CONSERVATOIRE NATIONAL DES ARTS ET METIERS

THESE DE DOCTORAT

Spécialité : **Informatique**

Présentée par : **Nestor LOPEZ**

Pour obtenir le grade de DOCTEUR DU CNAM

Sujet de la thèse :

Spécification Formelle de Systèmes Complexes Méthodes et Techniques

Soutenue le 31 Janvier 2002 devant le Jury composé de :

M. Christian CARREZ	Président
M. Dominique MERY	Rapporteur
M. Yves LEDRU	Rapporteur
M. Bill STODDART	Examinateur
M. Pierre DESFORGES	Examinateur
Mme. Véronique VIGUIE DONZEAU-GOUGE	Directeur
Mme. Marianne SIMONOT	Directeur

Remerciements

Plusieurs personnes ont participé directement ou indirectement dans la réalisation de ce travail. Je les remercie toutes car l'apport de chacune d'entre elles a été d'une façon ou d'une autre important pour atteindre nos objectifs. Je voudrais remercier en particulier :

Jean Raymond Abrial pour l'aide qu'il nous a offerte et donnée au début de nos recherches. Je voudrais aussi lui exprimer ma gratitude pour m'avoir tant aidé et guidé pendant mes études. Ses idées m'ont fortement marqué et son enthousiasme m'a toujours encouragé.

Véronique Viguié Donzeau-Gouge et Marianne Simonot pour avoir dirigé mes travaux avec continuité et patience.

Dominique Mery et Yves Ledru pour avoir accepté de rapporter cette thèse et pour leurs remarques et commentaires pertinents et utiles qui m'ont aidé grandement à améliorer mon travail et qui m'ont donné des bonnes idées pour le futur.

M. Christian Carrez pour avoir accepté de présider le Jury et M. Bill Stoddart pour avoir accepté de faire partie de ce même Jury.

Pierre Desforges de la R.A.T.P et Patrick Behm de MATRA TRANSPORT INTERNATIONAL (Siemens) pour avoir fourni le cas d'étude industriel dont nous avions besoin et pour m'avoir tant appuyé et aidé dans la phase pratique de mon travail. Je remercie également toutes les autres personnes de la R.A.T.P. et de Matra Transport International qui ont participé dans cette étude.

La Société STERIA (ClearSy) pour avoir appuié mon travail et pour avoir fourni la licence de l'atelier B dont j'ai eu besoin. Je remercie en particulier Thierry Servat.

La Société AEROSPATIALE pour avoir fourni le premier cas d'étude industriel qui a été fondamental dans la préparation de mon mémoire d'ingénieur et que nous avons utilisé aussi dans ces travaux. Je remercie en particulier M. Philippe Contensou et M. François Pilarski pour leur aide. Je remercie également toutes les autres personnes de l'AEROSPATIALE qui ont participé dans cette étude.

Aymrique Burlureau pour avoir validé à l'aide de l'atelier B les modèles que nous avons construit et pour son travail qui nous a beaucoup aidé.

Le CNAM et le laboratoire CEDRIC pour avoir fourni le cadre académique et de recherche nécessaire à la réalisation de mon travail. Je remercie mes amis et collègues du CNAM, de l'université Paris VI et de la communauté B pour

toutes les séances de travail et les discussions que nous avons eu et qui m'ont tant enrichi et aidé .

Je remercie très spécialement la Société LOUIS DREYFUS pour son support constant. M. Claude Cacheux et M. Ilyas Bayar qui ont tout fait pour que je puisse avancer dans mes études. Mme. Nicole Pardo pour son appui et pour sa patience dans la gestion administrative de mon dossier. M. Bernard Laferrière et M. Bernard Baldensperger pour leur appui et pour la confiance qu'ils m'ont accordée.

Pour terminer, je remercie Sylvie et Matéo pour toute leur patience et compréhension.

Table des matières

1 Spécification de Programmes **9**
- 1.1 Programmes Séquentiels : Evolution . . . 9
- 1.2 Vérification de Programmes . . . 10
  - 1.2.1 Logique de Hoare . . . 10
  - 1.2.2 Plus Faible Pré-Condition . . . 12
  - 1.2.3 Définition de la Notion de Correction Totale . . . 16
- 1.3 Dérivation de Programmes . . . 16
- 1.4 Raffinement de Programmes . . . 17
- 1.5 Raffinement de Spécifications . . . 18
  - 1.5.1 Spécification Elémentaire . . . 19
  - 1.5.2 Faisabilité d'une Spécification Elémentaire . . . 20
  - 1.5.3 Non Déterminisme Interne . . . 20
  - 1.5.4 Relation de Raffinement entre Spécifications . . . 20
  - 1.5.5 Spécifications Associées aux Instructions . . . 22
  - 1.5.6 Spécifications Associées aux Programmes . . . 26
- 1.6 Utilisation de Spécifications Elémentaires . . . 30
- 1.7 Raffinement de données. . . . 31
  - 1.7.1 Raffinement des Paramètres d'une Spécification . . . 33
- 1.8 Conclusion . . . 34

2 Modules **37**
- 2.1 Spécification d'un Module . . . 38
  - 2.1.1 Non Déterminisme Interne . . . 38
  - 2.1.2 Non Déterminisme Externe . . . 38
- 2.2 Relation de Raffinement entre Spécifications de Module . . . 39
- 2.3 Invariant d'un Module . . . 41
- 2.4 Opérations . . . 41
- 2.5 Implantation d'une Opération . . . 42
- 2.6 Implantation d'un Module . . . 43
- 2.7 Constantes . . . 46
- 2.8 Spécification de Programmes : Analyse . . . 47
  - 2.8.1 Utilisation d'une Seule Opération avec Paramètres . . . 47
  - 2.8.2 Utilisation d'un Module avec Plusieurs Opérations . . . 48
  - 2.8.3 Utilisation d'un Module Clos . . . 50

2.8.4 Utilisation d'un Module pour Décrire le Comportement de l'Environnement . . . 53
2.9 Variables Auxiliaires . . . 54
2.9.1 Transformation d'une Spécification Elémentaire à l'Aide de Variables Auxiliaires . . . 55
2.9.2 Transformation d'un Module à l'Aide de Variables Auxiliaires . . . 61
2.9.3 Raffinement de Variables Auxiliaires . . . 65
2.10 Propriétés de Vivacité . . . 65
2.11 Opérations Auxiliaires . . . 66
2.12 Conclusion . . . 69

3 Modules Partagés 71
3.1 Un Système de Transfert de Fichiers . . . 72
3.1.1 Spécification de l'Emetteur . . . 72
3.1.2 Spécification du Récepteur . . . 75
3.1.3 Spécification du Canal de Communication . . . 78
3.2 Composition de Modules . . . 79
3.3 Utilisation de Modules Partagés . . . 83
3.4 Implantation de Modules Partagés . . . 84
3.5 Propriétés de Vivacité . . . 84
3.6 Terminaison et Equité . . . 85
3.7 Conclusion . . . 86

4 Utilisation de la Méthode B 89
4.1 Méthodes Formelles . . . 89
4.2 Spécifications Elémentaires en B . . . 89
4.2.1 Forme Normale des Substitutions Généralisées . . . 90
4.3 Modules . . . 93
4.4 Implantations . . . 95
4.5 Conclusion . . . 96

5 Spécification de Systèmes 97
5.1 Systèmes Concurrents : Evolution . . . 98
5.2 Observation d'un Système . . . 99
5.3 Spécification d'Evénements . . . 99
5.4 Raffinement d'Evénements . . . 100
5.5 Spécification Evénementielle d'un Système . . . 101
5.6 Raffinement de Spécifications Système . . . 102
5.6.1 Introduction de Nouveaux Evénements . . . 102
5.6.2 Décomposition d'un Evénement . . . 103
5.7 Implantation d'une Spécification Système . . . 103
5.7.1 Dérivation du Module de Communication . . . 104
5.7.2 Transformation d'un Evénement en une Opération . . . 105
5.8 Spécification des Composants du Système . . . 106
5.9 Exemple : Le Système de Contrôle d'Alarmes . . . 106

5.9.1 Description du système 108
5.9.2 Comportement du Système 109
5.9.3 Propriétés du Système 109
5.9.4 Specification Formelle 110
5.9.5 Dérivation du Module de Communication 114
5.9.6 Spécification de Composants 116
5.10 Conclusion 117

6 Cas d'Etude Industriel **119**
6.1 Construction de la Spécification Informelle 120
6.2 Construction de la Spécification Formelle 121

7 Conclusion **125**

A Cas d'Etude : Spécification Informelle **129**
A.1 Notions fondamentales 129
A.1.1 Le transfert de voyageurs 130
A.1.2 L'évacuation d'urgence d'un train 130
A.1.3 Le transfert de service 130
A.1.4 Les tests de fonctionnement de portes 130
A.2 Eléments qui forment la ligne météor 130
A.2.1 Les voies où peuvent circuler les trains 131
A.2.2 Les trains 132
A.2.3 Les quais de station 133
A.2.4 Les zones de garage et de maintenance 135
A.2.5 Le poste de contrôle centralisé (PCC) 135
A.2.6 Le système de contrôle et de commande automatique . . . 135
A.3 Le système de contrôle de météor (SAET) 135
A.3.1 Fonctionnement général du SAET 136
A.3.2 Le transfert de voyageurs 136
A.3.3 Les situations dangereuses à bord des trains 138
A.3.4 Panne du système de contrôle 138
A.3.5 Ouverture intempestive d'une porte d'un train en mouvement 139
A.3.6 Ouverture intempestive d'une porte palière 139
A.3.7 Sécurité intrinsèque 139
A.4 Propriétés fondamentales 139
A.4.1 Propriétés concernant le transfert de voyageurs 140
A.4.2 Propriétés concernant la fonction de contrôle sécuritaire . 140
A.4.3 Propriétés concernant l'évacuation d'urgence et l'actionnement d'une poignée de déverrouillage dans un train. . . . 141
A.4.4 Propriétés concernant l'actionnement de la poignée de déverrouillage de service. 141
A.4.5 Propriétés concernant une panne du système de contrôle. 142
A.4.6 Propriétés concernant une ouverture intempestive d'une porte d'un train en mouvement. 142

A.4.7 Propriétés concernant une ouverture intempestive d'une porte palière. 142
A.4.8 Procédure d'intervention sur site du personnel d'exploitation lors de l'évacuation d'un train en inter-station. 143
A.5 Premier niveau de détail 143
A.5.1 Autres notions fondamentales 143
A.5.2 Propriétés concernant l'évacuation d'urgence et l'actionnement d'une poignée de déverrouillage dans un train . . 146
A.5.3 Propriétés concernant le côté cheminement d'un train . . 149
A.5.4 Propriétés concernant les états où des actions sont commandées 149
A.5.5 Propriétés concernant les autorisations délivrées par la fonction de contrôle sécuritaire 150
A.5.6 Propriétés concernant la réalisation effective d'actions sur les portes palières 152
A.5.7 Propriétés concernant le contrôle de l'état des portes localisées sur un quai 153
A.6 Découpage du système de contrôle 154
A.6.1 Pilote Automatique Embarqué 154
A.6.2 Pilote Automatique Section 154
A.6.3 Poste de Contrôle Centralisé 154
A.6.4 Façade de Quai 154
A.6.5 Définition des interfaces entre les différentes sous-systèmes 155

B Cas d'Etude : Spécification Formelle 159
B.1 Analyse du système 159
B.1.1 Identification des notions fondamentales 159
B.1.2 Classement des propriétés fondamentales 160
B.2 Modélisation des principes fondamentaux 160
B.2.1 Analyse de propriétés 161
B.2.2 Objets utilisés 163
B.2.3 Invariants du système 166
B.2.4 Dynamique du système 167
B.2.5 Premier modèle 168
B.2.6 Preuves effectuées sur le premier modèle 170
B.2.7 Deuxième niveau 173
B.2.8 Preuves effectuées sur le deuxième modèle 177
B.2.9 Troisième niveau 178
B.2.10 Quatrième niveau 182

Table des figures

1 Construction de la Spécification Formelle d'un Système Complexe 6

1.1 Spécification Elémentaire 19

2.1 Spécification de Module 38
2.2 Processus d'Implantation d'un Module 45
2.3 Transformation de la Spécification d'un Module 55
2.4 Utilisation de Spécifications de Module Transformées 64

3.1 Spécification d'un Système de Transfert de Fichiers 73
3.2 Utilisation de Modules Partagés 80

5.1 Dérivation de la Spécification du Module de Communication . . . 104
5.2 Méthode de Spécification Proposée 107

Introduction

L'évolution technologique des dernières années a été suivie par une automatisation croissante des systèmes modernes, ce qui a engendré une augmentation considérable de leur complexité. Un système est formé par des composants qui doivent coopérer ensemble afin de réaliser des fonctions complexes souvent effectuées par l'homme. Un exemple de ce phénomène est la ligne météor du métro parisien dont l'exploitation est entièrement automatique : Les trains n'ont plus de conducteur et sont contrôlés par un système informatique distribué sur la ligne.

La maîtrise de cette complexité est un problème difficile qui a mis à l'épreuve les méthodes classiques de construction de programmes, et qui a motivé des nombreuses recherches durant ces dernières années.

L'origine des erreurs d'un système peut être expliquée principalement par le fait que les spécifications sont souvent écrites en langage naturel. *Il est par conséquent impossible de garantir la correction du système produit par rapport à la spécification initiale.* Une mesure palliative à ce problème consiste à effectuer des tests sur le comportement des composants et sur le système tout entier afin de détecter les erreurs, puis de les corriger. L'efficacité de cette méthode est néanmoins limitée par l'impossibilité de tester tous les cas possibles. Le processus de test ne peut pas être exhaustif pour un système d'intérêt industriel car le nombre d'états à vérifier atteint rapidement les limites du réalisable.

Le problème engendré par la construction de systèmes complexes peut être formulé comme suit : *existe-t-il une méthode de construction de systèmes qui garantisse la correction d'un système par rapport au comportement attendu ?*

Par correction nous voulons dire que le système réagit de la façon attendue aux événements qui se produisent dans son environnement, et qu'il n'adopte pas de comportement non attendu ou non souhaité. Il est donc nécessaire d'exprimer clairement le comportement souhaité pour pouvoir ensuite comparer et décider si le comportement réel est en accord avec les besoins exprimés. Le problème peut être divisé en deux parties distinctes : la spécification et l'implantation. Nous allons aborder ces deux problématiques séparément. Pour cela nous allons tenter de répondre aux deux questions qui suivent :

1. *Est-il possible de spécifier sans aucune ambiguïté le comportement d'un système ?*
2. *En supposant que nous disposons d'une telle spécification, est-t-il possible*

de construire un système correct vis-à-vis de cette spécification ?

Spécification : Les *langues naturelles* sont actuellement l'outil le plus utilisé pour décrire le comportement d'un système. Elles offrent un très grand pouvoir d'expression, une bonne flexibilité et, le plus important, elles sont utilisées au quotidien dans la plus part des activités humaines. Il semble donc naturel de les utiliser en tant qu'outil de spécification. Les langues naturelles posent néanmoins deux problèmes. D'une part, elles permettent l'introduction d'idées ambiguës qui en conséquence, admettent plusieurs interprétations différentes. D'autre part, elles ne permettent pas de systématiser le traitement de la connaissance que nous avons sur le système.

Les *langages formels* sont un autre moyen utilisé pour décrire le comportement d'un système. Cette méthode permet d'éliminer le problème des ambiguïtés dans la spécification et permet de raisonner formellement sur les propriétés du système. Il y a cependant une difficulté qui surgit avec l'utilisation de langages formels : les personnes qui écrivent et qui lisent les spécifications doivent être formées et habituées aux méthodes mathématiques.

Implantation : Dans notre travail, nous nous situons dans le cadre du raffinement où la correction d'un système est obtenue à partir d'une spécification abstraite qui est ensuite transformée pas à pas jusqu'à l'obtention d'une implantation du système. A chaque étape nous devons prouver que la transformation effectuée est correcte. Les techniques de raffinement ont été utilisées avec succès pour construire des programmes séquentiels corrects. Nous les utilisons ici pour spécifier des systèmes complexes réels. Nous avons étudié des systèmes réels fournis par l'industrie pour découvrir les limites de ces techniques. Nous proposons des solutions aux problèmes rencontrés et nous formulons une méthodologie qui permet de construire des systèmes réels.

Motivation

Lors de la préparation de mon mémoire d'ingénieur au CNAM, nous avons spécifié un système industriel fourni par la Société Aérospatiale : le système de contrôle d'alarmes embarqué dans les Airbus A340 [Lop 96-1, Lop 96-2]. Pour cela nous avons utilisé la méthode B et la notion de module présentée dans [Abr 96-1]. Durant la réalisation de ces travaux, nous avons rencontré plusieurs difficultés qui sont expliquées en détail dans [Lop,Sim,Don 2000]. Notre but est de chercher et de proposer des solutions aux problèmes rencontrés.

Sujets Abordés

Nous savions dès le départ qu'il était possible de partir d'une spécification formelle pour dériver un programme correct à l'aide de techniques de raffinement [Abr 96-1, Bac 88, Morg 90]. Nous avions aussi acquis une première expérience dans l'écriture de spécifications formelles de systèmes industriels [Lop 96-1, Lop 96-2].

Pour pouvoir exprimer formellement des propriétés qui font intervenir plusieurs composants, il est nécessaire de regarder le système dans son ensemble ce

qui complexifie considérablement sa spécification. Nous avions connaissance de plusieurs travaux qui abordent ce problème [1] . Ils proposent d'utiliser un module formé par un ensemble de variables d'état et par un ensemble d'événements observables. Ainsi, l'évolution du système est dirigée par ses événements. Cette approche nous paraissait bonne et nous avons décidé de l'utiliser pour spécifier un système de taille industriel : le cas d'étude du chapitre 6. Nous avons identifié trois problèmes principaux :

1. Le nombre d'événements qu'il faut inclure dans la spécification croît rapidement avec l'introduction de détails du système. Très vite, nous avons eu l'impression de ne plus maîtriser ce que nous faisions. Le problème principal vient du fait que tous les événements doivent être regroupés en un seul module. Nous avons conclu qu'il était indispensable de chercher des moyens de réduire ce nombre en structurant la spécification.
2. Les problèmes de l'implantation de la spécification obtenue et de la dérivation des spécifications de composants à partir de la spécification événementielle n'est pas abordé, ce qui nous paraissait inacceptable parce que cela casse le processus de développement formel : l'implantation est déconnectée de la spécification. Nous avons décidé d'aborder ce sujet avec l'objectif de trouver un moyen de relier spécification et implantation, et de pouvoir utiliser les techniques connues de dérivation de programmes pour implanter les composants.
3. La spécification contient des variables abstraites qui permettent d'enregistrer l'évolution du système. Cette technique est utilisée pour exprimer des propriétés qui font intervenir plusieurs états du système, et pour prendre en compte les interactions du système avec son environnement.

   Ces variables sont très utiles dans la phase de spécification car elles permettent de tracer l'évolution du système, mais elles doivent disparaître pendant le processus d'implantation car souvent elles ne sont pas implantées. Il nous a paru indispensable de clarifier leur impact dans une spécification et de déterminer les moyens soit de les implanter soit de les éliminer durant le processus d'implantation.

Notre travail de recherche a été consacré principalement à l'étude de ces trois problèmes.

Organisation du Travail de Recherche

Notre travail s'est divisé en trois parties :

1. L'étude des techniques de spécification et de construction de programmes corrects utilisées à travers le temps. Nous nous sommes intéressés surtout aux techniques qui s'appuient sur les notions de système état transition et de construction de programmes par raffinements successifs. Notre objectif était de bien comprendre ces techniques afin de déterminer leur apport et leur limite.

[1] [Abr 96-2, Abr,Mus 97, Bac,Kur 83, But 96] entre autres.

2. La spécification d'un système réel fourni par l'industrie.
3. La recherche de solutions aux problèmes rencontrés durant la spécification de ce système, et l'élaboration d'une méthodologie de spécification et de développement qui puisse être utilisée dans l'industrie.

