triangleq$

$\wedge\ angle = 0$

$\wedge\ angle\_atteint' = \text{TRUE}$

$\wedge\ \text{UNCHANGED}\ \langle loc\_produit,\ loc1\_produit,\ loc\_prod,\ en\_marche,\ produit,\ step,\ step2,\ prod,\ va,\ vb,\ vc,$
$moteur\_convoyer,\ moteur\_malaxer,\ moteur\_tourner,\ moteur\_livrer,\ angle,\ position,\ step3,$
$W0\_atteint,\ contenu,\ WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ time\_atteint,\ produit\_livrer,\ n\_atteint\rangle$

$Basculer2 \triangleq$

$\wedge\ \ Basculer$

$\wedge\ \ moteur\_tourner = on$

$\wedge\ \ step3 = basculer$

$\wedge\ \ angle\_atteint = \text{TRUE}$

$\wedge\ \ moteur\_tourner' = off$

$\wedge\ \ step3' = marche\_moteur\_liv$

$\wedge\ \ \text{UNCHANGED}\ \langle contenu,\ va,\ vb,\ vc,\ moteur\_convoyer,\ moteur\_livrer,\ moteur\_malaxer,\ angle,\ W0\_atteint,$
$WA\_atteint,\ WB\_atteint,\ WC\_atteint,\ n\_atteint,\ time\_atteint,\ produit\_livrer,\ angle\_atteint\rangle$

$Init\_livrer \triangleq$

$\wedge\ step3 = marche\_moteur\_liv$

$\wedge\ moteur\_livrer = off$

$\wedge\ moteur\_livrer' = on$

$\wedge$ step3′ = livrer
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_malaxer, moteur_tourner, angle, position, contenu, W0_atteint,
WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Laisser_livrer $\triangleq$
$\wedge$ moteur_livrer = on
$\wedge$ contenu′ = (out :> ⟨ma, mb, n, tm⟩ @@ mixeur :> ⟨0, 0, 0, 0⟩ @@ cuve :> ⟨0, 0, 0, 0⟩ @@ no_where :> ⟨0, 0, 0, 0⟩)
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Detect_produit $\triangleq$
$\wedge$ contenu[out] = ⟨ma, mb, n, tm⟩
$\wedge$ produit_livrer′ = TRUE
$\wedge$ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc,
moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3,
W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, time_atteint, angle_atteint, n_atteint⟩

Livrer3 $\triangleq$
$\wedge$ Livrer2
$\wedge$ step3 = livrer
$\wedge$ moteur_livrer = on
$\wedge$ produit_livrer = TRUE
$\wedge$ moteur_livrer′ = off
$\wedge$ step3′ = terminer
$\wedge$ UNCHANGED ⟨contenu, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_malaxer, angle, W0_atteint,
WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Finir3 $\triangleq$
$\wedge$ Finir2
$\wedge$ step3 = terminer
$\wedge$ step3′ = commencer
$\wedge$ UNCHANGED ⟨contenu, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_malaxer, moteur_livrer, angle,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

Next3 $\triangleq$
$\vee$ Start3
$\vee$ Ouvrir_VA
$\vee$ Remplissage_cuveAC
$\vee$ Detect_WA
$\vee$ Obtenir_VA3
$\vee$ Ouvrir_VB
$\vee$ Remplissage_cuveBC
$\vee$ Detect_WB
$\vee$ Obtenir_VB3
$\vee$ Init_transvaser
$\vee$ Laisser_transvaser
$\vee$ Detect_WC
$\vee$ Transvaser3
$\vee$ Init_convoyer
$\vee$ Laisser_tomber_briques
$\vee$ Detect_briques
$\vee$ Obtenir_briques3
$\vee$ Init_malaxer
$\vee$ Laisser_malaxer
$\vee$ Detect_temps
$\vee$ Fabriquer3
$\vee$ Init_basculer
$\vee$ Laisser_basculer
$\vee$ Detect_angle