Partie Pratique

nous avons tenté de construire la spécification formelle d'un système complexe différent de celui traité dans [Lop 96-1, Lop 96-2] : le cas d'étude présenté dans le chapitre 6. Ce sujet a été fourni par les Sociétés RATP et Matra Transport International, avec la participation de la Société STERIA. Notre intention était de trouver par la pratique les problèmes qui peuvent être rencontrés pendant le processus de spécification d'un tel système. Notre but final était de trouver une méthodologie de spécification adaptée à ce type de systèmes qui permette de dériver les spécifications formelles des composants qui sont utilisés dans l'implantation finale, mais sans rompre le processus de raisonnement formel. Nous voulions aussi étudier le problème de la réutilisation de composants déjà existants. Nous avons adopté l'approche par événements proposé dans [Abr 96-2, Abr,Mus 97], et nous avons suivi la méthodologie utilisée dans [Lop 96-1, Lop 96-2]. Nous sommes arrivés au conclusions suivantes :

- La description des interactions d'un système avec son environnement doit être prise en compte très en amont du processus de spécification [2] . La raison est que ces interactions correspondent aux événements observables de l'extérieur et que la spécification du système doit refléter ces observations.
- Le modèle global du système doit prendre en compte les interactions qui peuvent exister entre les différents composants. La raison est que ces interactions ne peuvent pas être spécifiées à l'intérieur d'un seul composant.
- Les propriétés qui concernent uniquement un composant ne doivent pas être introduites dans le modèle global afin éviter l'explosion du nombre d'événements à traiter. Ceci permet de structurer la spécification.
- Il doit être possible de dériver une spécification indépendante pour chaque composant. Une raison est que ces composants doivent être implantés séparément. Une autre raison est la réutilisation des composants déjà existants, ce qui correspond à une logique industrielle de réutilisation de dispositifs existants.

Solutions Proposées

Pour résoudre le deuxième problème exposé plus haut, page 3, nous donnons des conditions qui permettent de relier une spécification basée sur la notion d'événement avec la spécification d'un module basée sur la notion d'opération, pré-condition plus post-condition. Cette mise en relation permet de transformer la spécification événementielle d'un système en une spécification *pré-post*, ce qui rend possible son implantation à l'aide de techniques connues de dérivation

[2] Cette idée est utilisée dans [Led 93].

de programmes par raffinements [Abr 96-1, Bac 80, Morg 90] et sans rompre le processus de raisonnement formel.

Dans notre approche, le rôle de la spécification événementielle est de décrire les interactions entre le système et son environnement et entre ses différents composants. La transformation de cette spécification en une spécification pré-post produit la spécification d'un module qui fournit les moyens de communication dont dispose le système.

Pour prendre en compte les aspects concurrents d'un système, nous proposons la notion de module partagé. Cette notion permet de spécifier séparément les composants du système, ce qui apporte une bonne réponse au problème de la structuration de la spécification. Premier problème exposé. La notion de module partagé est connue dans les langages de programmation, mais semble être nouvelle dans le domaine de la dérivation de programmes par raffinements. Dans [Abr 96-2, Abr,Mus 97] la communication entre composants est effectuée à l'aide de variables partagées. Dans [But 96], elle est effectuée à l'aide d'actions partagées. Nous utilisons les modules partagés pour le faire.

Pour traiter le problème des variables abstraites utilisées dans les spécifications, troisième problème exposé, nous utilisons la notion de variable auxiliaire [Aba,Lam 88, Lam,Sha 90, Owi,Gri 76, Sha 93]. Ce sont des variables qui permettent soit d'exprimer les propriétés souhaitées soit de faire la preuve de correction d'une implantation d'un module, mais qui ne sont pas implantées. Dans [Aba,Lam 88], elles sont utilisées pour compléter une spécification concrète afin de montrer qu'elle est un raffinement correct d'une spécification abstraite. Cette référence bibliographique contient une description détaillée de ces variables.

Pour répondre au besoin de réutilisation de modules existants, nous proposons, à l'aide des variables auxiliaires, un moyen de compléter une spécification pour l'adapter à la solution de problèmes particuliers tout en préservant l'implantation fournie initialement.

Méthodologie Proposée

Disposant de variables auxiliaires et de modules partagés, il nous a été possible de concevoir une démarche méthodologique qui permet la construction de la spécification formelle d'un système complexe et la dérivation des spécifications de ses composants. Elle comporte quatre phases, voir figure 1.

Dans la première phase, nous écrivons la *spécification informelle* dans une langue naturelle. Elle contient toutes les propriétés que le système doit satisfaire présentées dans un ordre qui respecte le niveau de détail des idées exprimées. C'est ce que nous faisons dans [Lop 96-1, Lop 96-2].

Dans la deuxième phase, nous construisons par petits pas une *spécification formelle* événementielle qui décrit les interactions du système avec l'environnement, et les interactions des différents composants. Elle contient aussi toutes les *hypothèses* que nous faisons sur l'environnement. La spécification informelle est le point de départ. Les propriétés sont exprimées dans un langage formel et sont introduites dans le même ordre de présentation suivi dans la spécification informelle, mais en incluant uniquement celles qui concernent plusieurs compo-

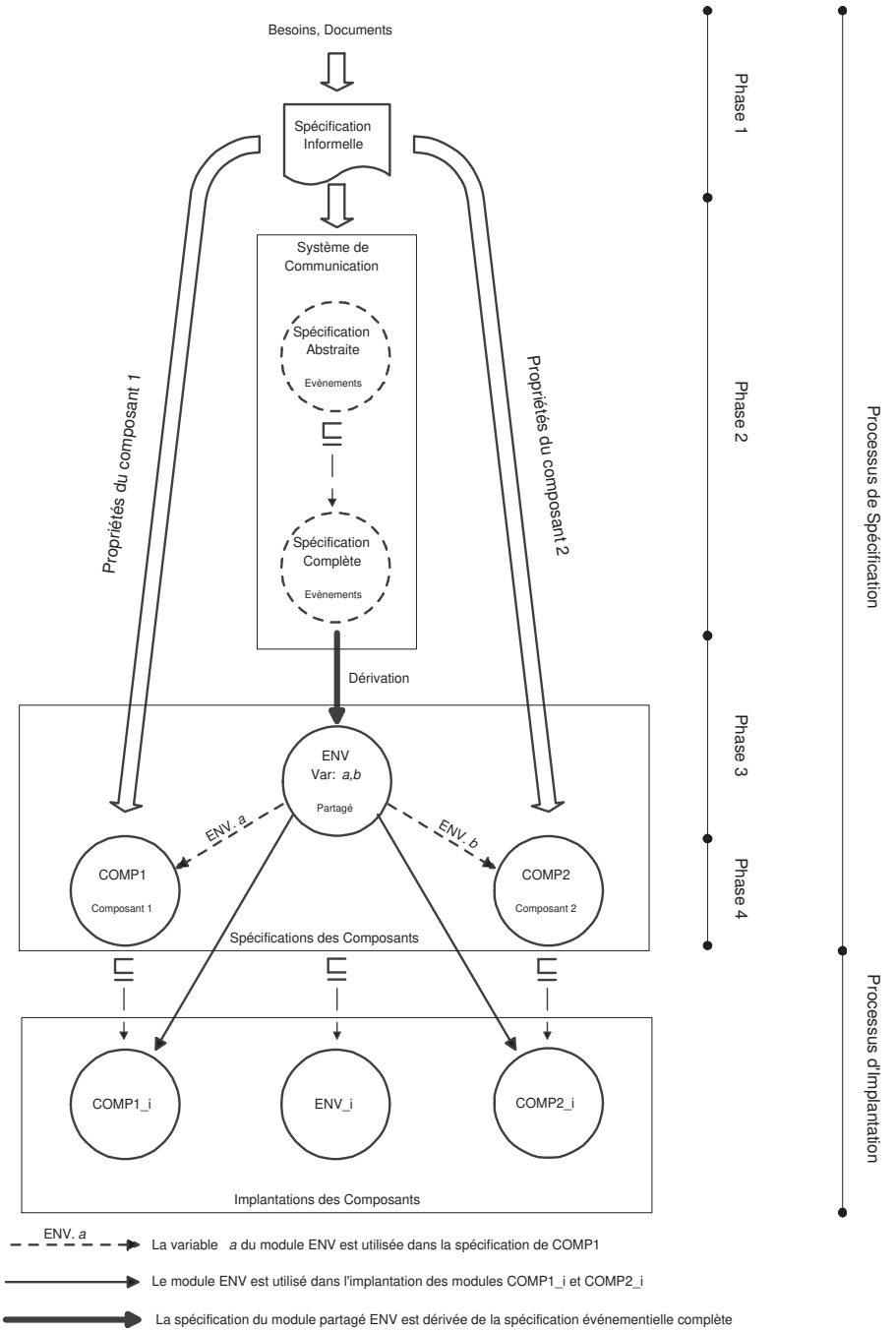

Figure 1 – Construction de la Spécification Formelle d'un Système Complexe

sants. Nous laissons de côté celles qui concernent uniquement un composant. Nous utilisons des techniques de raffinement pour conduire ce processus, voir le cas d'étude du chapitre 6, page 119. L'approche est descendante car nous allons du plus général vers le plus concret.

Dans la troisième phase, nous transformons la spécification événementielle en un ou plusieurs modules partagés qui servent dans les implantations des composants. Ces modules fournissent les mécanismes de communication internes et externes du système. Ils contiennent des variables auxiliaires qui permettent de spécifier le comportement souhaité pour chaque composant et de montrer que les implantations correspondantes sont correctes.

Dans la quatrième phase, nous écrivons séparément les spécifications des composants. Elles prennent en compte les propriétés décrites dans la spécification informelle qui sont relatives à chaque composant. Ces spécifications font intervenir les variables du module partagé dérivé dans la troisième phase.

A l'issue de ces quatre phases, chaque propriété informelle se trouve exprimée formellement dans une des spécifications obtenues. Il reste le processus d'implantation qui se fait par des techniques de raffinement traditionnelles.

La méthode proposée assure que tous les composants dérivés coopérant à travers les modules partagés forment un système dont le comportement est conforme à celui décrit par la spécification événementielle.

Le problème de la spécification des composants d'un système et des interactions avec l'environnement est abordé dans [Aba,Lam 95] dans le cadre de la logique temporelle d'actions. La spécification d'un système est vue ici comme la conjonction des spécifications de ses composants. Notre approche est différente car nous utilisons la notion de module partagé pour exprimer les propriétés globales du système. Nous spécifions séparément les composants et nous indiquons la façon de les implanter à l'aide de modules partagés.

La démarche méthodologique proposée s'appuie sur des notions théoriques connues qui sont en grand partie proposées dans la méthode B. De ce point de vue, nous avons adoptée une démarche très conservatrice.

Organisation du Document

Le document est divisé en 7 chapitres. Le chapitre 1, page 9, présente le cadre théorique général que nous utilisons. Les idées qu'il présente sont bien connues. Il contient une brève description de l'évolution des techniques qui ont été proposées à travers le temps pour spécifier et construire des programmes corrects. Deux approches très différentes sont étudiées : la vérification de programmes et la dérivation de programmes.

La notion de raffinement est traitée d'une façon plus complète car elle constitue l'outil théorique fondamental qui permet de conduire le processus de construction d'un système. Le chapitre présente deux relations de raffinement différentes : Le raffinement de programmes et le raffinement de spécifications. La première permet de raffiner un programme abstrait avec un autre plus concret. Ici les spécifications sont vues comme des programmes abstraits. La deuxième

permet de raffiner une spécification avec une autre. La fin du chapitre est dédiée à la notion de raffinement de données.

Nous utilisons un modèle relationnelle pour présenter le cadre théorique de notre travail car nous pensons qu'il facilite la compréhension des idées exposées. Notre but avec ce modèle n'est pas d'inventer une nouvelle méthode de développement formel. Nous voulons uniquement présenter les notions utilisées clairement et simplement.

Le chapitre 2, page 37, présente les concepts fondamentaux de module et de raffinement entre spécifications de module. Nous étudions l'apport des modules dans la spécification de programmes complexes et nous introduisons la notion de variable auxiliaire. Cette notion est étudiée en détail car elle joue un rôle très important dans le processus de spécification et d'implantation d'un système. La fin du chapitre est dédiée à la notion d'opération auxiliaire qui élargit considérablement l'utilisation qui peut être faite des variables auxiliaires. Un exemple de programme séquentiel est spécifié pour illustrer les idées présentées.

Le chapitre 3, page 71, introduit le concept de module partagé qui est fondamental dans la spécification et l'implantation de systèmes concurrents. Les difficultés engendrées par l'expression de propriétés de vivacité et d'équité sont étudiés, ainsi que le problème de la terminaison. Nous utilisons un petit exemple de système distribué pour développer ce concept.

Dans le chapitre 4, page 89, nous établissons les liens qui existent entre notre modèle théorique et la méthode B. Nous présentons d'abord les notions fondamentales de la méthode B. Puis, nous étudions la correspondance qui existe entre ces notions et celles de notre modèle théorique. Les lecteurs qui connaissent la méthode B peuvent laisser de côté ce chapitre.

Dans le chapitre 5, page 97, nous présentons d'abord une méthodologie de spécification de systèmes concurrents, centralisés et distribués, basée sur la notion d'événement. Elle permet de construire la spécification d'un système graduellement. Nous présentons ensuite la technique de transformation d'une spécification événementielle en une spécification pre-post d'un module partagé. La fin du chapitre est dédiée à l'étude de la problématique posée par la spécification de composants du système, et par la réutilisation de composants déjà existants. Nous utilisons ici le même cas d'étude industriel spécifié dans [Lop 96-1, Lop 96-2] mais en prenant en compte les aspects concurrents de ce système.

Le chapitre 6, page 119, contient une analyse du cas d'étude fourni par les Sociétés RATP et Matra Transport International dans le cadre d'une étude de faisabilité de l'utilisation de la méthode B très en amont dans le processus de construction d'un système. Il s'agit de l'étude et la modélisation de la fonction assurer la protection des voyageurs de la ligne Météor du métro Parisien. Les difficultés rencontrées lors de la réalisation de ce travail sont à l'origine de nombreuses idées présentées dans ce document, en particulier celles liées aux aspects concurrents d'un système. Les spécifications formelle et informelle que nous avons écrit pour ce système sont données en annexe.

Le chapitre 7, page 125, contient la conclusion.

Chapitre 1

Spécification de Programmes

Le raisonnement formel sur les programmes séquentiels et les systèmes concurrents a commencé au début des années 60 [Flo 67, McCa 63]. Ces deux disciplines ont évoluée d'une façon assez indépendante jusqu'aux années 80 où des travaux d'unification ont commencé à apparaître avec l'utilisation de techniques de construction de programmes par raffinements successifs et l'apparition de techniques de spécification incrémentale de systèmes concurrents basées sur la même idée de raffinement [Bac 90].

1.1 Programmes Séquentiels : Evolution

Floyd a été un des premiers à utiliser le raisonnement mathématique dans la preuve de programmes [Flo 67].

Hoare a défini la notion de correction partielle d'un programme par rapport à un couple de formules appelées pré-condition et post-condition. La pré-condition dénote les états initiaux dans lesquels le programme peut être utilisé, et la post-condition dénote les états après l'exécution du programme. Il a défini un ensemble d'axiomes et de règles d'inférence, connues sous le nom de logique de Hoare, qui permettent de prouver la correction partielle d'un programme [Hoa 69]. La notion de spécification formelle apparaît clairement ici. Le problème qui consiste à montrer la terminaison d'un programme n'est pas pris en compte.

Quelques années plus tard, Dijkstra a introduit la notion de plus faible pré-condition et de programme non déterministe [Dij 76]. Il associe à chaque programme une fonction qui transforme un prédicat, la post-condition, en un autre prédicat qui correspond à la plus faible pré-condition qui garantit la terminaison du programme. Cette idée permet de donner une sémantique précise à chaque instruction élémentaire d'un langage et à chaque programme écrit dans ce langage. La fonction associée à chaque programme est un transformateur de prédicats. Dijkstra donne en plus plusieurs propriétés importantes que tout programme qui termine doit satisfaire. Il formule en particulier la loi du miracle exclu qui dit que la post-condition *false* ne peut pas être établie par un pro-

gramme qui termine, et la loi de continuité qui dit que l'indéterminisme d'un programme doit être borné.

En 1971 Dijkstra et Wirth ont proposé des méthodes de dérivation de programmes par raffinements successifs [Dij 71, Wir 71]. A la même époque, d'autres méthodes de transformation de programmes ont été proposées [Ger 75].

Plus tard, Back a formalisé un calcul qui permet de dériver un programme en partant de sa spécification [Bac 80, Bac 88]. Il traite les spécifications comme des programmes exécutables dans un univers idéalisé, bien que dans certains cas celles-ci ne soient pas exécutables par une machine réelle. L'idée principale est de traiter spécifications et programmes dans le même cadre, sans les différencier. Il propose une relation de raffinement qui permet de conduire le processus de développement pas à pas jusqu'à l'obtention d'un programme qui est une implantation correcte de la spécification initiale.

D'autres travaux indépendants ont aboutit à des notions semblables [Morr 87, Morg 88, Abr 96-1]. Des idées importantes telles que le raffinement algorithmique qui permet de construire un algorithme complexe par petit pas ; le raffinement des données qui permet de changer l'espace d'états sur lequel sont définies les variables du programme ; ou l'utilisation de spécifications qui ne respectent pas les lois du miracle exclu et de continuité ont été développées ici.

Cette évolution a transformé l'approche initiale de vérification de programmes en une approche très différente de dérivation de programmes par raffinements. Dans le premier cas, le programme est construit d'abord, puis sa correction par rapport à sa spécification est vérifiée. Dans le deuxième cas, la spécification est donnée d'abord, puis un programme correct vis-à-vis de celle-ci est dérivé.

1.2 Vérification de Programmes

La vérification de programmes est une technique qui permet de vérifier si un programme est correct vis-à-vis de sa spécification. Ce problème peut être divisé en deux parties. D'un coté, il faut vérifier que le programme réalise bien sa spécification. D'un autre coté, il faut vérifier qu'il termine, c'est à dire qu'au bout d'une période de temps finie l'exécution s'arrête.

Un programme est partiellement correct s'il réalise bien sa spécification. Un programme est totalement correct si, en plus, il termine.

1.2.1 Logique de Hoare

Hoare a proposé une méthode axiomatique de vérification de programmes [Hoa 69]. Il suggère pour cela d'écrire la spécification d'un programme sous la forme d'un couple de formules de la logique classique de premier ordre.

Hoare introduit une notation qui permet d'établir la relation qui existe entre un programme et sa spécification. Soit Q un programme. Soit x une variable, ou une liste de variables distinctes, qui représente l'état du programme. Soit $P(x)$ une formule où x est libre. Elle dénote les états possibles du programme avant son exécution. Soit $R(x)$ une autre formule où x est libre. Elle dénote les états

possibles du programme après son exécution. La formule

$$\{P(x)\}Q\{R(x)\}$$

est interprétée comme suit : si avant l'exécution du programme Q la formule $P(x)$ est satisfaite, alors la formule $R(x)$ sera satisfaite à la fin de son exécution. Il est important de noter que la problématique liée à la terminaison d'un programme n'est pas traitée. La méthode permet de vérifier seulement la correction partielle d'un programme.

Exemple 1.1. Formules bien formées écrites en logique de Hoare [1] .

1. $\{true\}\ x := 1\ \{x > 0\}$
2. $\{x > 10\}\ x := x + 1\ \{x > 11\}$
3. $\{x = 0 \wedge y = 10\}\ x := x + y\ \{x > 0\}$
4. $\{x > 0 \wedge n > 0\}\ while\ n > 0\ do\ x := x + 1\ end\ \{x > 0\}$
5. $\{true\}\ x := y\ \{x > 0\}$
6. $\{true\}\ Q\ \{false\}$

Ici, $true$ et $false$ sont des formules particulières. Soit Σ un ensemble qui représente l'espace d'états. $true$ est une formule qui est satisfaite par tous les états dans Σ. Elle dénote la totalité de l'espace d'états. $false$ est une formule qui n'est satisfaite par aucun état dans Σ. Elle dénote l'ensemble vide.

Le système de déduction associé est formé d'un schéma d'axiome et d'un ensemble de règles d'inférence qui permettent de prendre en compte les instructions principales d'un langage de programmation impératif : l'affectation, la séquence, l'instruction conditionnelle et la boucle.

Ces règles sont présentées ci-après [2] :

$$\frac{}{\{P[x \leftarrow E]\}\ x := E\ \{P\}} \quad \text{Schéma d'axiome}$$

$$\frac{\{P\}\ Q\ \{R\} \qquad R \Rightarrow S}{\{P\}\ Q\ \{S\}} \quad \text{Conséquence 1}$$

$$\frac{\{P\}\ Q\ \{R\} \qquad S \Rightarrow P}{\{S\}\ Q\ \{R\}} \quad \text{Conséquence 2}$$

$$\frac{\{P \wedge B\}\ Q\ \{R\} \qquad P \wedge \neg B \Rightarrow R}{\{P\}\ if\ B\ then\ Q\ end\ \{R\}} \quad \text{if 1}$$

[1] Les quatre premières formules sont valides ; les deux dernières ne le sont pas.

[2] $P[x \leftarrow E]$ dénote la formule obtenue en substituant toutes les occurrences libres de x dans P par E, et en évitant toute capture de variable.

$$\frac{\{P \wedge B\}\ Q1\ \{R\} \qquad \{P \wedge \neg B\}\ Q2\ \{R\}}{\{P\}\ if\ B\ then\ Q1\ else\ Q2\ end\ \{R\}} \quad \text{if 2}$$

$$\frac{\{P\}\ Q1\ \{R1\} \qquad \{R1\}\ Q2\ \{R\}}{\{P\}\ Q1; Q2\ \{R\}} \quad \text{séquence}$$

$$\frac{\{P \wedge B\}\ Q\ \{P\} \qquad P \wedge \neg B \Rightarrow R}{\{P\}\ while\ B\ do\ Q\ end\ \{R\}} \quad \text{while}$$

Les quatre premières formules de l'exemple 1.1 sont valides. Elles peuvent être déduites en utilisant le système de déduction précédant et les axiomes de l'arithmétique. Les deux dernières formules ne le sont pas. Elles ne peuvent pas être prouvées dans ce système.

Les règles que nous avons présentées ne tiennent pas compte du fait que les expressions utilisées dans un programme ou dans sa spécification doivent être bien définies. Par exemple, l'expression x/y a un sens seulement si $y \neq 0$. Il est donc nécessaire de prendre en compte cette problématique séparément.

Le schéma d'axiome permet de traiter l'instruction d'affectation. Il exprime l'idée que si une formule $P(x)$ doit être satisfaite après l'affectation d'une expression E à la variable x, alors il est nécessaire que la formule $P(E)$ soit satisfaite avant la réalisation de l'affectation.

La règle de conséquence 1 permet de montrer qu'un programme qui établit une post-condition R établit aussi une post-condition S à condition que R soit plus fort que S, c'est dire si $R \Rightarrow S$.

La règle de conséquence 2 dit que si un programme est correct par rapport à la spécification P, R alors, si S dénote un sous-ensemble des états dénotés par P, ce même programme est correct par rapport à la spécification S, R.

Ces deux règles permettent de combiner des formules de la logique de Hoare avec des formules de la logique classique.

La règle *while* introduit la notion *d'invariant de boucle*. Elle dit que pour prouver la correction partielle d'une boucle il suffit de trouver une propriété P invariante dans chaque pas de boucle et tel que si la condition de boucle B n'est pas satisfaite alors la post-condition est satisfaite. La règle n'inclue pas la notion de variant car cette notion est directement liée à la preuve de terminaison du programme. Elle permet donc de montrer des formules en logique de Hoare pour des programmes qui ne terminent pas. La quatrième formule de l'exemple 1.1 illustre ce phénomène.

Les autres règles permettent de prendre en compte la séquence et l'instruction conditionnelle.

1.2.2 Plus Faible Pré-Condition

Dijkstra a proposé une méthode qui permet de vérifier la correction totale d'un programme [Dij 76].

Pour un programme S et une post-condition R, il propose de calculer une condition qui caractérise tous les états initiaux qui garantissent la terminaison de S, et qui laissent le système dans un état qui satisfait R. Cette condition est appelée la plus faible pré-condition de R par rapport à S.

Il définit pour chaque programme S une fonction, notée WP, qui associe à chaque post-condition R un prédicat qui correspond à la plus faible pré-condition. Ainsi, la plus faible pré-condition de R par rapport à S est $WP(S, R)$.

Cette fonction permet de voir un programme comme un transformateur de prédicats qui prend en entrée la post-condition à établir, et qui rend la plus faible pré-condition correspondante.

Dijkstra formule cinq propriétés importantes que tout programme *faisable* doit satisfaire. C'est à dire, qu'un programme qui ne satisfait pas une de ces propriétés ne peut pas exister [3] :

Propriété 1.1. *Loi du miracle exclu.*
Soit S un programme quelconque. Nous avons :

$$WP(S, false) = false$$

La propriété 1.1 dit qu'il n'existe pas de programme réel qui permet d'établir n'importe quelle post-condition. Elle dit, en particulier, qu'aucun programme réel ne peut établir une post-condition fausse.

Propriété 1.2. *Soit S un programme. Soit Q et R deux post-conditions. Pour tout état initial, nous avons.*

$$Si \quad Q \Rightarrow R \quad alors \quad WP(S, Q) \Rightarrow WP(S, R)$$

Si $Q \Rightarrow R$ est satisfait, tout état initial qui permet d'établir la post-condition Q permet d'établir la post-condition R.

Propriété 1.3. *Soit S un programme. Soit Q et R deux post-conditions. Pour tout état initial, nous avons :*

$$WP(S, Q) \wedge WP(S, R) \quad = \quad WP(S, Q \wedge R)$$

Tout état initial qui permet d'établir les post-conditions Q et R permet d'établir la post-condition $Q \wedge R$. D'autre part, tout état initial qui permet d'établir la post-condition $Q \wedge R$ permet aussi d'établir Q et d'établir R.

Propriété 1.4. *Soit S un programme. Soit Q et R deux post-conditions. Pour tout état initial, nous avons :*

$$WP(S, Q) \vee WP(S, R) \quad \Rightarrow \quad WP(S, Q \vee R)$$

[3] Nous présentons ces 5 propriétés tel qu'elles apparaissent dans [Dij 76].

Tout état initial qui permet d'établir la post-condition Q ou la post-condition R permet d'établir la post-condition $Q \vee R$. L'implication dans l'autre sens n'est pas valide.

Propriété 1.5. *Loi de continuité. Soit S un programme. Soit $C_0, C_1, C_2, ...$ une suite infinie de prédicats telle que $C_i \Rightarrow C_{i+1}, \quad i \geq 0$. Pour tout état initial, nous avons :*

$$WP(S, \exists_i \,.\, (i \geq 0 \wedge C_i)) \quad = \quad \exists_j \,.\, (j \geq 0 \wedge WP(S, C_j))$$

La propriété 1.5 établit que le non déterminisme d'un programme qui termine doit être borné. C'est à dire, qu'il n'est pas possible de construire un programme qui termine et qui peut produire, pour une même entrée, un nombre infini de résultats différents.

Pour pouvoir calculer la WP d'un programme, il est nécessaire d'associer une fonction à chaque instruction du langage. Nous allons définir cette fonction pour les intructions fondamentales d'un langage de programmation impératif : l'affectation, la séquence, l'instruction conditionnelle et la boucle.

Nous supposons que R est la post-condition à établir.

- *sans effet.*
  skip est une instruction qui termine toujours et qui n'a aucun effet sur les variables du programme. Par conséquent, le prédicat R doit être satisfait avant l'exécution du programme. Sa plus faible pré-condition est :

$$WP(skip, R) \quad \triangleq \quad R$$

- *abort.*
  abort est une instruction qui ne termine pas. Elle ne peut établir aucune post-condition qui dénote un état valide. Sa plus faible pré-condition est :

$$WP(abort, R) \quad \triangleq \quad false$$

- *Affectation.*
  $x := E$ est une instruction qui termine et qui affecte l'expression E à la variable x. Nous supposons ici que E est un terme qui a un sens et qui n'est pas defini récursivement. C'est à dire, que l'expression E est évaluée dans son domaine de définition. Sa plus faible pré-condition est :

$$WP(x := E, R) \quad \triangleq \quad R[x \leftarrow E]$$

  Exemple 1.2. Calcul de la WP pour l'instruction d'affectation.

$$WP(x := x - 1, x > 0) \;=\; (x > 0)[x \leftarrow x - 1] \;=\; x - 1 > 0 \;=\; x > 1$$

- *Séquence.*
  Soient $S1$ et $S2$ deux programmes qui terminent. $S1; S2$ est une instruction qui termine et qui exécute d'abord $S1$ puis $S2$. Sa plus faible pré-condition est :

$$WP(S1; S2, R) \quad \triangleq \quad WP(S1, WP(S2, R))$$

Exemple 1.3. Calcul de la WP pour la séquence.

$$WP(x := x-1\ ;\ x := x-1, x > 0) =$$
$$WP(x := x-1, WP(x := x-1, x > 0)) =$$
$$WP(x := x-1, x > 1) = x > 2$$

– *Garde.*
$G \Longrightarrow S$ est une instruction qui n'est pas utilisée seule, elle sert comme élément de base pour fabriquer d'autres instructions plus complexes. Sa plus faible pré-condition est :

$$WP(G \Longrightarrow S, R) \quad \triangleq \quad G \Rightarrow WP(S, R)$$

– *Choix.*
$if\ G_1 \Longrightarrow S_1 \,[]\, G_2 \Longrightarrow S_2 \,[]\, ... \,[]\, G_n \Longrightarrow S_n\ fi$, est une instruction qui termine si et seulement si une des gardes $G_1, G_2, ..., G_n$ est satisfaite et $S1$, $S2$, ..., S_n sont des programmes qui terminent. Elle permet de raisonner sur des programmes non déterministes car si plusieurs gardes sont satisfaites, une d'entre elles est choisie par le programme d'une façon non déterministe. Sa plus faible pré-condition est :

$$\begin{aligned} WP(if\ G_1 \Longrightarrow S_1 \,[]\ & G_2 \Longrightarrow S_2 \,[]\, ... \,[]\, G_n \Longrightarrow S_n\ fi, R) \quad \triangleq \\ & (G_1 \vee \cdots \vee G_n) \wedge \\ & WP(G_1 \Longrightarrow S_1, R)) \wedge \cdots \wedge WP(G_n \Longrightarrow S_n, R)) \end{aligned}$$

– *Instruction Conditionnelle.*
$if\ G\ then\ S_1\ else\ S_2\ end$, est un cas particulier de choix. Elle peut s'écrire comme suit : $if\ G \Longrightarrow S_1 \,[]\, \neg G \Longrightarrow S_2\ fi$

Exemple 1.4. Calcul de la WP pour l'instruction conditionnelle.

$$WP(if\ x > 0\ then\ y := 10/x\ else\ skip\ end\ , y > 0) =$$
$$(x > 0 \Rightarrow WP(y := 10/x, y > 0) \wedge (\neg(x > 0) \Rightarrow WP(skip, y > 0) =$$
$$(x > 0 \Rightarrow 10/x > 0) \ \wedge\ (x \leq 0 \Rightarrow y > 0)$$

– *Choix Itératif.*
$do\ G_1 \Longrightarrow S_1 \,[]\, G_2 \Longrightarrow S_2 \,[]\, ... \,[]\, G_n \Longrightarrow S_n\ od$, est une instruction qui permet de raisonner aussi sur des programmes non déterministes. Dans ce cas, il n'est pas nécessaire qu'une des gardes soit satisfaite. Ceci doit d'ailleurs se produire pour garantir la terminaison du programme. Si aucune garde n'est satisfaite, l'instruction termine sans modifier aucune variable du programme. Le calcul de la plus faible pré-condition de l'itération n'est pas trivial. Pour cela il est nécessaire d'introduire la notion de variant. Nous ne donnerons pas ici sa définition ([Dij 76], page 35).
– *Boucle.*
La boucle $while\ G\ do\ S\ end$, est un cas particulier de choix itératif. Elle peut s'écrire comme suit : $do\ G \Longrightarrow S\ od$.

Les propriétés 1.1 à 1.5 sont satisfaites par chacune des instructions du langage précédemment décrites à l'exception de l'instruction gardée qui ne satisfait pas la loi du miracle exclu.

1.2.3 Définition de la Notion de Correction Totale

La plus faible pré-condition permet de définir formellement la notion de correction totale d'un programme par rapport à une spécification, ce qui est noté $P < S > Q$.

Définition 1.1. Correction totale d'un programme.
Le programme S est totalement correct vis-à-vis de la spécification P, Q, si et seulement si P implique la plus faible pré-condition de Q par rapport à S. Formellement :

$$P < S > Q \quad \Leftrightarrow \quad P \Rightarrow WP(S, Q) \qquad \text{Déf 1.1}$$

Ici, la $WP(S, Q)$ dénote tous les états qui garantissent la terminaison de S et l'établissement de la post-condition Q. En conséquence, l'ensemble d'états dénoté par P doit être inclus dans l'ensemble d'états dénoté par $WP(S, Q)$.

1.3 Dérivation de Programmes

Le processus de vérification de la correction d'un programme suppose que le programme soit construit d'abord, puis que sa correction soit vérifiée. Cette méthode engendre deux difficultés importantes :

1. Les preuves générées sont très difficiles à réaliser.
2. Il est impossible d'exprimer des propriétés abstraites sur le comportement du programme car les variables qu'il utilise sont de bas niveau. Elles représentent des structures de données directement implantables.

Une approche très différente a été proposée dans les années 70 à 80 [Dij 71, Ger 75, Wir 71]. L'idée ici est de partir d'une spécification abstraite et de la raffiner peu à peu jusqu'à l'obtention d'un programme totalement correct vis-à-vis de celle-ci. Au début des années 80, Back a proposé une formalisation de cette méthode [Bac 80, Bac 88]. D'autres travaux ont permit de développer cette idée [Aba,Lam 88, Abr 96-1, But 96, Morg 88, Morg 90, Morr 87] et d'aboutir aux notions de raffinement algorithmique et de raffinement de données.

Cette approche offre deux possibilités bien distinctes : nous pouvons soit regarder les spécifications comme des programmes, soit associer des spécifications aux programmes.

Dans le premier cas les objets traités sont les programmes. Nous devons donc disposer d'une relation de raffinement entre programmes.

Dans le deuxième cas les objets traités sont les spécifications. Nous devons donc disposer d'une relation de raffinement entre spécifications.

Nous présenterons les deux méthodes, mais nous utiliserons seulement la deuxième pour exprimer les idées théoriques de notre travail. Nous faisons ce choix parce que nous préférons raisonner sur des spécifications plutôt que sur des programmes.

1.4 Raffinement de Programmes

Cette méthode consiste à regarder les spécifications comme des programmes abstraits qui peuvent s'exécuter dans un univers idéalisé [Abr 96-1, Bac 88, Morg 88]. Dans ce cas, le processus de développement permet de transformer par petit pas un programme abstrait en un programme concret qui peut s'exécuter dans une machine réelle.

Ces transformations sont effectuées à l'aide d'une relation de raffinement, $\sqsubseteq^p$ [4], qui permet de remplacer le programme abstrait par un autre programme qui est totalement correct vis-à-vis de toutes les spécifications pour lesquelles le programme abstrait est totalement correct. C'est à dire, que si le programme S_1 est raffiné par S_2, et si S_1 est totalement correct vis-à-vis d'une spécification quelconque, alors S_2 est aussi totalement correct vis-à-vis de cette spécification. Cette propriété peut s'exprimer formellement comme suit [Bac 88, Bac 89].

Propriété 1.6. *Préservation de la correction totale.*
Soit S_1 et S_2 deux programmes quelconques. Nous avons :

$$S_1 \sqsubseteq^p S_2 \quad \Leftrightarrow \quad \forall_{P,Q} \cdot (P < S_1 > Q \Rightarrow P < S_2 > Q)$$

Par conséquent, l'ensemble de spécifications pour lesquelles S_1 est totalement correct est inclus dans l'ensemble de spécifications pour lesquelles S_2 est totalement correct.

La relation de raffinement entre programmes doit être réflexive car si un programme est remplacé par lui-même, la propriété de correction totale est préservée. Elle doit aussi être transitive puisque si un programme est raffiné par un deuxième qui a son tour est raffiné par un troisième, ce dernier doit être un raffinement du premier. Formellement :

Propriété 1.7. *Réflexivité de la relation de raffinement.*
Soit S un programme. Nous avons :

$$S \sqsubseteq^p S$$

Propriété 1.8. *Transitivité de la relation de raffinement.*
Soit S_1, S_2, S_3, trois programmes. Nous avons :

$$S_1 \sqsubseteq^p S_2 \quad \wedge \quad S_2 \sqsubseteq^p S_3 \quad \Rightarrow \quad S_1 \sqsubseteq^p S_3$$

La propriété de transitivité est importante car elle permet le développement par petit pas.

La notion de plus faible pré-condition permet de définir la relation de raffinement de telle sorte que les trois propriétés énoncées soient satisfaites [Bac 88, Morg 88, Morg 90, Morr 89] :

[4] Nous utiliserons le symbole $\sqsubseteq^p$ pour désigner la relation de raffinement entre programmes.

Définition 1.2. Relation de Raffinement entre Programmes.
Le programme S_1 est raffiné par le programme S_2 si et seulement si pour tout prédicat R, la WP de S_1 implique la WP de S_2. Formellement :

$$S_1 \sqsubseteq^p S_2 \quad \Leftrightarrow \quad \forall_R \,.\, (WP(S_1, R) \quad \Rightarrow \quad WP(S_2, R)) \qquad \text{Déf 1.2}$$

L'idée de regarder une spécification comme un programme engendre la nécessité de calculer la WP d'une spécification. Nous présentons ici la forme simple proposée dans [Morg 88]. Soit $[P, Q]$ une instruction de spécification [5] où P correspond à la pré-condition et Q à la post-condition. La plus faible pré-condition de la spécification $[P, Q]$ est :

$$WP([P, Q], R) \quad \triangleq \quad P \wedge \forall_x \,.\, (Q \Rightarrow R)$$

Une forme plus complexe, due à la nécessité d'écrire des spécifications utilisant des états avant et après est proposée dans [Morg 88]. Cette formule viole la loi du miracle exclu, propriété 1.1, car

$$WP([true, false], false) = true$$

La validité de la méthode n'est cependant pas mise en cause par ce phénomène car un programme miraculeux ne peut être raffiné que par un autre programme miraculeux. Ceci veux dire qu'il est impossible de construire un programme réel qui raffine une spécification miraculeuse car, d'après la définition 1.2, si ce programme existe il doit établir la post-condition $false$ ce qui est impossible.

Afin de pouvoir utiliser le raffinement dans le développement de programmes, une quatrième propriété doit être satisfaite. L'objectif est de pouvoir remplacer une partie d'un programme par un raffinement de seulement cette partie pour obtenir un raffinement du programme initial. Ceci peut être exprimé formellement comme suit :

Propriété 1.9. *Monotonie de la relation de raffinement.*
Soit T un programme. Soit $P[T]$ un programme dont une partie est T. Soit T_1 un programme qui raffine T. Nous avons :

$$T \sqsubseteq^p T_1 \quad \Rightarrow \quad P[T] \sqsubseteq^p P[T_1]$$

Une conséquence importante de cette propriété est qu'il est possible de raffiner séparément une partie d'un programme pour obtenir un raffinement correct de celui-ci. Il sera nécessaire de montrer que chaque instruction du langage préserve cette propriété [Abr 96-1, Bac,Wri 90, Morg 90].