$\lor$ Basculer2
$\lor$ Init_livrer
$\lor$ Laisser_livrer
$\lor$ Detect_produit
$\lor$ Livrer3
$\lor$ Finir3

---

trvars3 $\triangleq$ ⟨va, vb, vc, step3, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, contenu,
W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint⟩

TrFairness3 $\triangleq$ $\mathrm{WF}_{\mathrm{trvars}}$(Ouvrir_VA) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_WA) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Obtenir_VA3)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Ouvrir_VB) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_WB) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Obtenir_VB3)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Ouvrir_VC) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_WC) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Transvaser3)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Init_convoyer) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_briques) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Obtenir_briques3)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Init_malaxer) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_temps) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Fabriquer3)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Init_basculer) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_angle) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Basculer2)
$\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Init_livrer) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_produit) $\land$ $\mathrm{WF}_{\mathrm{trvars}}$(Livrer3)

Spec3 $\triangleq$ Init3 $\land$ $\Box[\mathrm{Next3}]_{\mathrm{trvars3}}$ $\land$ TrFairness3

Invariant3 $\triangleq$

$\land$ step3 $\in$ {commencer, fabriquer} $\implies$ step2 = step3
$\land$ step2 $\in$ {commencer, fabriquer} $\implies$ step3 = step2
$\land$ step3 $\in$ {ouvrir_VA, verser_VA} $\implies$ step2 = verser_VA
$\land$ step2 = verser_VA $\implies$ step3 $\in$ {ouvrir_VA, verser_VA}
$\land$ step3 $\in$ {ouvrir_VB, verser_VB} $\implies$ step2 = verser_VB
$\land$ step2 = verser_VB $\implies$ step3 $\in$ {ouvrir_VB, verser_VB}
$\land$ step3 $\in$ {ouvrir_VC, transvaser} $\implies$ step2 = transvaser
$\land$ step2 = transvaser $\implies$ step3 $\in$ {ouvrir_VC, transvaser}
$\land$ step3 $\in$ {marche_moteur_bas, basculer} $\implies$ step2 = basculer
$\land$ step2 = basculer $\implies$ step3 $\in$ {marche_moteur_bas, basculer}
$\land$ step3 $\in$ {marche_moteur_liv, livrer, terminer} $\implies$ step2 = livrer
$\land$ step2 = livrer $\implies$ step3 $\in$ {marche_moteur_liv, livrer, terminer}
$\land$ step3 $\in$ {ouvrir_VA, verser_VA} $\implies$ loc_prod = no_where
$\land$ step3 = commencer $\implies$ loc_prod $\in$ {no_where, out}
$\land$ step3 $\in$ {ouvrir_VB, verser_VB, ouvrir_VC, transvaser} $\implies$ loc_prod = cuve
$\land$ loc_prod = cuve $\implies$ step3 $\in$ {ouvrir_VB, verser_VB, ouvrir_VC, transvaser}
$\land$ step3 $\in$ {verser_brique, marche_moteur_mal, malaxer, marche_moteur_bas, marche_moteur_conv, marche_moteur_liv, basculer, livrer}
$\implies$ loc_prod = mixeur
$\land$ step3 = terminer $\implies$ loc_prod = out
$\land$ $\neg$(vc = open $\land$ angle = 0)
$\land$ $\neg$(moteur_convoyer = on $\land$ angle = 0)
$\land$ $\neg$(moteur_malaxer = on $\land$ angle = 0)
$\land$ $\neg$(vc = open $\land$ moteur_convoyer = on)
$\land$ $\neg$(vc = open $\land$ moteur_malaxer = on)
$\land$ $\neg$(va = open $\land$ vb = open)
$\land$ $\neg$(va = open $\land$ vc = open)
$\land$ $\neg$(vb = open $\land$ vc = open)
$\land$ $\forall$ x, y : ($\land$ x $\in$ (0 . . ma) $\land$ y $\in$ (0 . . mb) $\land$ vc = close $\land$ vb = open $\land$ contenu[cuve] = ⟨x, y, 0, 0⟩ $\land$ y > 0 $\implies$ $\land$ x = ma)
$\land$ $\forall$ x, y, z : ($\land$ x $\in$ (0 . . ma) $\land$ y $\in$ (0 . . mb) $\land$ z $\in$ (0 . . n) $\land$ contenu[mixeur] = ⟨x, y, z, 0⟩ $\land$ z > 0 $\implies$ $\land$ x = ma $\land$ y = mb)
$\land$ $\forall$ x, y, z, t : (x $\in$ (0 . . ma) $\land$ y $\in$ (0 . . mb) $\land$ z $\in$ (0 . . n) $\land$ t $\in$ (0 . . tm) $\land$ contenu[mixeur] = ⟨x, y, z, t⟩ $\land$ t > 0
$\implies$ $\land$ x = ma $\land$ y = mb $\land$ z = n)