1.5 Raffinement de Spécifications

L'idée directrice ici est d'associer une spécification à chaque instruction du langage et à chaque programme. Les objets raffinés ne sont pas les programmes

[5] La notation [P,Q] est utilisée pour désigner une spécification vue comme un programme.

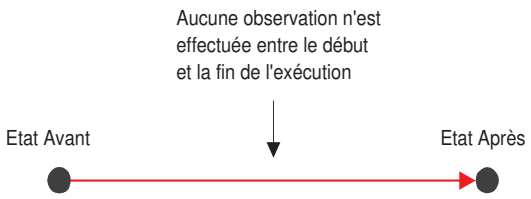

Figure 1.1 – Spécification Elémentaire

mais les spécifications. La spécification associée à une instruction du langage définit la sémantique de l'instruction. Dans ce contexte, le processus de développement permet de transformer une spécification abstraite en un programme qui peut s'exécuter dans une machine réelle. La spécification associée au programme obtenu raffine la spécification abstraite du départ.

Nous donnerons ici une définition de la relation de raffinement entre spécifications qui n'utilise pas la notion de plus faible pré-condition, mais qui satisfait les propriétés de réflexibilité, transitivité et monotonie énoncées plus haut.

1.5.1 Spécification Elémentaire

Pour décrire le comportement des programmes que nous allons étudier, nous allons adopter la technique de spécification proposée par Hoare : pré-condition, post-condition [Hoa 85], et nous allons utiliser des prédicats avant-après [Jon 86, Lam,Sha 90, Spivey 92].

Une spécification élémentaire sera donc formée par deux formules $P(x)$ et $Q(x, x')$, où x, x' sont libres. La première formule dépend seulement de x car cette variable représente l'état avant l'exécution du programme associé à la spécification. La deuxième dépend de x et de x' car elle relie les états avant aux états après l'exécution de ce même programme. Un observateur externe voit uniquement le début et la fin de l'exécution. Aucune observation n'est possible pendant l'exécution du programme. La figure 1.1 illustre cette idée.

Ces deux formules peuvent être reliées comme suit :

$$P(x) \;\Rightarrow\; Q(x, x') \tag{1.1}$$

Cette dernière formule capture l'idée intuitive que nous avons d'un programme élémentaire. Si la pré-condition est satisfaite lorsqu'il est lancé, alors le programme modifie son état de telle sorte que la post-condition soit satisfaite. Par contre, si la pré-condition n'est pas satisfaite lors du démarrage, aucune garantie n'est donnée sur son comportement : tous les comportements sont admis.

La formule 1.1 dénote l'ensemble de transitions acceptées pour un programme qui implante la spécification qu'elle décrit.

Exemple 1.5. Spécifications Elémentaires.

– Un programme qui reçoit en entrée deux entiers et qui rend leur addition a pour spécification :

$$y_1 \in INT \wedge y_2 \in INT \;\Rightarrow\; z' = y_1 + y_2$$

– Un programme qui reçoit en entrée deux nombres naturels et qui rend leur division a pour spécification :

$$y_1 \in NAT \wedge y_2 \in NAT \wedge y_2 > 0 \;\Rightarrow\; z' = y_1 / y_2$$

– Un programme qui incrémente une variable qui dénote un nombre naturel a pour spécification :

$$x \in NAT \;\Rightarrow\; x' = x + 1$$

1.5.2 Faisabilité d'une Spécification Elémentaire

Une spécification n'est pas forcément faisable. C'est à dire, qu'il existe des spécifications pour lesquelles il est impossible de fabriquer un dispositif qui les satisfait.

Définition 1.3. Faisabilité d'une spécification.
Une spécification élémentaire est faisable si pour tout état qui satisfait la pré-condition il existe un état qui satisfait la post-condition. Formellement :

$$\forall_x \,.\, (P(x) \quad \Rightarrow \quad \exists_{x'} \,.\, Q(x, x')) \qquad \text{Déf 1.3}$$

1.5.3 Non Déterminisme Interne

Une spécification peut être non déterministe car la post-condition Q dénote une relation. Si la spécification est faisable, à chaque état avant elle associe un ou plusieurs états après. Ce type de non déterminisme peut être qualifié d'interne car le concepteur est libre de le réduire pendant le processus de développement. Il devra d'ailleurs le faire à l'aide de la relation de raffinement. L'environnement n'a aucune influence sur les choix qui sont faits.

1.5.4 Relation de Raffinement entre Spécifications

Le but de la relation de raffinement que nous allons définir, $\sqsubseteq$, est de permettre au concepteur de construire un programme réel en partant d'une spécification abstraite du programme. L'idée sous-jacente étant de pouvoir remplacer la spécification initiale par la spécification associée au programme obtenu [6] de telle sorte qu'un observateur externe ne perçoive aucune différence dans le comportement de ces deux objets.

[6] Cette spécification correspond à l'abstraction fonctionnelle du programme.

Ceci veut dire que si nous avons la possibilité d'appliquer un jeu de tests au programme, les résultats produits peuvent être obtenus en appliquant les mêmes tests à la spécification initiale. La spécification initiale peut admettre beaucoup plus de comportements que ceux admis par le programme car elle est indéterministe. Par contre, tout comportement admis par le programme est nécessairement un comportement admis par sa spécification.

Une première condition pour pouvoir effectuer ce remplacement est que le programme réel fonctionne dans tous les états initiaux où la spécification donnée au départ fonctionne. En d'autres mots, l'ensemble d'états dénoté par la pré-condition de la spécification abstraite doit être inclus dans l'ensemble d'états dénoté par la pré-condition de la spécification concrète [7] . En conséquence, lors du processus de développement les pré-conditions sont affaiblies. Ceci peut être exprimé formellement comme suit [8] :

Propriété 1.10. *Affaiblissement de la Pré-condition.*
Soit S_1 et S_2 deux spécifications. Si S_1 est raffinée par S_2 alors la pré-condition de S_1 est affaiblie. Formellement :

$$S_1 \sqsubseteq S_2 \quad \Rightarrow \quad \forall_x \bullet (P_1(x) \Rightarrow P_2(x))$$

Pendant le processus de développement le concepteur doit introduire les détails qui ont été laissés de côté dans la spécification abstraite. Ces détails réduisent l'indéterminisme interne de celle-ci.

Prenons par exemple la spécification : $true \Rightarrow z' > 10$. La post-condition indique seulement que le résultat doit être plus grand que 10. Un programme qui rend, disons, 50 comme résultat est correct vis-à-vis de cette spécification. Il y a une infinité de programmes donnant des résultats différents qui sont aussi corrects.

Pendant le processus de développement, le concepteur doit faire des choix qui conduisent le programme à rendre un résultat plutôt qu'un autre. Pour cela il doit renforcer la post-condition. La relation de raffinement doit donc permettre l'introduction de détails dans une spécification en renforçant la post-condition initiale. Ceci peut s'exprimer formellement comme suit :

Propriété 1.11. *Renforcement de la Post-Condition.*
Soit S_1 et S_2 deux spécifications. Si S_1 est raffinée par S_2 alors sous la pré-condition de S_1 la post-condition de S_1 est renforcée. Formellement :

$$S_1 \sqsubseteq S_2 \quad \Rightarrow \quad \forall_{x,x'} \bullet (P_1(x) \Rightarrow (Q_2(x,x') \Rightarrow Q_1(x,x')))$$

L'ensemble de transitions dénoté par la post-condition de la spécification concrète qui correspondent à des transitions dont l'état avant satisfait la pré-condition abstraite est inclus dans l'ensemble de transitions dénoté par la post-condition de la spécification abstraite.

[7] Nous allons appeler *abstraite* la spécification donnée initialement et nous allons appeler *concrète* la spécification qui est directement associée au programme. Cette dernière correspond à la fonction qui décrit le comportement du programme, voir section 1.5.5.

[8] Par convention S_i est une spécification de la forme $P_i(x) \Rightarrow Q_i(x,x')$ et i est un nombre naturel.

Les transitions de la spécification concrète dont l'état avant ne satisfait pas la pré-condition abstraite sont acceptées car dans ce cas aucune garantie n'est demandée sur le comportement du programme.

Nous allons définir la relation de raffinement entre spécifications comme la conjonction de ces deux dernières formules [9] .

Définition 1.4. Relation de raffinement entre spécifications.
Soit S_1 et S_2 deux spécifications. S_1 est raffinée par S_2 si et seulement si la pré-condition de S_1 est affaiblie, et si pour tous les états qui satisfont la pré-condition de S_1 la post-condition de S_1 est renforcée. Formellement :

$$S_1 \sqsubseteq S_2 \quad \Leftrightarrow \quad \forall_{x,x'} \cdot (P_1(x) \Rightarrow P_2(x) \wedge (Q_2(x,x') \Rightarrow Q_1(x,x'))) \quad \text{Déf 1.4}$$

L'ensemble d'états dénoté par $P_1(x)$ est inclus dans l'ensemble d'états dénoté par $P_2(x)$. L'ensemble de transitions dénoté par S_2 est inclus dans l'ensemble de transitions dénoté par S_1.

Il existe encore une condition supplémentaire qui doit être satisfaite pour que la définition du raffinement soit complète : les spécifications S_1 et S_2 doivent être des formules bien définies. C'est à dire, que les expressions qu'elles contiennent sont bien définies. Dans la suite du document, nous allons supposer que c'est toujours le cas. Le problème de la bonne définition de termes et de formules dans le cadre de la méthode B est traité dans [Beh,Bur,Mey 98, Bur 99] :

Cette relation est réflexive et transitive. Elle permet donc le développement par petit pas.

Propriété 1.12. *Réflexivité de la relation de raffinement entre spécifications. Soit S une spécification. Nous avons :*

$$S \sqsubseteq S$$

Propriété 1.13. *Transitivité de la relation de raffinement entre spécifications. Soit S_1, S_2, S_3, trois spécifications. Nous avons :*

$$S_1 \sqsubseteq S_2 \wedge S_2 \sqsubseteq S_3 \quad \Rightarrow \quad S_1 \sqsubseteq S_3$$

La définition 1.4 permet de montrer aisément les propriétés 1.10 à 1.13.

1.5.5 Spécifications Associées aux Instructions

Dans cette section nous allons présenter le langage de programmation que nous utilisons. Il s'agit essentiellement du langage proposé dans la méthode B. Les expressions utilisées ne sont pas définies récursivement. Elles sont donc calculables lorsqu'elles sont évaluées dans leurs domaines de définition.

[9] Cette définition est celle donnée dans [Abr 96-1], page 503.

Pour pouvoir utiliser la relation de raffinement que nous venons de définir, nous devons associer à chaque instruction du langage une spécification qui donne sa sémantique. Pour cela, nous allons définir une fonction, notée SL, qui permet d'obtenir les spécifications associées aux instructions du langage.

La pré-condition de la spécification produite par cette fonction correspond à la condition de terminaison de l'instruction. Nous faisons ici l'hypothèse que toutes les spécifications utilisées comme paramètres dans l'instruction sont associées à des programmes qui terminent. Par exemple, la fonction SL associe à l'instruction $S_1; S_2$ une pré-condition qui correspond à sa condition de terminaison, mais seulement si S_1 et S_2 sont des spécifications associées à des programmes qui terminent [10]. La post-condition de la spécification produite décrit l'effet de l'instruction [11].

Le langage comporte deux types d'instruction : de spécification et de programmation. Les instructions de spécification sont utilisées pour décrire le comportement du programme. Elles doivent être raffinées pour produire le code souhaité. Les instructions de programmation ne sont pas raffinées ; elles sont utilisées pour implanter les spécifications. La fonction SL est définie comme suit :

Instructions de spécification

- *Instruction Gardée.*
  $G \Longrightarrow S_1$ est une instruction qui termine si et seulement si la garde G implique $P_1(x)$. Sa post-condition est $Q_1(x, x')$ mais seulement pour les états qui satisfont la garde. Formellement :

$$SL(\, G_1 \Longrightarrow S_1 \,) \quad \triangleq \quad (G_1(x) \Rightarrow P_1(x)) \;\Rightarrow\; G_1(x) \wedge Q_1(x, x') \tag{1.2}$$

  Cette instruction n'est pas utilisée seule pour écrire des programmes. Elles sert à définir d'autres instructions. Elle est aussi très utile pour spécifier les évènements qu'un systèmes permet d'observer.

- *Choix.*
  $if\ S_1 \,[]\, S_2\ fi$ est une instruction qui termine si et seulement si les deux pré-conditions sont satisfaites. Elle laisse le système dans un état où une des deux post-conditions est satisfaite. Formellement :

$$SL(\, if\ S_1 \,[]\, S_2\ fi \,) \quad \triangleq \quad P_1(x) \wedge P_2(x) \;\Rightarrow\; Q_1(x, x') \vee Q_2(x, x') \tag{1.3}$$

[10] Cette condition correspond au prédicat $trm(S)$ qui décrit la condition de terminaison d'une substitution généralisée dans la méthode B, [Abr 96-1], pages : 289 à 290 pour la définition, 308 pour la composition parallèle, 375 pour la séquence, 395 et 396 pour la boucle.

[11] Cette condition correspond au prédicat avant après $prd_x(S)$ d'une substitution généralisée dans la méthode B, [Abr 96-1], pages 292 à 294 pour la définition, 308 pour la composition parallèle, 375 pour la séquence, 395 et 396 pour la boucle.

Instructions de programmation

- *sans effet.*
  skip est une instruction qui termine toujours et qui n'a aucun effet sur les variables du programme.

$$SL(skip) \quad \triangleq \quad true \ \Rightarrow \ x' = x \tag{1.4}$$

- *Affectation.*
  $x := E$ est une instruction qui termine si et seulement si les deux conditions qui suivent sont satisfaites :
  - L'expression E est bien définie. C'est à dire que toutes les fonctions nécessaires à son évaluation sont utilisées dans leurs domaines de définition. E n'est pas définie récursivement.
  - L'expression E peut être correctement évaluée. C'est à dire que le système utilisé pour exécuter le programme est en mesure d'évaluer l'expression E d'une façon telle qu'il n'y ait pas de débordement.

  Nous allons noter le prédicat associé à cette condition $\Delta_{ce}(E)$ [12] .
  Par conséquent, l'instruction $x := E$ termine si et seulement si $\Delta_{ce}(E)$ est satisfait. Elle affecte l'expression E à la variable x.

$$SL(\ x := E\ ) \quad \triangleq \quad \Delta_{ce}(E) \ \Rightarrow \ x' = E \tag{1.5}$$

  Si l'état du programme dépend d'une autre varible x_1, la spécification est :

$$SL(\ x := E\ ) \quad \triangleq \quad \Delta_{ce}(E) \ \Rightarrow \ x' = E \wedge {x_1}' = x_1 \tag{1.6}$$

- *Séquence.*
  $S_1; S_2$ est une instruction qui termine si et seulement si, d'une part, $P_1(x)$ est satisfaite, et d'autre part, si chaque état a' qui satisfait $Q_1(x, a')$ satisfait aussi $P_2(a')$. La post-condition est la composition des deux relations dénotées par S_1 et S_2. Formellement :

$$\begin{array}{lll} SL(\ S1; S2\ ) & \triangleq & P_1(x) \wedge \forall_{a'} \centerdot (Q_1(x, a') \Rightarrow P_2(a')) \\ & & \Rightarrow \\ & & \exists_{b'} \centerdot (Q_1(x, b') \wedge Q_2(b', x')) \end{array} \tag{1.7}$$

- *Instruction Conditionnelle.*
  $if\ G\ then\ S_1\ else\ S_2\ end$ est un cas particulier de choix. Elle peut s'écrire comme suit : $if\ G \Longrightarrow S_1 \ [] \ \neg G \Longrightarrow S_2\ fi$.

[12] La problématique liée à la bonne définition de termes et de formules est étudiée en détaille dans [Beh,Bur,Mey 98, Bur 99].

D'une façon similaire à l'affectation, nous devons tenir compte ici de la capacité du système à évaluer l'expression booléenne associée à la garde. Nous allons noter le prédicat associé à cette condition $\Delta_{cb}(G)$.
En conséquence, la spécification associée à l'instruction conditionnelle est :

$$\begin{array}{ll} SL(& \mathit{if}\ G\ \mathit{then}\ S_1\ \mathit{else}\ S_2\ \mathit{end}\ ) \quad \triangleq \\ & \Delta_{cb}(G(x)) \wedge \\ & (G(x) \Rightarrow P_1(x)) \wedge (\neg G(x) \Rightarrow P_2(x)) \\ & \Rightarrow \\ & (G(x) \wedge Q_1(x,x')) \vee (\neg G(x) \wedge Q_2(x,x')) \end{array} \tag{1.8}$$

– *Boucle.*
while G do S_1 invariant I variant V end est une instruction qui exécute le programme associé à la spécification S_1 tant que la condition de boucle G est satisfaite. Pour définir la terminaison de la boucle, il est nécessaire d'introduire deux notions supplémentaires : le variant et l'invariant de boucle. Le *variant* est une expression qui dénote un élément d'un ensemble bien ordonné et qui décroît à chaque itération. Nous supposons qu'il s'agit d'un nombre naturel. L'*invariant* est un prédicat qui est satisfait au démarrage de la boucle et qui est rétablit à chaque itération.
Une condition suffisante pour qu'une boucle termine est donnée par la conjonction de conditions qui suivent :
– au départ l'invariant est satisfait :
$I(x)$,
– l'invariant implique que le variant est un nombre naturel :
$\forall_x \cdot (I(x) \Rightarrow V(x) \in Nat)$,
– l'invariant et la condition de boucle impliquent la pré-condition de S_1 :
$\forall_x \cdot (I(x) \wedge G(x) \Rightarrow P_1(x))$,
– sous l'invariant et la condition de boucle, S_1 rétablit l'invariant :
$\forall_{x,x'} \cdot (I(x) \wedge G(x) \Rightarrow (Q_1(x,x') \Rightarrow I(x')))$,
– sous l'invariant et la condition de boucle S_1 fait décroître le variant :
$\forall_{x,x'} \cdot (I(x) \wedge G(x) \Rightarrow (Q_1(x,x') \Rightarrow V(x') < V(x)))$.
La conjonction de ces conditions est la pré-condition de la spécification associée à la boucle [13].
De même que pour l'instruction conditionnelle, le système doit être capable d'évaluer correctement la condition de boucle. $I(x) \Rightarrow \Delta_{cb}(G(x))$ doit donc faire partie de la pré-condition.
La boucle termine quand sa condition n'est plus satisfaite. La post-condition est donc $I(x') \wedge \neg G(x')$.

[13] Une étude détaillée de la boucle est fait dans [Abr 96-1], pages 373 à 398.

La spécification complète est :

$$\begin{array}{ll} SL(& \mathit{while}\ G\ \mathit{do}\ S_1\ \mathit{invariant}\ I\ \mathit{variant}\ V\ \mathit{end}\ ) \quad \triangleq \\ & I(x)\ \Rightarrow\ \Delta_{cb}(G(x))\wedge \\ & I(x)\wedge \\ & \forall_x \cdot (I(x)\ \Rightarrow\ V(x) \in Nat) \\ & \forall_x \cdot (I(x) \wedge G(x)\ \Rightarrow\ P_1(x)) \\ & \forall_{x,x'} \cdot (I(x) \wedge G(x)\ \Rightarrow\ (Q_1(x,x')\ \Rightarrow\ I(x'))) \\ & \forall_{x,x'} \cdot (I(x) \wedge G(x)\ \Rightarrow\ (Q_1(x,x')\ \Rightarrow\ V(x') < V(x))) \\ & \Rightarrow \\ & I(x') \wedge \neg G(x') \end{array} \tag{1.9}$$

– *Composition parallèle.*
$S_1(x_1, x_1')\ ||\ S_2(x_2, x_2')$ est une instruction qui termine si et seulement si les deux pré-conditions sont satisfaites. Elle laisse le système dans un état où les des deux post-conditions sont satisfaites. Cette instruction constitue un mécanisme élémentaire de synchronisation. Formellement :

$$SL(\ S_1\ ||\ S_2\ ) \quad \triangleq \quad P_1(x_1) \wedge P_2(x)\ \Rightarrow\ Q_1(x_1, x_1') \wedge Q_2(x_2, x_2') \tag{1.10}$$

Cette instruction fait intervenir deux spécifications définies sur des espaces d'états différents. Il n'y a donc pas d'interaction entre elles. D'un point de vu pratique, elle peut être vue comme la composition des deux spécifications élémentaires dans un ordre quelconque : $S_1; S_2$ ou $S_2; S_1$. L'effet est identique.