Liveness3 $\triangleq$

$\land$ step3 = commencer $\land$ contenu[out] = ⟨0, 0, 0, 0⟩ $\leadsto$ contenu[out] = ⟨ma, mb, n, tm⟩
$\land$ va = open $\leadsto$ contenu[cuve] = ⟨ma, 0, 0, 0⟩
$\land$ va = close $\land$ vb = open $\land$ vc = close $\leadsto$ contenu[cuve] = ⟨ma, mb, 0, 0⟩
$\land$ vc = open $\land$ va = close $\land$ vb = close $\leadsto$ contenu[mixeur] = ⟨ma, mb, 0, 0⟩
$\land$ contenu[cuve] = ⟨ma, mb, 0, 0⟩ $\land$ contenu[mixeur] = ⟨0, 0, 0, 0⟩ $\leadsto$ contenu[cuve] = ⟨0, 0, 0, 0⟩ $\land$ contenu[mixeur] = ⟨ma, mb, 0, 0⟩

THEOREM Spec3 $\Longrightarrow$ Spec2
THEOREM Spec3 $\Longrightarrow$ $\Box$Init3
THEOREM Spec3 $\Longrightarrow$ $\Box$TypeInvariant3
THEOREM Spec3 $\Longrightarrow$ $\Box$Invariant3
THEOREM Spec3 $\Longrightarrow$ Liveness3

5 Module du quatrième raffinement

MODULE mix4

EXTENDS mix3, Naturals, TLC
CONSTANTS d_max
VARIABLES d, detect
ASSUME d_max $\in$ Nat
$\wedge$ d_max $\neq 0$

TypeInvariant4 $\triangleq$ $\wedge$ TypeInvariant3
$\wedge$ d $\in$ (0 . . d_max)
$\wedge$ detect $\in$ BOOLEAN

Init4 $\triangleq$ $\wedge$ Init3
$\wedge$ d $= 0$
$\wedge$ detect $=$ FALSE
Start4 $\triangleq$ $\wedge$ Start3
$\wedge$ d$'$ $= 0$
$\wedge$ detect$'$ $=$ FALSE
Ouvrir_VA2 $\triangleq$
$\wedge$ Ouvrir_VA
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$
Remplissage_cuveAC2 $\triangleq$
$\wedge$ Remplissage_cuveAC
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$
Detect_WA2 $\triangleq$
$\wedge$ Detect_WA
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$
Obtenir_VA4 $\triangleq$
$\wedge$ Obtenir_VA3
$\wedge$ UNCHANGED $\langle$contenu, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$
Ouvrir_VB2 $\triangleq$
$\wedge$ Ouvrir_VB
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$
Remplissage_cuveBC2 $\triangleq$
$\wedge$ Remplissage_cuveBC
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer,

moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Detect_WB2 $\triangleq$

∧ Detect_WB

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Obtenir_VB4 $\triangleq$

∧ Obtenir_VB3

∧ UNCHANGED ⟨contenu, va, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Init_transvaser2 $\triangleq$