– *Choix non Borné .*
var a in $S_1(x, x', a, a')$ *end* est une instruction qui termine si et seulement si pour toutes les valeurs avant de la variable locale a la pré-condition de S_1 est satisfaite. Elle laisse le système dans un état où il existe une valeur avant et une valeur après de la variable locale telle que la post-condition de S_1 est satisfaite. Formellement :

$$SL(\ \mathit{var}\ a\ \mathit{in}\ S_1(x,a)\ \mathit{end}\ ) \ \triangleq\ \forall_a \cdot P_1(x,a)\ \Rightarrow\ \exists_{a,a'} \cdot Q_1(x,a,x',a') \tag{1.11}$$

Cette instruction permet d'introduire une variable locale dans une partie d'un programme.

1.5.6 Spécifications Associées aux Programmes

Afin de pouvoir calculer la spécification d'un programme Pr quelconque écrit dans le langage que nous venons de présenter, nous allons définir une nouvelle fonction $SP(Pr)$ à l'aide de la fonction SL comme suit :

$$SP(\ P_1; P_2\ ) \quad \triangleq \quad SL(SP(P_1); SP(P_2)) \tag{1.12}$$

$$SP(\ if\ G\ then\ P_1\ else\ P_2\ end\ ) \triangleq SL(\ if\ G\ then\ SP(P_1)\ else\ SP(P_2)\ end\ ) \tag{1.13}$$

$$SP(\ while\ G\ do\ P_1\ invariant\ I\ variant\ V\ end\ ) \triangleq SL(\ while\ G\ do\ SP(P_1)\ invariant\ I\ variant\ V\ end\ ) \tag{1.14}$$

$$SP(\ P_1\ ||\ P_2\ ) \triangleq SL(\ SP(P_1)\ ||\ SP(P_2)\ ) \tag{1.15}$$

$$SP(\ var\ a\ in\ P_1\ end\ ) \triangleq SL(\ var\ a\ in\ SP(P_1)\ end\ ) \tag{1.16}$$

Nous n'avons pas donné de définition ni pour l'instruction gardée ni pour le choix parce que ces instructions ne peuvent pas être utilisées seules dans un programme.

La fonction SP établit un lien entre programmes et spécifications. Dans notre contexte, ce lien est indispensable pour définir la notion de correction d'un programme vis-à-vis d'une spécification.

La relation de raffinement entre spécifications doit aussi avoir la propriété de monotonie. Si la spécification associée à un programme T qui est une partie d'un programme Pr est raffinée par la spécification associée à un autre programme T_1, alors la spécification associée à $Pr[T]$ est raffinée par la spécification associée à $Pr[T_1]$. Formellement :

Propriété 1.14. *Monotonie de la relation de raffinement entre spécifications. Soit T un programme. Soit $Pr[T]$ un programme dont une partie est T. Soit T_1 un programme dont la spécification associée $SP(T_1)$ raffine $SP(T)$. Nous avons :*

$$SP(T) \sqsubseteq SP(T_1) \quad \Rightarrow \quad SP(Pr[T]) \sqsubseteq SP(Pr[T_1])$$

Une conséquence importante de cette propriété est qu'il est possible de raffiner séparément une partie d'un programme pour obtenir un raffinement correct de celui-ci. Il sera nécessaire de montrer que chaque instruction du langage préserve cette propriété [Abr 96-1, Bac,Wri 90, Morg 90].

La fonction SP et la notion de raffinement entre spécifications permettent de définir formellement la notion de correction totale d'un programme vis-à-vis d'une spécification, comme suit [14] :

[14] Une autre démarche serait de définir d'abord la notion de correction totale, puis de définir ensuite la relation de raffinement et de montrer que celle-ci préserve la correction totale.

Définition 1.5. Correction totale d'un programme définie à l'aide de la relation de raffinement.
Soit $S \triangleq P(x) \Rightarrow Q(x,x')$ **une spécification quelconque. Soit** Pr **un programme et** $SP(Pr)$ **sa spécification associée.** Pr **est totalement correct vis-à-vis de** S **si et seulement si** $SP(Pr)$ **raffine** S**. Formellement :**

$$P(x) < Pr > Q(x,x') \quad \Leftrightarrow \quad S \sqsubseteq SP(Pr) \qquad \text{Déf 1.5}$$

Exemple 1.6. Dérivation d'un Programme à Partir d'une Spécification.
Nous voulons construire un programme pour échanger les valeurs de deux variables internes. Soit x et y ces deux variables. La spécification du programme est : $true \Rightarrow x' = y \wedge y' = x$.

Nous introduisons d'abord une variable locale a afin de garder la valeur de x pendant l'échange, puis la variable x est affectée avec la valeur de y. Finalement, y est affectée avec la valeur de a. Le processus de développement de ce programme est représenté dans l'arborescence qui suit :

$$\dfrac{\dfrac{\dfrac{true \Rightarrow x' = y \wedge y' = x}{true \Rightarrow x' = y \wedge y' = x}\, var\ a\ in}{\dfrac{true \Rightarrow a' = x \wedge x' = x \wedge y' = y}{a := x}:= \qquad true \Rightarrow x' = y \wedge y' = a}\,;$$

La feuille $true \Rightarrow x' = y \wedge y' = a$ est implantée comme suit :

$$\dfrac{true \Rightarrow x' = y \wedge y' = a}{\dfrac{true \Rightarrow x' = y \wedge y' = y \wedge a' = a}{x := y}:= \qquad \dfrac{true \Rightarrow y' = a \wedge x' = x \wedge a' = a}{y := a}:=}\,;$$

Si nous enlevons les spécifications dans cette arborescence, nous obtenons :

$$\dfrac{\dfrac{\dfrac{}{}\, var\ a\ in}{\dfrac{}{a := x}:= \qquad \dfrac{}{\dfrac{}{x := y}:= \qquad \dfrac{}{y := a}:=}\,;}\,;$$

Ce qui correspond au programme suivant :

$$var\ a\ in \quad a := x; (x := y; y := a) \quad end$$

Dans cet exemple, nous sommes partis de la racine de l'arbre, c'est à dire de la spécification abstraite, et nous avons construit le programme pas à pas. A chaque pas, nous avons introduit une instruction du langage. Nous devons donc prouver que la spécification associée à la nouvelle instruction raffine bien la spécification qu'elle implante. Pour cela, nous devons d'abord calculer la spécification associée à l'instruction à l'aide des formules 1.4 à 1.16, page 22, ensuite nous devons utiliser la définition 1.4, page 24, pour générer les obligations de preuve correspondantes. Finalement, nous devons décharger ces obligations.

Cette démarche permet de contrôler efficacement le processus de preuve car à chaque étape nous sommes guidés par l'intuition que nous avons sur la correction de l'instruction que nous introduisons. Une fois que nous avons prouvé la correction d'un pas de raffinement, nous pouvons le laisser de coté et nous concentrer sur l'introduction de nouvelles instructions. Le processus de développement est terminé lorsque toutes les feuilles de l'arbre sont des instructions élémentaires du langage. Dans notre cas, ce sont des affectations de variables.

Une démarche très différente serait de partir des feuilles de l'arbre et de construire la spécification associée au programme qui en résulte. Dans ce cas, la preuve de raffinement pourrait s'avérer beaucoup plus difficile.

A titre d'exemple nous allons montrer la correction de l'instruction '*var*' et du premier ';' dans l'exemple 1.6.

Pour l'instruction '*var*', d'après la formule 1.11, nous avons :

$$SL(\quad var\ a\ in\quad true\ \Rightarrow\ x' = y \wedge y' = x\quad end)\ =$$

$$\forall_a \,.\, (true)\ \Rightarrow\ \exists_{a,a'} \,.\, (x' = y \wedge y' = x)$$

La formule obtenue peut s'écrire :

$$true\ \Rightarrow\ \exists_{a,a'} \,.\, (x' = y \wedge y' = x)$$

Pour prouver que cette formule est un raffinement correct de la spécification initiale $true\ \Rightarrow\ x' = y \wedge y' = x$, nous utilisons la définition 1.4, page 22. Nous obtenons les deux obligations de preuve qui suivent :

$$\vdash\quad true\ \Rightarrow\ true$$

et

$$\vdash\quad \exists_{a,a'} \,.\, (x' = y \wedge y' = x)\ \Rightarrow\ x' = y \wedge y' = x$$

La preuve de ces deux obiligations est triviale.

Pour l'instruction ';' nous avons, d'après la formule 1.7, page 24 :

$$SL(\quad true\ \Rightarrow\ a' = x \wedge x' = x \wedge y' = y\quad ;\quad true\ \Rightarrow\ x' = y \wedge y' = a)\ =$$

$$\begin{array}{l}(true \wedge \forall_{a'',x'',y''} \,.\, (a'' = a \wedge x'' = x \wedge y'' = y) \Rightarrow true)\ \Rightarrow \\ \exists_{a'',x'',y''} \,.\, ((a'' = x \wedge x'' = x \wedge y'' = y) \wedge (x' = y'' \wedge y' = a''))\end{array}$$

La formule obtenue peut s'écrire :

$$\begin{array}{l}true\ \Rightarrow \\ \exists_{a'',x'',y''} \,.\, ((a'' = x \wedge x'' = x \wedge y'' = y) \wedge (x' = y'' \wedge y' = a''))\end{array}$$

Nous devons prouver que cette formule est un raffinement correct de ($true\ \Rightarrow\ x' = y \wedge y' = x$). Nous procédons de la même façon que dans le cas précédant. Nous obtenons les deux obligations de preuve qui suivent :

$$\vdash\quad true\ \Rightarrow\ true$$

et

$$\vdash \quad \exists_{a'',x'',y''} \cdot ((a'' = x \wedge x'' = x \wedge y'' = y) \wedge (x' = y'' \wedge y' = a'')) \;\Rightarrow\; x' = y \wedge y' = x$$

Ces deux obligations peuvent être déchargées facilement à l'aide du calcul de prédicats.

Cet exemple permet de constater la nécessité d'outils d'aide au développement. Dans ce cas précis, deux outils s'avèrent indispensables. Le premier est un générateur d'obligations de preuve : nous avons effectué 6 raffinements dans l'exemple. Pour chacun d'entre eux nous devons générer les obligations de preuve correspondantes. Le deuxième est un outil de preuve interactif qui permette de décharger les obligations de preuve automatiquement ou d'une façon interactive.

1.6 Utilisation de Spécifications Elémentaires

Les spécifications élémentaires peuvent être utilisées pour spécifier des programmes élémentaires et des procédures. Nous verrons cependant dans le chapitre 2, page 37, qu'elles ne suffissent pas pour spécifier des programmes complexes qui interagissent fortement avec leur environnement.

Nous appelons programme élémentaire un programme qui communique avec son environnement uniquement par des paramètres d'entrée et de sortie. Du point de vue d'un observateur externe, rien d'autre ne se passe entre le moment où l'entrée est reçue et le moment où la sortie est envoyée, voir figure 1.1, page 19.

La spécification d'un programme élémentaire prend la forme [15] :

$$z : (\Delta_{ce}(y) \wedge P(y) \;\Rightarrow\; Q(y, z')) \tag{1.17}$$

y représente les entrées fournies par l'environnement. Cette variable ne peut pas être modifiée par le programme car nous utilisons uniquement le passage de paramètres par valeur [16]. z' représente les sorties. Cette variable est modifiée par le programme ce qui correspond à l'effet de celui-ci sur l'environnement. La pré-condition demande que le programme soit appelée avec un paramètre qui peut être calculé par le système : $\Delta_{ce}(y)$ [17] .

Nous appelons procédure un programme élémentaire qui en plus de ses paramètres peut faire référence à des variables globales.

[15] Dans les spécifications nous devons inclure des formules qui indiquent les variables qui ne doivent pas être modifiées, ce qui est très encombrant. Plusieurs solutions ont été proposées [Abr 96-1, Morg 90]. La méthode B apporte une réponse adéquate puisque dans une substitution généralisée il n'est pas nécessaire de faire référence aux variables non affectées.

Dans la suite, nous allons noter $x : S(x, w, x')$ une spécification qui utilise l'état avant des variables x et w mais qui modifie uniquement la variable x. La post-condition contient la formule $w' = w$.

[16] Il est possible d'écrire des spécifications de programme qui modifient leur paramètres. Ceci correspond au passage de paramètres par référence. Il est nécessaire dans ce cas de restreindre leur utilisation de telle sorte que le phénomène d'*alias* soit évité [Morg 90]. Nous n'utilisons pas ce type de spécification.

[17] Ceci correspond à la notion de *Protected_ Call* en B, [Abr 96-1], page 596.

La spécification d'une procédure prend la forme :

$$x, z : (\Delta_{ce}(y) \wedge P(x, y) \ \Rightarrow \ Q(x, y, x', z')) \tag{1.18}$$

La variable x représentent les variables globales qui peuvent être modifiés par la procédure.

Une procédure doit d'abord être déclarée, puis elle est utilisée à l'intérieur d'un programme d'une façon semblable à une instruction du langage. Nous allons noter l'appel d'une procédure nommée $Proc$ qui a un paramètre d'entrée et un paramètre de sortie de la façon suivante [18] :

$$z \longleftarrow Proc(y)$$

Lors de l'appel, les paramètres formels, y et z, sont remplacés par les paramètres actuels : ceux réellement utilisés. La spécification qui en résulte est utilisée pour montrer la correction du raffinement effectué [19] . Le remplacement ne peut se faire que si toutes les variables utilisées dans les expressions des paramètres actuels sont liées dans la spécification de la procédure. Si cela n'était pas le cas, les variables de la spécification paramétrée doivent être renommées avant d'effectuer le remplacement.

1.7 Raffinement de données.

Nous allons conclure ce chapitre avec la notion de raffinement de données que nous utiliserons dans le chapitre suivant pour définir la relation de raffinement entre modules, voir section 2.2, page 39 [20] .

Le relation de raffinement de la définition 1.4, page 22, ne permet pas de changer l'espace de définition des variables de la spécification initiale. Lors de la description d'un système, il est très utile de pouvoir faire des abstractions sur les objets qu'il traite réellement. De la même façon qu'un langage de programmation nous permet de gérer des objets qui ne sont pas des simples bits : les nombres entiers ou les chaînes de caractères, il est important de pouvoir décrire un système à l'aide d'objets abstraits tel que les ensembles, les fonctions, ou les relations. Ces abstractions permettent d'exprimer le problème d'une façon plus compréhensible par un être humain, mais elles engendrent la nécessité de pouvoir transformer les objets abstraits en objets concrets directement implantables dans un langage de programmation. Le rôle principal de la relation de raffinement que nous allons définir est de permettre ces transformations.

Pour pouvoir transformer un espace d'états, nous devons fournir une relation totale qui établit les liens entre les objets de l'espace abstrait et les objets de l'espace concret [Aba,Lam 88, Abr 96-1]. Elle sera notée $J(x, w)$, où x est une

[18] Nous utiliseront la syntaxe proposée dans la méthode B. Il serait aussi possible d'utiliser la notation $z' \longleftarrow Proc(y)$.

[19] Dans la méthode B, la copie des paramètres d'entrée et l'envoi de résultats lors d'un appel d'une procédure sont effectués implicitement, [Abr 96-1], page 314.

[20] Cette nouvelle relation sert uniquement dans le cadre du raffinement de modules où il est très utile de pouvoir transformer l'espace d'états sur lequel est défini un module.

variable qui dénote les états de l'espace abstrait et w est une variable qui dénote les états de l'espace concret. Nous l'appelleront *invariant de collage* [Abr 96-1].

Une première condition pour pouvoir effectuer un tel raffinement est que la spécification concrète puisse fonctionner dans tous les états où la spécification abstraite fonctionne. Dans d'autres mots, l'image produite par J de l'ensemble d'états dénotés par la pré-condition de la spécification abstraite doit être inclus dans l'ensemble d'états dénotés par la pré-condition de la spécification concrète. Cette condition correspond à l'affaiblissement de la pré-condition abstraite. Ceci peut être exprimé formellement comme suit :

Propriété 1.15. *Affaiblissement de la Pré-condition avec Transformation de l'Espace d'Etats.*
Soit $S_1(x, x')$ et $S_2(w, w')$ deux spécifications définies sur des espaces d'états différents. Soit $J(x, w)$ une relation totale définie entre ces deux espaces. Si S_1 est raffinée par S_2 alors la pré-condition de S_1 est affaiblie. Formellement :

$$S_1 \sqsubseteq_J S_2 \quad \Rightarrow \quad \forall_{x,w} \cdot (P_1(x) \wedge J(x, w) \Rightarrow P_2(w))$$

Le renforcement de la post-condition doit aussi être effectué. Dans ce cas, pour tous les états après produits par la post-condition concrète, il doit exister un état abstrait qui est relié à l'état concret par la relation J et qui est produit par la post-condition abstraite. Ceci peut s'exprimer formellement comme suit :

Propriété 1.16. *Renforcement de la Post-Condition avec Transformation de l'Espace d'Etats.*
Soit $S_1(x, x')$ et $S_2(w, w')$ deux spécifications définies sur des espaces d'états différents. Soit $J(x, w)$ une relation totale définie entre ces deux espaces. Si S_1 est raffinée par S_2 alors la post-condition de S_1 est renforcée. Formellement :

$$\begin{aligned} S_1 \sqsubseteq_J S_2 \quad \Rightarrow \quad \forall_{x,w,w'} \cdot (\quad & P_1(x) \wedge J(x, w) \Rightarrow \\ & (Q_2(w, w') \Rightarrow \exists_{x'} \cdot (Q_1(x, x') \wedge J(x', w')))) \end{aligned}$$

La définition de la relation de raffinement avec transformation de l'espace d'états est la conjonction des deux dernières propriétés. Elle peut être exprimée formellement comme suit.

Définition 1.6. Relation de raffinement entre spécifications avec transformation de l'espace d'états.
Soit $S_1(x, x')$ et $S_2(w, w')$ deux spécifications définies sur des espaces d'états différents. Soit $J(x, w)$ une relation totale définie entre ces deux espaces. S_1 est raffinée par S_2 si et seulement si, la pré-condition de S_1 est affaiblie, et si pour tous les états qui satisfont la pré-condition de S_1 la post-condition de S_1 est renforcée. Formellement :

$$\begin{aligned} S_1 \sqsubseteq_J S_2 \quad \Leftrightarrow \quad \forall_{x,w,w'} \cdot (\quad & P_1(x) \wedge J(x, w) \Rightarrow \\ & P_2(w) \wedge \\ & (Q_2(w, w') \Rightarrow \exists_{x'} \cdot (Q_1(x, x') \wedge J(x', w')))) \end{aligned}$$

Déf 1.6

Les spécifications S_1 et S_2 doivent être des formules bien définies. Elle doivent donc satisfaire la formule $\Delta_p(S_1) \wedge \Delta_p(S_2)$.

La définition que nous venons de donner correspond plutôt à un schéma qui permet de produire une relation de raffinement spécifique pour chaque invariant de collage : $J(x, w)$. En particulier, si l'invariant de collage est $x = w$, c'est à dire si l'espace reste inchangé, nous obtenons la relation de raffinement de la définition 1.4, page 22.

La propriété de réflexibilité du raffinement est donc satisfaite par cette relation, puisque dans ce cas l'espace d'états n'est pas changé.