∧ Init_transvaser

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Laisser_transvaser2 $\triangleq$

∧ Laisser_transvaser

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Detect_WC2 $\triangleq$

∧ Detect_WC

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WB_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Transvaser4 $\triangleq$

∧ Transvaser3

∧ UNCHANGED ⟨contenu, va, vb, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Init_convoyer2 $\triangleq$

∧ Init_convoyer

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect⟩

Avancer_brique $\triangleq$

∧ step3 = verser_brique

∧ moteur_convoyer = on

∧ d < d_max

∧ detect = FALSE

∧ d′ = d + 1

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, contenu, detect⟩

Detect_brique $\triangleq$

∧ step3 = verser_brique

∧ moteur_convoyer = on

∧ d = d_max

∧ detect = FALSE

∧ detect′ = TRUE

∧ d′ = 0

∧ UNCHANGED ⟨loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, contenu⟩

Laisser_tomber_briques2 $\triangleq$

∧ moteur_convoyer = on

$\wedge \exists\, nb \in 0 \,..\, n-1 : contenu[mixeur] = \langle ma,\ mb,\ nb,\ 0\rangle \quad \wedge contenu' = [contenu \text{ EXCEPT } ![mixeur] = \langle ma,\ mb,\ nb+1,\ 0\rangle] \quad \wedge detect' = \text{FALSE}$

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d$\rangle$

Detect_briques2 $\triangleq$

$\wedge$ Detect_briques

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Obtenir_briques4 $\triangleq$

$\wedge$ Obtenir_briques3

$\wedge$ UNCHANGED $\langle$contenu, va, vb, vc, moteur_malaxer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Init_malaxer2 $\triangleq$

$\wedge$ Init_malaxer

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Laisser_malaxer2 $\triangleq$

$\wedge$ Laisser_malaxer

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Detect_temps2 $\triangleq$

$\wedge$ Detect_temps

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, n_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Fabriquer4 $\triangleq$

$\wedge$ Fabriquer3

$\wedge$ UNCHANGED $\langle$contenu, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Init_basculer2 $\triangleq$

$\wedge$ Init_basculer

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_livrer, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Laisser_basculer2 $\triangleq$

$\wedge$ Laisser_basculer

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, contenu, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Detect_angle2 $\triangleq$

$\wedge$ Detect_angle

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, time_atteint, produit_livrer, n_atteint, d, detect$\rangle$

Basculer3 $\triangleq$

$\wedge$ Basculer2

$\wedge$ UNCHANGED $\langle$contenu, va, vb, vc, moteur_convoyer, moteur_livrer, moteur_malaxer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Init_livrer2 $\triangleq$

$\wedge$ Init_livrer

$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, angle, position, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Laisser_livrer2 $\triangleq$
$\wedge$ Laisser_livrer
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Detect_produit2 $\triangleq$
$\wedge$ Detect_produit
$\wedge$ UNCHANGED $\langle$loc_produit, loc1_produit, loc_prod, en_marche, produit, step, step2, prod, va, vb, vc, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, position, step3, W0_atteint, contenu, WA_atteint, WB_atteint, WC_atteint, time_atteint, angle_atteint, n_atteint, d, detect$\rangle$

Livrer4 $\triangleq$
$\wedge$ Livrer3
$\wedge$ UNCHANGED $\langle$contenu, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_malaxer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Finir4 $\triangleq$
$\wedge$ Finir3
$\wedge$ UNCHANGED $\langle$contenu, va, vb, vc, moteur_convoyer, moteur_tourner, moteur_malaxer, moteur_livrer, angle, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