La propriété de transitivité du raffinement peut être reformulée comme suit :

Propriété 1.17. *Transitivité de la relation de raffinement entre spécifications avec transformation de l'espace d'états.*
Soit $S_1(x, x')$, $S_2(x, x')$, $S_3(w, w')$, $S_4(w, w')$, *quatre spécifications.* S_1 *et* S_2 *sont définies sur le même espace d'états ;* S_3 *et* S_4 *sont définies sur un autre espace d'états. Soit* $J(x, w)$ *une relation définie entre ces deux espaces. Nous avons :*

$$S_1 \sqsubseteq S_2 \wedge S_2 \sqsubseteq_J S_3 \quad \Rightarrow \quad S_1 \sqsubseteq_J S_3$$

et

$$S_2 \sqsubseteq_J S_3 \wedge S_3 \sqsubseteq S_4 \quad \Rightarrow \quad S_2 \sqsubseteq_J S_4$$

1.7.1 Raffinement des Paramètres d'une Spécification

Les paramètres d'une spécification élémentaire ne peuvent pas être raffinés. Prenons par exemple la spécification d'un programme qui prend en entrée un nombre naturel et qui rend comme résultat son double :

$$y \in Nat \;\Rightarrow\; z' = 2 * y \wedge y' = y.$$

Supposons que nous effectuons un raffinement de données avec l'invariant de collage et la spécification qui suivent :

$$J(y, z, y1, z1) \quad \triangleq \quad y1 = y \wedge z1 = 2 * z$$

et

$$y1 \in Nat \;\Rightarrow\; z1' = 4 * y1 \wedge y1' = y1$$

A l'aide de la relation de raffinement de la définition 1.6, page 32, nous obtenons les deux obligations de preuve qui suivent :

$$\vdash \quad y \in Nat \wedge y1 = y \wedge z1 = 2 * z \;\Rightarrow\; y1 \in Nat$$

et

$$\vdash \quad y \in Nat \wedge y1 = y \wedge z1 = 2 * z \;\Rightarrow\; z1' = 4 * y1 \wedge y1' = y1 \;\Rightarrow\; \exists_{y',z'} \cdot (z' = 2 * y \wedge y' = y \wedge y1' = y' \wedge z1' = 2 * z')$$

Ces deux obligations peuvent être déchargées à l'aide de la théorie des ensembles et des axiomes de l'arithmétique. Le raffinement effectué est donc correct.

Avec ce raffinement, nous avons altéré les objets visibles de l'extérieur. Ceci doit être interdit car l'idée que nous avons et de construire un programme dont le comportement observable est aussi observable avec la spécification initiale.

Dans l'exemple, si $y = 1$, la spécification initiale produira un résultat $z' = 2$. Mais le programme produira $z1' = 4$. D'un point de vue externe, ce programme n'est pas correct. Par conséquent, les entrées et les sorties d'un programme ne peuvent pas être raffinées.

Ceci pose un problème important car notre intention quand nous écrivons une spécification est justement de faire des abstractions qui éliminent les détails d'implantation. Mais, puisque les entrées et les sorties ne peuvent pas être raffinées, nous devons dès le départ utiliser des objets concrets dans la spécification. Nous proposerons dans les chapitres suivants une méthode qui permet d'écrire des spécifications qui utilisent des entrées et des sorties abstraites.

1.8 Conclusion

Nous avons présenté deux approches différentes utilisées dans le processus de construction de programmes corrects. La première permet de vérifier qu'un programme est correct vis-à-vis de sa spécification : le programme est écrit d'abord puis une preuve de correction est effectuée. La deuxième permet de dériver un programme correct en partant de sa spécification. La relation de raffinement est l'outil théorique qui permet de conduire le processus de développement.

Deux méthodes de construction de programmes par raffinements ont été étudiées. La première traite les spécifications comme des programmes abstraits. Les objets raffinés sont les programmes. La deuxième associe des spécifications aux programmes. Les objets raffinés sont les spécifications. Nous utilisons la deuxième méthode pour présenter les idées théoriques de ce travail car nous pensons qu'il est plus simple et naturel de raisonner sur des spécifications que sur des programmes.

Nous avons donné les instructions d'un langage de spécification et de programmation impérative non récursif très proche de celui proposé dans la méthode B, et nous avons défini la fonction SL qui associe une spécification à chaque instruction du langage de telle sorte que le processus de développement puisse être conduit en raisonnant uniquement sur des spécifications.

Nous avons défini aussi la fonction SP qui associe une spécification à chaque programme écrit dans le langage présenté. Ainsi, la sémantique d'un programme est donnée par la spécification produite par cette fonction. La fonction SP établit le lien qui permet de définir la notion de correction totale d'un programme vis-à-vis d'une spécification : définition 1.5, page 28.

La démarche de développement proposée est descendante car nous partons d'une spécification abstraite qui est transformée graduellement jusqu'à l'obtention d'un programme correct. Ceci permet de contrôler efficacement le processus de preuve puisqu'à chaque pas de raffinement nous sommes guidés par l'intuition que nous avons sur la correction de l'instruction que nous introduisons.

La transformation d'une spécification par petits pas est possible grâce à la

propriété de transitivité du raffinement : propriété 1.13, page 22. La propriété de monotonie du raffinement, propriété 1.14, page 27, permet de découper en parties le processus de développement. L'espace de définition d'une spécification peut être changé grâce à la relation de raffinement de la définition 1.6, page 32.

Les entrées et les sorties indiquées dans la spécification d'un programme ne peuvent pas être changées pendant le processus de développement car ce sont des objets observables de l'extérieur. La définition de correction utilise cette hypothèse. Ceci nous force dès le départ à inclure dans la spécification des entrées et des sorties concrètes, ce qui limite considérablement la réalisation d'abstractions dans la description du programme. Nous proposons une solution à ce problème dans le chapitre suivant.

Chapitre 2

Modules

Dans le chapitre 1 nous avons présenté la notion bien connue de spécification d'un programme à l'aide de deux formules écrites en logique de premier ordre. Nous voulons maintenant tenter de répondre à la question qui suit : *Dans la pratique, que pouvons nous spécifier de cette façon ?*

Il est clair que les programmes dont les interactions avec l'environnement se limitent à la lecture d'entrées au début et à l'envoi de résultats à la fin peuvent être spécifiés avec cette méthode, voir section 1.6, page 30.

Mais que pouvons nous dire d'un programme qui a d'autres interactions avec son environnement ? Un système réactif par exemple doit lire en permanence des informations qui proviennent de son environnement. Nous pouvons dans ce cas spécifier séparément ces interactions avec une opération de lecture : $z \longleftarrow Get$, et une opération d'écriture : $Put(x)$. La question qui se pose est *comment pouvons nous relier ces spécifications de telle sorte que nous puissions décrire le comportement du système dans son ensemble ?* Nous pourrions vouloir dire par exemple que dans un état bien précis le programme doit effectuer une lecture. Comment pouvons nous décrire cet état ? Pour le faire, nous devons pouvoir exprimer des propriétés sur les états internes du programme. Le problème que nous avons est qu'une seule spécification pré-post ne permet pas de le faire. Nous devons donc disposer d'un moyen de relier plusieurs spécifications élémentaires.

La notion connue de module fournit ce moyen. La spécification d'un module est formée par un ensemble de variables d'état et un ensemble de relations de transition [1] données sous la forme de spécifications élémentaires qui déterminent les états qui sont atteignables à partir de n'importe quel autre état. Nous retrouvons ici l'idée de spécification avant-après pour chaque opération du module [Abr 96-1, Bac,Kur 83, But 96, Lyn,Tut 87]. La figure 2.1 illustre cette idée. L'état du module reste inchangé entre la fin de l'exécution d'une de ses opérations et le début de la suivante.

Ce chapitre est dédié à l'étude des modules. Il présente d'abord la notion de spécification de module. Ensuite, il donne la relation de raffinement entre spécifi-

[1] A chaque *Opération* offerte par le module correspond une relation de transition.

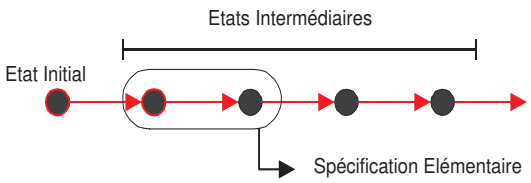

Figure 2.1 – Spécification de Module

cations de module, puis les techniques utilisées dans leur implantation. A la fin, il contient une analyse des techniques de spécification de programmes complexes à l'aide de spécifications de module. Dans cette partie, deux notions fondamentales sont introduites : les variables auxiliaires [Aba,Lam 88, Lam,Sha 90, Owi,Gri 76] et les opérations auxiliaires.

2.1 Spécification d'un Module

La spécification d'un module est un triplet $(x, O_0, (O_1, \ldots, O_i))$, avec $(0 < i \leq N)$, où x est une variable partagée, ou une liste de variable distinctes, qui dénote l'état du module ; O_0 est une spécification d'initialisation de la forme $true \Rightarrow Q_0(x')$ qui détermine les états initiaux ; et $(O_1, \ldots, O_i)$ est un ensemble de spécifications élémentaires de la forme $x, z : \Delta_{ce}(y) \wedge P_i(x, y) \Rightarrow Q_i(x, y, x', z')$ qui spécifient les opérations du module.

Un module est utilisé comme suit : lors de sa création, l'opération d'initialisation affecte les variables du module avec des valeurs qui correspondent à un état initial valide. Puis, à chaque fois qu'une de ses opérations est utilisée, les variables sont modifiées de telle sorte que l'état après satisfasse la post-condition de l'opération. Toutes les opérations du module sont observables de l'extérieur et leur exécution est atomique du point de vue d'un observateur externe. La variable x n'est pas observable de l'extérieur, elle est encapsulée dans le module.

2.1.1 Non Déterminisme Interne

Il y a deux types de non déterminisme associés aux modules : interne et externe. Le non déterminisme interne peut apparaître dans chaque spécification élémentaire lorsque la post-condition associe plusieurs états après à un état avant, voir section 1.5.3, page 20. Ce type de non déterminisme doit être éliminé lors de l'implantation du module.

2.1.2 Non Déterminisme Externe

Le non déterminisme externe apparaît avec l'introduction de plusieurs opérations car l'ordre des appels ainsi que les paramètres d'entrée utilisés sont

déterminés par l'utilisateur. La seule condition qui est requise est que les pré-conditions des opérations soient satisfaites lors de chaque appel. Le non déterminisme externe doit être préservé pendant le processus de développement car l'implantation d'un module doit fonctionner dans tous les cas indiqués dans la spécification initiale.

2.2 Relation de Raffinement entre Spécifications de Module

La relation que nous allons définir, $\sqsubseteq_J$, permet de transformer la spécification abstraite d'un module en une implantation dont la spécification associée [2] est un raffinement de la spécification abstraite [3].

La spécification d'un module est raffinée par une autre spécification de module si un observateur externe est incapable de les différencier en regardant leur comportement. Ceci est le cas lorsque les trois conditions qui suivent sont satisfaites :

Condition 2.1. Raffinement de spécifications de module.

1. Les deux modules ont la même signature. C'est à dire qu'un observateur externe ne peut pas faire la différence entre les deux modules en observant les opérations qu'ils offrent. Une conséquence de ceci est que les paramètres des opérations ne peuvent pas être raffinés. Une autre conséquence est que le module concret a exactement les mêmes opérations que l'abstrait.
2. Chaque opération de la spécification concrète fonctionne dans tous les cas où l'opération correspondante de la spécification abstraite fonctionne. Cette condition est satisfaite si les pré-conditions des opérations abstraites sont affaiblies par les pré-conditions concrètes. Un observateur externe ne voit pas de différence quand il utilise les opérations des deux modules dans les pré-conditions abstraites.
3. Chaque opération concrète, utilisée dans la pré-condition abstraite, renforce la post-condition abstraite correspondante. Cette condition doit être satisfaite aussi pour la spécification d'initialisation. Elle garantit que toute transition valide dans la spécification concrète et aussi valide dans l'abstraite [Aba,Lam 88]. Ainsi, toute transition observée de l'extérieur avec le module concret est une transition observable avec le module abstrait. Les deux spécifications peuvent avoir des variables internes différentes car ces variables ne sont pas observables de l'extérieur. Dans ce cas, il doit exister une relation qui associe au moins un état abstrait à chaque état concret [Aba,Lam 88].

Ces trois conditions sont satisfaites si la spécification d'initialisation abstraite est raffinée par la concrète, et si chaque spécification élémentaire abstraite est

[2] Cette spécification correspond à l'abstraction fonctionelle du module implanté.

[3] Afin de simplifier l'exposé, nous appelleront *abstraite* la spécification raffinée et *concrete* la spécification qui raffine.

raffinée par la concrète correspondante. C'est de cette façon que nous allons définir la relation de raffinement entre spécifications de module [Abr 96-1].

Définition 2.1. Relation de raffinement entre spécifications de module.
Soit $M1(x)$ et $M2(w)$ deux spécifications de module qui ont la même signature et définies sur deux espaces d'états différents. Soit $J(x,w)$ une relation totale définie entre ces deux espaces d'états. $M1$ est raffinée par $M2$ si et seulement si la spécification d'initialisation de $M1$ est raffinée par la spécification d'initialisation de $M2$, et si chaque spécification d'opération de $M1$ est raffinée par la spécification correspondante de $M2$. Formellement :

$$M1 \sqsubseteq_J M2 \quad \Leftrightarrow \quad O1_0 \sqsubseteq_J O2_0 \quad \wedge \quad O1_i \sqsubseteq_J O2_i \qquad avec\ (i \in 1..N)$$

Déf 2.1

Cette définition ne permet pas le raffinement des paramètres des opérations car les deux spécifications ont exactement la même signature. Le raffinement de données est donc admis uniquement entre les variables encapsulées par le module.

Un théorème important qui en résulte est que toutes les séquences d'états admises par la spécification concrète sont aussi des séquences d'états admises par la spécification abstraite [Aba,Lam 88, Bac,Wri 90]. Par conséquent, à toute exécution concrète correspond une exécution abstraite qui produit exactement le même effet sur l'environnement [4]. Les preuves de théorèmes analogues sont données dans [Aba,Lam 88, Abr 96-1, Bac,Wri 90].

La relation de raffinement entre modules est réflexive et transitive. Ces propriétés peuvent s'exprimer formellement comme suit :

Propriété 2.1. *Réflexivité de la relation de raffinement entre spécifications de module. Soit M un module. Nous avons :*

$$M \sqsubseteq M$$

Cette propriété résulte de la propriété de réflexivité de la relation de raffinement entre spécifications, propriété 1.7, page 17, et de la définition 2.1.

Propriété 2.2. *Transitivité de la relation de raffinement entre spécifications de module avec transformation de l'espace d'états.*
Soit $M_1(x,x')$, $M_2(x,x')$, $M_3(w,w')$, $M_4(w,w')$, quatre spécifications. M_1 et M_2 sont définies sur le même espace d'états ; M_3 et M_4 sont définies sur un autre espace d'états. Soit $J(x,w)$ une relation totale définie entre ces deux espaces. Nous avons :

[4] Une exécution d'un module est une suite d'opérations du module appelées en séquence ; l'une après l'autre. La suite peut être finie ou infinie.

$$M_1 \sqsubseteq M_2 \wedge M_2 \sqsubseteq_J M_3 \quad \Rightarrow \quad M_1 \sqsubseteq_J M_3$$

et

$$M_2 \sqsubseteq_J M_3 \wedge M_3 \sqsubseteq M_4 \quad \Rightarrow \quad M_2 \sqsubseteq_J M_4$$

Cette propriété résulte de la propriété de transitivité de la relation de raffinement entre spécifications, propriété 1.17, page 33, et de la définition 2.1. Elle permet le développement par petits pas.

La spécification abstraite d'un module peut donc être remplacée par son implantation car toutes les propriétés satisfaites par la spécification abstraite sont aussi satisfaites par la spécification concrète associée à l'implantation obtenue.

2.3 Invariant d'un Module

Les spécifications de module permettent d'exprimer d'une façon très commode des propriétés qui sont satisfaites par tous les états qui peuvent être atteints par le système. Ces propriétés sont connues sous le nom de propriétés de sûreté parce qu'elles interdissent les états qui sont considérés dangereux ou indésirables. Les modules peuvent générer des suites infinies d'états. Ceci permet de spécifier des programmes qui ne terminent pas ce qui est souvent le cas dans les systèmes cycliques. Pour le faire, nous nous intéressons aux propriétés qui sont satisfaites par tous les états atteignables ; elles forment *l'invariant du module*. En conséquence, la spécification d'initialisation doit établir l'invariant, et la post-condition de chaque opération doit le rétablir.

L'invariant simplifie le processus de preuve car il est toujours présent en tant qu'hypothèse. Nous allons permettre l'écriture explicite d'invariants dans les spécifications de module [Abr 96-1, Morg 90].

2.4 Opérations

Deux opérations d'un même module ne peuvent pas être exécutées d'une façon concurrente. Ceci veut dire qu'une fois qu'une opération démarre, aucune autre ne peut démarrer avant que celle qui est en cours d'exécution ne termine [5] . A premier abord, cette restriction semble injustifiée et contraignante. Il y a cependant deux bonnes raisons pour cela. La première est due à l'activité de preuve que nous devons effectuer lors du processus de développement d'un programme. Pour montrer la correction de l'implantation d'une opération d'un module nous avons besoin à tout moment de connaître le contexte dans lequel les instructions du programme sont utilisées. Une autre opération qui serait exécutée simultanément pourrait modifier le contexte de la première ce qui rendrait le processus de preuve impossible à réaliser. Par exemple, nous supposons que

[5] Dans la pratique il est plus simple de nommer les opérations d'un module plutôt que de les numéroter. Nous allons donc désigner les opérations avec des noms différents. Ceci est en particulier le cas dans la méthode B.

l'invariant d'un module est satisfait au début et à la fin de l'exécution de chacune de ses opérations : la pré-condition contient l'invariant et la post-condition le rétablit. Mais, rien n'est garantit lorsque une opération est en cours d'exécution : l'invariant n'est pas nécessairement satisfait. Il serait donc erroné de supposer que lors du démarrage d'une autre opération du module l'invariant est satisfait.

La deuxième raison est plutôt méthodologique. Nous regardons un module comme une 'couche' logicielle qui offre des services aux couches supérieures. Chaque service offert est atomique du point de vue du programme qui l'utilise. Ainsi, lorsqu'un service est demandé, le programme doit attendre la fin de l'exécution avant de pouvoir demander un autre service du même module.

Une conséquence de ceci est qu'une opération d'un module ne peut pas être utilisée pour implanter une autre opération du même module, ce qui nous semble sain d'un point de vue méthodologique [6] .

2.5 Implantation d'une Opération

Les opérations d'un module sont implantées séparément. Chaque opération est vue comme un programme indépendant qui agit sur les variables du module.

En vertu de la propriété de monotonie du raffinement, propriété 1.14, page 27, pour montrer que la spécification d'une opération concrète raffine la spécification d'une opération abstraite il suffit de montrer que les feuilles de l'arbre syntaxique du programme abstrait sont correctement raffinées. Prenons par exemple une opération abstraite dont le programme a la structure qui suit :

$$\dfrac{\dfrac{S_{1.1}}{S_{1.1.1}\quad S_{1.1.2}}I_{1.1} \qquad \dfrac{S_{1.2}}{S_{1.2.1}\quad S_{1.2.2}}I_{1.2}}{S_1}I_1$$

Nous pouvons raffiner séparément les quatre feuilles de cet arbre pour obtenir un raffinement correct de l'opération toute entière. Ainsi, le programme concret qui suit raffine le programme de l'arbre à condition que les raffinements des feuilles indiqués soient corrects :

$$\dfrac{\dfrac{\dfrac{S_{1.1.1}}{S_{1.1.1.R}}I_{1.1.1} \quad \dfrac{S_{1.1.2}}{S_{1.1.2.R}}I_{1.1.2}}{S_{1.1}}I_{1.1} \quad \dfrac{\dfrac{S_{1.2.1}}{S_{1.2.1.R}}I_{1.2.1} \quad \dfrac{S_{1.2.2}}{S_{1.2.2.R}}I_{1.2.2}}{S_{1.2}}I_{1.2}}{S_1}I_1$$

Le processus d'implantation d'une opération est terminé quand toutes les feuilles de l'arbre sont des instructions du langage de programmation. Dans notre contexte, il s'agira d'instructions d'affectation et d'appel de procédure.