Next4 $\triangleq$
$\vee$ Start4
$\vee$ Ouvrir_VA2
$\vee$ Remplissage_cuveAC2
$\vee$ Detect_WA2
$\vee$ Obtenir_VA4
$\vee$ Ouvrir_VB2
$\vee$ Remplissage_cuveBC2
$\vee$ Detect_WB2
$\vee$ Obtenir_VB4
$\vee$ Init_transvaser2
$\vee$ Laisser_transvaser2
$\vee$ Detect_WC2
$\vee$ Transvaser4
$\vee$ Init_convoyer2
$\vee$ Laisser_tomber_briques2
$\vee$ Detect_briques2
$\vee$ Obtenir_briques4
$\vee$ Avancer_brique
$\vee$ Detect_brique
$\vee$ Init_malaxer2
$\vee$ Laisser_malaxer2
$\vee$ Detect_temps2
$\vee$ Fabriquer4
$\vee$ Init_basculer2
$\vee$ Laisser_basculer2
$\vee$ Detect_angle2
$\vee$ Basculer3
$\vee$ Init_livrer2
$\vee$ Laisser_livrer2
$\vee$ Detect_produit2
$\vee$ Livrer4
$\vee$ Finir4

---

trvars4 $\triangleq$ $\langle$va, vb, vc, step3, moteur_convoyer, moteur_malaxer, moteur_tourner, moteur_livrer, angle, contenu, W0_atteint, WA_atteint, WB_atteint, WC_atteint, n_atteint, time_atteint, produit_livrer, angle_atteint, d, detect$\rangle$

TrFairness4 $\triangleq$ $\mathrm{WF}_{\mathrm{trvars}}$(Ouvrir_VA) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_WA) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Obtenir_VA3) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Ouvrir_VB) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Detect_WB) $\wedge$ $\mathrm{WF}_{\mathrm{trvars}}$(Obtenir_VB3)

$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Ouvrir\_VC}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_WC}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Transvaser3})$
$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Init\_convoyer}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_briques}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Obtenir\_briques3})$
$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Init\_malaxer}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_temps}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Fabriquer3})$
$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Init\_basculer}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_angle}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Basculer2})$
$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Init\_livrer}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_produit}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Livrer3})$
$\wedge\ \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Avancer\_brique}) \wedge \mathrm{WF}_{\mathrm{trvars}}(\mathrm{Detect\_brique})$

$\mathrm{Spec4} \triangleq \mathrm{Init4} \wedge \Box[\mathrm{Next4}]_{\mathrm{trvars4}} \wedge \mathrm{TrFairness4}$

$\mathrm{Liveness4} \triangleq$
$\wedge\ \mathrm{moteur\_convoyer} = \mathrm{on} \rightsquigarrow \mathrm{detect} = \textsc{true}$

THEOREM $\mathrm{Spec4} \Longrightarrow \mathrm{Spec3}$
THEOREM $\mathrm{Spec4} \Longrightarrow \Box\mathrm{Init4}$
THEOREM $\mathrm{Spec4} \Longrightarrow \Box\mathrm{TypeInvariant4}$
THEOREM $\mathrm{Spec4} \Longrightarrow \mathrm{Liveness4}$

INSTITUT NATIONAL
POLYTECHNIQUE
DE LORRAINE

AUTORISATION DE SOUTENANCE DE THESE
DU DOCTORAT DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE LORRAINE

o0o

VU LES RAPPORTS ETABLIS PAR :
Monsieur Jacques JULLIAND, Professeur, LIFC, Université de Franche Comté, Besançon

Monsieur Riadh ROBBANA, Maître de Conférences, Ecole Polytechnique de Tunisie, Tunisie

Le Président de l'Institut National Polytechnique de Lorraine, autorise :

Madame MOSBAHI Olfa épouse KHALGUI

NANCY BRABOIS
2, AVENUE DE LA
FORET-DE-HAYE
BOITE POSTALE 3
F - 54501
VANDŒUVRE CEDEX

à soutenir devant un jury de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée :

"Développement formel de systèmes automatisés"

en vue de l'obtention du titre de :

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

Spécialité : « **Informatique** »

Fait à Vandoeuvre, le 05 février 2008
Le Président de l'I.N.P.L.,
F. LAURENT