[6] Ceci est le cas dans la méthode B.

Nous adopterons cette méthode de développement systématiquement. En procédant de cette façon, nous interdisons le remplacement d'instructions de programmation qui ont été introduites plus tôt, ce qui n'engendre pas de perte de généralité. L'exemple 1.6, page 28, illustre la démarche que nous adoptons.

En théorie, il est possible de remplacer d'autres parties de l'arbre à condition que la spécification qui en résulte raffine bien la spécification du départ. Dans ce cas, il serait nécessaire de faire intervenir dans le processus de preuve tous les invariants de collage qui ont été utilisés depuis le début du développement, ce qui rend le processus de preuve plus difficile à effectuer. Nous n'effectuerons pas ce type de raffinement car il ne correspond pas à l'idée que nous avons de construction de programmes par petits pas.

2.6 Implantation d'un Module

Nous allons diviser les variables d'un module en deux ensembles disjoints. Le premier contient les variables abstraites qui devront être raffinées pendant le processus de développement car elles ne sont pas directement implantables [7] . Le deuxième contient les variables locales [8] , qui correspondent à des objets directement implantables dans le langage de programmation utilisé pour générer le code du programme. Par exemple, un entier informatique ou un octet [9]. Lors de la génération du code associé au module les variables locales seront déclarées directement ; elles ne peuvent donc pas être raffinées.

Le processus de développement d'un module comporte deux activités différentes :

- L'écriture d'algorithmes qui implantent les opérations.
- L'implantation des variables abstraites avec des objets informatiques directement utilisables dans l'environnement choisi. Ces objets peuvent être soit des variables locales soit les variables *d'autres modules* utilisés dans l'implantation.

La relation de raffinement entre modules est l'outil qui permet de contrôler la validité des transformations effectuées. Il est possible d'identifier trois types de transformation distincts suivant le but recherché :

1. *Introduction d'instructions de programmation.* Les variables du module restent inchangées, mais les opérations sont modifiées afin d'introduire du détail dans les algorithmes qui les implantent. Ce type de transformation peut être effectué opération par opération car chaque opération peut être traitée comme un programme indépendant.

[7] La technique qui permet d'implanter une variable consiste à changer l'espace d'états d'une spécification dans un raffinement. L'invariant de collage permet de relier la variable abstraite à la variable concrète.

[8] Les variables locales correspondent aux variables concrètes dans la méthode B.

[9] Un entier informatique est un nombre entier compris entre *minint* et *maxint*. *minint* étant le plus petit entier représentable dans le système utilisé, *maxint* étant le plus grand entier représentable dans ce même système.

2. *Transformation de l'espace d'états.* Les variables de la spécification abstraite sont raffinées par les variables de la spécification concrète. Les variables *locales* restent inchangées. Il est donc nécessaire de fournir un invariant de collage entre les deux espaces : $J(x, w)$.

3. *Utilisation de modules.* D'autres modules sont utilisés pour implanter les variables abstraites et les opérations du module abstrait. Ce type de transformation est fondamental car il permet de diviser le processus de développement en plusieurs parties indépendantes.

   L'intention que nous avons quand nous utilisons une instance d'un module dans un programme est de cacher tous les détails d'implantation de ce module : ses variables et les algorithmes qui implantent ses opérations. L'utilisateur voit seulement les spécifications des opérations que le module offre. Ainsi un programme élémentaire peut utiliser un ou plusieurs modules dans son implantation. Pour cela, il doit d'abord les déclarer, d'une façon similaire à la déclaration d'une variable locale. Puis, il peut utiliser les opérations qu'ils fournissent.

   L'utilisation de modules dans l'implantation d'un autre module peut se faire de la façon indiquée dans le paragraphe précédent. D'abord, il faut déclarer les objets qui correspondent aux modules utilisés en leur donnant des noms différents. Puis nous pouvons appeler les opérations qu'ils offrent depuis les opérations du module en cours d'implantation [10] . Les modules déclarés sont partagés par toutes les opérations du module qui les utilise : leurs opérations peuvent être appelées depuis n'importe quelle opération du module qui est en cours d'implantation [11] .

   Ce type de transformation permet d'implanter une spécification de module abstraite avec une spécification de module concrète qui utilise des instances d'autres modules [12] . Pour montrer qu'une telle transformation est correcte, nous devons procéder comme suit : Les variables de la spécification concrète et celles des modules utilisés servent à implanter les variables de la spécification abstraite. Les noms de toutes les variables de la spécification concrète sont différents [13]. Nous devons donc fournir un invariant de collage $J(x, w)$ qui lie les variables du module abstrait aux variables du module concret. Puis, nous devons prouver que le raffinement est correct : chaque opération abstraite est raffinée par l'opération concrète correspondante, et l'initialisation abstraite est raffinée par l'ensemble d'initialisations des instances des modules utilisés et du module

[10] Un même module peut être déclaré plusieurs fois dans une implantation. Les instances auront des noms différents.

[11] La démarche que nous décrivons ici est très proche de celle proposée dans la méthode B. La clause $IMPORTS$ en B permet de déclarer tous les modules qui seront utilisés dans l'implantation d'opérations [Abr 96-1], page 561. Cette implantation doit se faire dans une $IMPLEMENTATION$ [Abr 96-1], page 559. Ceci correspond au module concret qui permet de raffiner le module abstrait.

[12] Les déclarations circulaires ne sont pas permises. Le cas qui suit est donc interdit : M1 utilise une instance de M2 et M2 utilise une instance de M1.

[13] Ceci peut être effectué en préfixant les noms de ces variables avec les noms des instances correspondantes des modules utilisés.

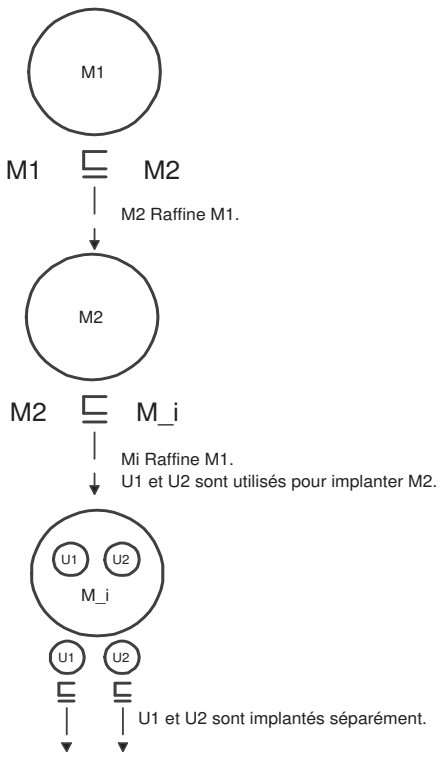

Figure 2.2 – Processus d'Implantation d'un Module

concret appelées en séquence dans un ordre quelconque.

Les invariants et les variants de boucle des implantations peuvent contenir les variables des modules déclarés car l'invariant et le variant servent seulement à fournir une preuve de correction du raffinement. Ce sont des formules logiques qui n'engendrent pas de code exécutable, voir [Abr 96-1], page 751.

L'implantation d'un module est terminée quand toutes ses opérations ont été complètement implantées, et qu'il ne reste aucune variable abstraite à implanter. Les modules utilisés dans une implantation seront à leur tour implantés séparément en utilisant les mêmes principes. La figure 2.2 illustre cette idée :

Le processus d'implantation d'un module peut être schématisé comme suit :

Soit $M1 = (V1_a, O1_0, O1_1 \ldots O1_n)$ la spécification du module abstrait à implanter. $V1_a$ représente les variables abstraites. $V1_l$ représente les variables locales : $M1$ n'en a pas. $OI1$ représente les instances des modules utilisés : $M1$ n'en a pas. $O1_0$ est la spécification d'initialisation. $O1_1 \ldots O1_n$, sont les spécifications des opérations du module.

Soit $M2 = (V2_a, V2_l, O2_0, O2_1 \ldots O2_n)$ le module utilisé pour effectuer le

premier raffinement. Des variables locales sont introduites : $V2_l$. Les variables abstraites restent inchangées : $V1_a = V2_a$. L'implantation d'opérations a débutée : $O1_i \sqsubseteq O2_i$, $i \in 0 \ldots n$.

Soit $M3 = (V3_a, V3_l, O3_0, O3_1 \ldots O3_n, J3)$ le module utilisé pour effectuer le deuxième raffinement. Des nouvelles variables locales sont introduites : $V2_l \subset V3_l$. Des variables abstraites ont disparu : $V3_a \subset V2_a$. Un invariant de collage a été fourni : $J3$. L'implantation d'opérations continue.

Soit $M4 = (V4_a, V4_l, OI4, O4_0, O4_1 \ldots O4_n, J4)$ le module utilisé pour effectuer le troisième raffinement. Des nouvelles variables locales sont introduites : $V3_l \subset V4_l$. Des variables abstraites ont disparu : $V4_a \subset V3_a$. Un invariant de collage a été fourni : $J4$. D'autres modules ont été déclarés : $OI4$. L'implantation d'opérations continue.

Soit $Mk = (Vk_l, OIk, Ok_0, Ok_1 \ldots Ok_n, Jk)$ le module utilisé pour effectuer le dernier raffinement. Toutes les variables abstraites ont disparu : Vk_a est vide. Un invariant de collage a été fourni : Jk. L'implantation d'opérations a été complétée. Ce dernier module contient seulement des variables locales : Vk_l, et des instances d'autres modules : OIk.

$$
\begin{array}{ll}
 & M1 = (V1_a, O1_0, O1_1 \ldots O1_n) \\
\sqsubseteq & \\
 & M2 = (V2_a, V2_l, O2_0, O2_1 \ldots O2_n) \\
\sqsubseteq_{J3} & \\
 & M3 = (V3_a, V3_l, O3_0, O3_1 \ldots O3_n, J3) \\
\sqsubseteq_{J4} & \\
 & M4 = (V4_a, V4_l, OI4, O4_0, O4_1 \ldots O4_n, J4) \\
\sqsubseteq_{J5} & \\
 & \vdots \\
\sqsubseteq_{Jk} & \\
 & Mk = (Vk_l, OIk, Ok_0, Ok_1 \ldots Ok_n, Jk)
\end{array}
$$

2.7 Constantes

Les spécifications de module peuvent faire référence à des objets qui ne changent jamais. Ce sont les *constantes* du système [Abr 96-1]. L'utilisation de constantes permet d'éviter des erreurs de spécification car nous pourrons vérifier formellement que toutes les post-conditions des opérations d'un module les laissent inchangées.

De même que pour les variables, nous les séparerons en deux ensembles disjoints : abstraites et locales. Les constantes abstraites doivent être implantées avec d'autres constantes, de la même façon que les variables abstraites sont implantées avec d'autres variables. Les constantes locales ne peuvent pas être raffinées, elles seront des objets du programme final.

Le fait que les constantes soient nécessairement implantées avec d'autres constantes garantit qu'elles restent inchangées tout au long de l'exécution du

programme.

2.8 Spécification de Programmes : Analyse

Quand nous spécifions un programme, nous faisons des abstractions dans le but de simplifier la compréhension des idées exprimées. Par exemple, pour décrire le tri d'un fichier de nombres entiers, nous supposons que le fichier existe et que nous avons le moyen d'y accéder et de le modifier. Avec cette hypothèse, nous donnons la spécification qui suit : écrire un programme qui trie un fichier qui contient des nombres entiers [14] :

$$F \in seq(INT) \;\Rightarrow\; F' = perm(F) \wedge sorted(F')$$

Cette spécification élimine toutes les complications qui apparaissent lors d'un développement réel : la façon d'indiquer le nom du fichier à trier, la nécessité d'éviter tous les effets de bord que le programme peut causer dans son environnement, le traitement des erreurs qui peuvent apparaître en cours d'exécution et qui sont dues à une mauvaise utilisation du programme, ou les possibles accès concurrents au fichier, ce qui peut se traduire par l'impossibilité de le trier. Pendant le processus de développement tous ces aspects doivent être pris en compte car ils sont à l'origine de nombreux dysfonctionnements.

Le but de la spécification d'un programme est de décrire le comportement de celui-ci sans donner de détail sur son implantation. Durant l'exécution, un observateur externe ne doit pas constater de différence entre ce qu'il observe et ce que la spécification dit qu'il est possible d'observer. Il est donc naturel de penser que la construction de la spécification d'un programme soit fortement guidée par les événements qui peuvent être observés de l'extérieur lorsqu'il est exécuté.

Nous allons utiliser les modules en tant qu'unité fondamentale de programmation. Ceci n'engendre pas de perte de généralité car tout programme qui peut être spécifié sous la forme d'une spécification élémentaire peut aussi être spécifié avec un module qui contient une seule opération.

Dans cette section, nous allons analyser les possibilités qui offrent les modules pour spécifier le comportement d'un programme. Le travail de spécification consistera essentiellement à décrire les interactions du programme avec son environnement ; c'est à dire, les évènements qui peuvent être observés lors de son exécution.

2.8.1 Utilisation d'une Seule Opération avec Paramètres

Une première possibilité que nous avons pour décrire ces interactions est d'utiliser des paramètres d'entrée et de sortie. Cette méthode est cependant très limitée car elle restreint les interactions à l'acquisition d'entrées au début de l'exécution et à l'envoi des résultats à la fin. Dans la pratique, peu de

[14] $perm(F)$ dénote une permutation de F, $sorted(F')$ est un prédicat qui définit les conditions qui déterminent que la suite F' est triée.

programmes peuvent être spécifiés de cette façon sans faire d'hypothèse trop simplificatrice.

Analysons par exemple un programme qui trouve le maximum d'un groupe non connu par avance d'entiers fournis par l'utilisateur. Son comportement peut être décrit comme suit : le programme doit d'abord obtenir le premier nombre, puis questionner l'utilisateur pour savoir s'il y a d'autres nombres à évaluer. Si la réponse est "oui", il doit obtenir le deuxième nombre et reformuler la question. Ce processus s'arrête quand l'utilisateur indique qu'il n'y a plus de nombre à évaluer. Le programme doit alors envoyer le résultat qui doit correspondre au maximum des nombres fournis par l'utilisateur. Il est impossible de spécifier un tel comportement avec une opération paramétrée car les nombres à évaluer sont renseignés par l'utilisateur tout au long de l'exécution du programme, ce qui est très différent de les fournir tous d'un seul coup au début de l'exécution.

Un argument qui est souvent utilisé dans ce cas consiste à dire qu'il suffit de supposer que l'opération contient un paramètre d'entrée 'fictif' qui lui passe tous les entiers que l'utilisateur compte évaluer, et un paramètre de sortie qui envoie le résultat. Par exemple [15] :

$$z \longleftarrow Maximum(arr) \quad \triangleq \quad arr \in seq(INT) \ \Rightarrow \ z' = max(ran(arr))$$

Le problème avec ce raisonnement est qu'un observateur d'une exécution du programme verrait seulement deux événements : le lancement du programme avec le paramètre d'entrée arr et la fin du programme avec le résultat obtenu z, ce qui ne correspond pas au comportement décrit d'une façon informelle plus haut, car il y a beaucoup plus d'événements qui peuvent être observés ; deux pour chaque réponse donnée par l'utilisateur :

- la réponse à la question -*Y a-t-il d'autres nombres à évaluer ?*-,
- la réponse à la demande -*Entrez le nombre suivant à évaluer :*-.

La spécification que nous voulons construire doit décrire tous ces évènements, ce qui n'est pas fait dans la formule donnée plus haut.

2.8.2 Utilisation d'un Module avec Plusieurs Opérations

Une deuxième possibilité consiste à utiliser un module, $MAXI$, qui offre deux opérations. La première, $Enter(n)$, permet d'obtenir un nouvel entier à prendre en compte. La deuxième, $m \longleftarrow Maximum$, rend le maximum des entiers déjà obtenus, voir [Abr 96-1], page 508 [16] .

Dans la spécification d'une opération, nous écrirons I pour désigner l'inva-

[15] $max(ran(arr))$ dénote le maximum du co-domaine de arr.

[16] Le partie "MACHINE" permet de donner un nom au module. Les mots "VARIABLES, INVARIANT, INITIALIZATION et OPERATIONS" indiquent la partie du module qui est décrite.

La spécification en B du module $MAXI$ est :

riant à l'état avant et I' pour désigner l'invariant à l'état après [17] . La spécification de ce module $MAXI$ est :

MODULE MAXI	Maximum d'un ensemble d'entiers.
VARIABLES y	Contient les entiers à évaluer.
INVARIANT $y \in F(INT)$	$F(INT)$ dénote un ensemble fini d'entiers.
INITIALIZATION $true \Rightarrow y' = \emptyset$	Au départ y est vide.
OPERATIONS $Enter(n) \hat{=}$ $n \in INT \wedge I \Rightarrow y' = y \cup \{n\} \wedge I'$	n est la valeur fournie par l'utilisateur.
$m \longleftarrow Maximum \hat{=}$ $y \neq \emptyset \wedge I \Rightarrow m' = max(y) \wedge I'$	m' est la réponse envoyée à l'utilisateur.

Le problème avec ce module est qu'il ne peut pas être utilisé seul. Les opérations qu'il offre doivent être appelées par le programme que nous souhaitons spécifier, ce qui signifie que les opérations $Enter$ et $Maximum$ ne sont pas observables de l'extérieur. Nous pouvons cependant supposer que les opérations du module $MAXI$ sont visibles de l'extérieur. Le problème avec ce raisonnement est que dans la réalité, pour implanter $MAXI$ nous allons devoir utiliser un module d'interface dont le but est de permettre la communication entre le programme et son environnement. Ce module d'interface ne peut pas être utilisé pour implanter l'opération $Enter$ par exemple, parce que cette opération doit être appelée avec un paramètre qui correspond au nouvel entier donné par l'utilisateur, ce qui signifierai que l'interaction avec l'utilisateur a été déjà effectuée et que l'appel à l'opération d'interface a été fait ailleurs. Plus que ça, l'opération $Enter$ doit être appelée avec un paramètre valide qui satisfait sa pré-condition. Il est donc nécessaire de vérifier avant l'appel de cette opération que l'information donnée par l'utilisateur correspond bien à un entier compris entre *maxint*

MACHINE
MAXI

VARIABLES
y

INVARIANT
$y \in F(INT)$

INITIALIZATION
$y := \emptyset$

OPERATIONS
$Enter(n) \hat{=}$ PRE $n \in INT$ $THEN$ $y := y \cup \{n\}$ END ;

$m \longleftarrow Maximum \hat{=}$ PRE $y \neq \emptyset$ $THEN$ $m := max(y)$ END

Dans le chapitre 4 nous établirons les liens qui existent entre notre modèle théorique et B.

[17] Ceci n'est pas nécessaire en B car ce travail est effectué lors de la preuve de cohérence d'une machine abstraite.

et *minint*. L'opération *Enter* ne peut donc pas être observée de l'extérieur.

La spécification du module $MAXI$ est plutôt la spécification d'un composant qui peut être utilisé pour implanter le programme que nous souhaitons construire. Cette technique est très utile pour spécifier des composants qui serviront dans une implantation. Nous pouvons l'utiliser par exemple pour spécifier des objets pour gérer des piles, des listes, des arbres, des graphes, etc. Mais elle ne permet pas de prendre en compte les interactions d'un système avec son environnement.

Notre problème reste entier : *comment pouvons-nous spécifier les interactions de notre programme avec son environnement ?*

2.8.3 Utilisation d'un Module Clos

Une troisième possibilité est de spécifier un module avec une seule opération sans paramètres. Ceci permet d'utiliser des variables abstraites pour modéliser les entrées et les sorties du programme. La spécification du module MAX qui suit illustre cette technique.

La spécification contient trois variables abstraites : In, Out, Con.

- In représente les entiers fournis par l'utilisateur. Son type est une suite d'entiers, ce que nous écrivons $In \in seq(INT)$ [18].
  Cette variable est utilisée comme suit : au début de l'exécution du programme, elle dénote la suite vide car à ce moment aucune entrée n'a été reçue. Pendant l'exécution, chaque entier obtenu est inséré en queue de la suite. A la fin, la variable dénote une suite qui contient tous les entiers reçus dans l'ordre de leur arrivée.
- Out représente la sortie envoyée par le programme. Son type est aussi une suite d'entiers, ce que nous écrivons $Out \in seq(INT)$.
  Cette variable est utilisée d'une façon semblable à In mais pour garder les résultats envoyés à l'utilisateur.
- Con représente la condition d'arrêt du programme. Son type est booléen, ce que nous écrivons $Con \in BOOL$.
  La variable est utilisée comme suit : au début elle contient la valeur $false$ car c'est l'utilisateur qui doit donner l'ordre d'arrêt du programme. A la fin, elle contient la valeur *true*, ce qui signifie que l'utilisateur a donné cet ordre. Après, aucune nouvelle entrée ne peut être obtenue par le programme.

Nous n'avons pas besoin d'inclure une variable pour garder le nombre maximum calculé car cette information est envoyée à l'utilisateur pendant l'exécution du programme : elle n'affecte pas l'état du module MAX. Nous aurons besoin par contre d'utiliser une variable locale dans l'implantation de l'opération qui effectue le calcul.

Les formules qui déterminent le type des variables sont placées dans l'invariant puisqu'elles sont toujours satisfaites. L'invariant I est :

$$In \in seq(INT) \wedge Out \in seq(INT) \wedge Con \in BOOL$$

[18] Nous supposons que INT, $seq(INT)$ et $BOOL$ ont été préalablement définis [Abr 96-1].

La spécification d'initialisation affecte les variables avec leurs état initiaux. Au départ In et Out sont vides puisque aucune entrée n'a été reçue et aucun résultat n'a été envoyé. Con est $false$. La spécification d'initialisation peut être écrite comme suit :

$$true \Rightarrow In' = \emptyset \wedge Out' = \emptyset \wedge Con' = false \wedge I' \tag{2.1}$$

La spécification de l'opération qui décrit le comportement du programme sera appelée *Maximum*. Elle est écrite de la façon suivante : La pré-condition est l'invariant du module car le programme peut être utilisé à tout moment. La post-condition contient les propriétés que le programme doit satisfaire. Nous voulons exprimer qu'une seule sortie doit être produite et que le résultat envoyé doit être égal au maximum des entiers lus. Formellement [19] :

$$size(Out') = 1 \wedge Out'(1) = max(ran(In')) \tag{2.2}$$

Nous voulons dire en plus qu'à la fin la condition d'arrêt est satisfaite et que l'invariant est rétablit. Formellement :

$$Con' = true \wedge I'$$

La spécification complète du module est [20] :

[19] max est une fonction qui prend un ensemble d'entiers et qui rend le maximum. ran est une fonction qui prend une suite et qui rend son co-domaine [Abr 96-1].

[20] La spécification correspondante du module MAX en B est :

MACHINE
MAX

VARIABLES
In, Out, Con

INVARIANT
$In \in seq(INT) \wedge Out \in seq(INT) \wedge Con \in BOOL$

INITIALIZATION
$In, Out, Con : (In = \emptyset \wedge Out = \emptyset \wedge Con = false)$

OPERATIONS
Maximum $\hat{=}$
$In, Out, Con : (size(Out) = 1 \wedge Out(1) = max(ran(In)) \wedge Con = true)$

MODULE	
MAX	Nom du programme
VARIABLES	
In, Out, Con	
INVARIANT	
$In \in seq(INT) \wedge$	In est une suite d'entiers qui représente les entrées reçues.
$Out \in seq(INT) \wedge$	Out est une suite d'entiers qui représente les sorties envoyées.
$Con \in BOOL$	Con est un booléen qui représente la condition d'arrêt du programme.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$In = \emptyset \wedge$	Il n'y a pas d'entrée.
$Out = \emptyset \wedge$	Il n'y a pas de sortie.
$Con = false \wedge$	La condition d'arrêt est $false$.
I'	L'invariant est établit.
OPERATIONS	
Maximum $\hat{=}$	
$I \Rightarrow$	A la fin :
$size(Out') = 1 \wedge$	Une seule sortie a été envoyée.
$Out'(1) = max(ran(In')) \wedge$	Le résultat envoyé est le maximum des entiers reçus.
$Con' = true \wedge$	La condition d'arrêt est satisfaite.
I'	L'invariant est rétablit.

Le problème avec cette spécification est qu'elle n'impose aucune contrainte sur la façon de raffiner les variables In, Out et Con. Si nous fournissons uniquement ce module, nous devrons faire l'hypothèse que lors de son implantation les opérations de communication avec l'environnement sont correctement utilisées, ce qui n'est pas acceptable car le développeur aurait toute liberté d'action lors de l'implantation. Par exemple, rien n'empêche de raffiner l'opération $Maximum$ de tel sorte qu'à la fin la post-condition soit satisfaite, même si aucune interaction avec l'environnement n'a été effectuée. La formule suivante est un raffinement correct de l'opération $Maximum$ [21] :

$$I \Rightarrow In' = (1 \mapsto 10) \wedge Out' = (1 \mapsto 10) \wedge Con' = true$$

Il est en effet facile de montrer les deux obligations de preuve générées par ce raffinement, voir définition 1.4, page 22 :

$$I \Rightarrow I$$

et

$$I \Rightarrow (In' = (1 \mapsto 10) \wedge Out' = (1 \mapsto 10) \wedge Con' = true \Rightarrow (size(Out') = 1 \wedge Out'(1) = max(ran(In')) \wedge Con' = true \wedge I'))$$

Le difficulté vient du fait que nous n'avons aucun moyen de forcer le développeur à programmer effectivement des appels à des opérations de communication avec l'environnement. Nous devons donc introduire des mécanismes qui forcent ces interactions.

[21] $(a \mapsto b)$ dénote un couple qui appartient à une relation.

2.8.4 Utilisation d'un Module pour Décrire le Comportement de l'Environnement

Nous allons proposer une quatrième méthode qui consiste à spécifier le programme avec un module qui contient une seule opération sans paramètres, comme le module MAX du cas précédent, et à spécifier l'environnement avec un autre module ENV [22]. L'idée d'inclure la description de l'environnement dans la spécification est proposée dans d'autres travaux [Led 93].

Le module ENV fournit au programme l'interface de communication avec l'environnement : Il doit permettre de lire des entiers en provenance de l'extérieur, d'envoyer des résultats, et de poser des questions à l'utilisateur. Ce sont les seuls événements qui peuvent être observés de l'extérieur : *lire un entier, envoyer un résultat, et poser une question.* La spécification est [23] :

MODULE ENV	Environnement
OPERATIONS	
$z \longleftarrow$ Read $\triangleq$ $true \Rightarrow \exists_a \cdot (a \in INT \wedge z' = a)$	Permet de lire un entier. a est la valeur fournie par l'utilisateur. La valeur est envoyée au programme.
Write$(n) \triangleq$ $n \in INT \Rightarrow skip$	Permet d'écrire un entier. Il n'y a pas d'effet sur l'état du module.
$c \longleftarrow$ More $\triangleq$ $true \Rightarrow \exists_a \cdot (a \in BOOL \wedge c' = a)$	Permer de poser une question. a est la réponse donnée par l'utilisateur. La réponse est envoyée au programme.

Ce module offre des possibilités qui vont bien au delà de nos besoins. Le programme peut par exemple envoyer plusieurs résultats à l'utilisateur, ou poser consécutivement 100 fois la même question, ce qui n'est pas souhaité. Il est donc nécessaire de pouvoir exprimer d'autres propriétés qui interdissent ces comportements non désirés.

Cette méthode permet de spécifier avec précision le comportement de l'environnement. Elle engendre cependant deux questions importantes :

1. Quel sens pouvons nous donner à un programme qui est spécifié comme un *univers clos* ? c'est à dire avec un module sans paramètres d'entrée ou de sortie. C'est le cas du module MAX, voir page 51. Il est clair que

[22] Le comportement de l'environnement peut être spécifié avec un ou plusieurs modules indépendants. Ceci dépend de la nature du problème traité.

[23] La spécification correspondante du module ENV en B est :

MACHINE
ENV

OPERATIONS
$z \longleftarrow$ Read $\triangleq$ $ANY \quad a \quad WHERE \quad a \in INT \quad THEN \quad z := a \quad END$;

Write$(n) \triangleq$ $PRE \quad n \in INT \quad THEN \quad skip \quad END$;

$c \longleftarrow$ More $\triangleq$ $ANY \quad a \quad WHERE \quad a \in BOOL \quad THEN \quad c := a \quad END$

dans ce cas toutes les variables abstraites peuvent être raffinées librement à condition de respecter la relation de raffinement entre modules.

2. Comment pouvons nous spécifier les interactions souhaitées entre le programme et son environnement ?

Pour apporter une réponse à ces deux questions, nous allons introduire des mécanismes qui permettent de spécifier l'utilisation que le programme peut faire du module d'environnement et qui permettent de spécifier les liens qui existent entre les deux modules qui forment la spécification. Pour cela, nous allons utiliser dans les spécifications des variables auxiliaires [24].

2.9 Variables Auxiliaires

Jusqu'à présent, les variables que nous avons utilisées dans les spécifications représentent des objets qui doivent être implantés, voir section 2.6, page 43. Il est cependant très utile de pouvoir inclure dans les spécifications des variables qui ne seront pas implantées [Aba,Lam 88, Lam,Sha 90, Owi,Gri 76]. Elles sont appelées *auxiliaires* car elles servent soit à exprimer d'autres propriétés du système, soit à effectuer les preuves générées pendant le processus de développement.

L'idée est de partir d'une spécification donnée et de la transformer à l'aide de variables auxiliaires en une nouvelle spécification de telle sorte que toute implantation correcte de la spécification initiale soit aussi une implantation correcte de la spécification transformée lorsque nous ignorons les variables auxiliaires qu'elle contient. De cette façon, il est possible d'utiliser la spécification transformée pendant le processus de développement pour prouver la correction des raffinements effectués, tout en gardant l'implantation du module donnée au départ. Le but de cette transformation est de compléter la spécification initiale afin de restreindre l'usage que le programme que nous souhaitons construire peut en faire, et de permettre l'expression d'autres propriétés qui font intervenir des notions qui ne sont pas présentes directement dans la spécification d'origine.

Puisque nous ne connaissons pas les détails de l'implantation fournie initialement, les modifications que nous allons effectuer sur la spécification ne peuvent en aucun cas altérer le comportement qu'elle décrit. Notre but n'est pas de raffiner cette spécification pour l'implanter, mais seulement de la compléter pour l'adapter aux besoins de spécification et de preuve du problème traité.

Cette démarche est très utile dans la pratique car elle permet de fournir des spécifications de module d'application générale qui ont été déjà implantées : librairies, classes d'objets, etc., pour être utilisées en tant que composants d'un système complexe. Ceci permet donc la réutilisation de composants.

Par exemple, la spécification du module ENV donnée plus haut est trop permissive par rapport au programme que nous voulons construire. D'un autre coté, le module ENV permet bien d'implanter le module MAX. Nous allons

[24] Nous les utiliserons pour désigner des valeurs apparues pendant l'exécution d'un programme : history variables [Aba,Lam 88, Lam,Sha 90, Owi,Gri 76], pour désigner des valeurs qui vont apparaître pendant cette exécution : prophecy variables [Aba,Lam 88], ou pour spécifier l'ordre d'appel aux opérations d'un module : control variables [Sha 93].

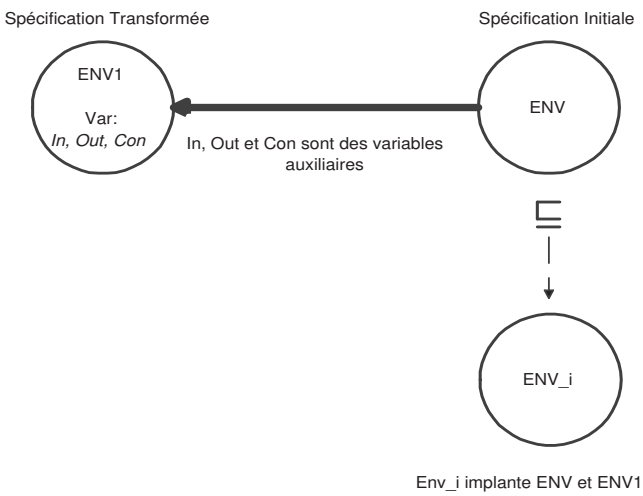

Figure 2.3 – Transformation de la Spécification d'un Module

donc transformer la spécification du module ENV afin d'introduire les notions dont nous avons besoin pour montrer la correction de l'implantation que nous allons effectuer du module MAX. La figure 2.3 illustre cette idée.

2.9.1 Transformation d'une Spécification Elémentaire à l'Aide de Variables Auxiliaires

Soit $S \triangleq P(x) \Rightarrow Q(x, x')$ une spécification élémentaire. Soit w une nouvelle variable ou une liste de variables distinctes de x. Nous allons transformer S en une nouvelle spécification $S_t \triangleq P_t(x, w) \Rightarrow Q_t(x, w, x', w')$ de telle sorte que la condition qui suit soit satisfaite :

Condition 2.2. Transformation d'une Spécification Elémentaire à l'Aide de Variables Auxiliaires.

1. La pré-condition de S est renforcée par la pré-condition de S_t :

$$\forall_{x,w} \cdot (P_t(x, w) \Rightarrow P(x))$$

2. La post-condition de S_t implique la post-condition de S :

$$\forall_{x,w,x',w'} \cdot (Q_t(x, w, x', w') \Rightarrow Q(x, x'))$$

3. La variable x' ne dépend ni de w ni de w'. Ceci est le cas lorsque sous la pré-condition de S_t, toutes les transitions admises par la post-condition de S sont aussi des transitions admises par la post-condition de S_t :

$$\forall_{x,w,x'} \cdot (P_t(x, w) \Rightarrow (Q(x, x') \Rightarrow \exists_{w'} \cdot Q_t(x, w, x', w'))) \qquad (2.3)$$

La première partie permet de restreindre l'utilisation de la spécification S aux situations qui nous intéressent. Dans notre exemple nous voulons interdire au programme *maximum* d'envoyer plusieurs sorties ; nous devront donc renforcer la pré-condition de l'opération $Write$ du module ENV à l'aide de variables auxiliaires. Dans cet exemple l'effet de la variable auxiliaire sera de forcer le développeur à utiliser correctement le module ENV car il devra prouver que les opérations sont appelées dans les pré-conditions.

Cette démarche peut paraître contradictoire avec celle du raffinement d'une spécification où la pré-condition est affaiblie. Cependant, le but de la transformation n'est pas de raffiner ou d'implanter le module ENV car il nous a été donné ; nous voulons juste l'utiliser d'une façon adaptée à l'implantation du module MAX. En renforçant les pré-conditions nous interdisons les utilisations non désirées des opérations du module MAX. Un cas extrême est celui qui consiste à interdire totalement à un programme l'utilisation d'une opération fournie par un module. Pour cela, il suffit de rendre $false$ la pré-condition de l'opération. Ceci peut paraître étrange et inutile. Néanmoins, si nous nous plaçons dans un cadre où nous souhaitons réutiliser des modules existants, cette possibilité peut s'avérer utile car elle nous permet d'interdire une opération jugée *dangereuse* dans la solution d'un problème particulier.

La deuxième partie garantit que S_t n'affaiblit pas la post-condition de S. Ceci veut dire que dans l'espace d'états de S toute transition admise par S_t est aussi admise par S.

La troisième partie garantit que les variables de la spécification initiale S ne dépendent pas des variables auxiliaires de S_t. *Ceci permet d'ignorer les variables auxiliaires lors de l'implantation de S_t et donc d'utiliser l'implantation fournie pour S.* Cette partie garantit aussi que le non déterminisme interne de S est préservé par S_t. C'est à dire que pour toute transition admise par S il existe un état w' tel que S_t est satisfait. Ceci est important car nous ne savons pas de quelle façon le non déterminisme interne de la spécification initiale sera éliminé. Nous savons seulement qu'une implantation correcte de celle-ci nous sera fournie.

Ces idées sont illustrées dans les exemples qui suivent :

Exemple 2.1. Utilisation de variables auxiliaires pour compter le nombre de fois qu'une opération d'un module est appelée par un programme.

Soit S la spécification d'une opération sans paramètres qui prend la forme :

$$S \quad \triangleq \quad P(x) \;\Rightarrow\; Q(x, x')$$

Nous voulons ajouter une variable auxiliaire N qui permet de compter le nombre de fois où l'opération est appelée durant l'exécution d'un programme qui utilise cette opération. Initialement, $N = 0$. Pour faire cela, nous allons transformer S en $S1$ de la façon suivante :

$$S1 \quad \triangleq \quad P(x) \wedge N \in NAT \;\Rightarrow\; Q(x, x') \wedge N' = N + 1 \wedge N' \in NAT$$

$S1$ est une transformation correcte de S car la condition 2.2 est satisfaite.

Nous voulons maintenant utiliser N pour spécifier que l'opération S ne doit pas être utilisée au delà d'un nombre fini de fois a. Pour cela, nous allons transformer $S1$ en $S2$ comme suit :

$$S2 \quad \triangleq \quad P(x) \wedge N \in NAT \wedge N < a \;\Rightarrow\; Q(x,x') \wedge N' = N+1 \wedge N' \in NAT$$

Il est facile de vérifier que $S2$ est aussi une transformation correcte de S.

Cet exemple permet d'illustrer le rôle de chaque partie de la condition 2.2 :

1. La pré-condition de $S2$ restreint clairement l'utilisation qui peut être faite de S aux cas souhaités : $N < a$.
2. La post-condition de $S2$ garantie que la post-condition $Q(x,x')$ est satisfaite. La transformation n'introduit donc pas de nouveau comportement dans l'opération transformée.
3. Toutes les transitions admises par $Q(x,x')$ sont aussi admises par la post-condition de $S2$ car la formule 2.3 est satisfaite. x' ne dépend donc ni de w ni de w'. par ailleurs, l'indéterminisme interne de S est préservé.

Exemple 2.2. Utilisation de variables auxiliaires pour tracer les valeurs reçues par un programme comme résultat des appels à une opération [25].

Soit S la spécification d'une opération d'un module qui prend la forme :

$$z \longleftarrow S \quad \triangleq \quad true \;\Rightarrow\; \exists_x \,.\, (x \in INT \wedge z' = x)$$

Nous voulons ajouter une variable auxiliaire In qui permet de tracer les valeurs reçues par le programme comme résultat des appels à cette opération. Initialement, $In = \emptyset$. Pour cela, nous allons transformer S en $S1$ comme suit [26] :

$$z \longleftarrow S1 \quad \triangleq \quad I \;\Rightarrow\; \exists_x \,.\, (x \in INT \wedge z' = x \wedge In' = In \leftarrow x \wedge I')$$

Il est facile de montrer que $S1$ est une transformation correcte de S

Nous allons maintenant formuler deux théorèmes qui déterminent le cadre d'utilisation des variables auxiliaires.

Soit $S \triangleq P(x) \Rightarrow Q(x,x')$ une spécification élémentaire faisable.

Soit $Si \triangleq Pi(x) \Rightarrow Qi(x,x')$ une spécification faisable qui implante S. Nous avons donc $S \sqsubseteq Si$.

Soit w une variable différente de x.

Soit $S_t \triangleq P_t(x,w) \Rightarrow Q_t(x,w,x',w')$ une transformation correcte de la spécification S. S_t satisfait donc la condition 2.2.

Soit Si_t la spécification Si transformée comme suit :

$$Si_t \quad \triangleq \quad Pi(x) \;\Rightarrow\; Qi(x,x') \wedge Q_t(x,w,x',w') \tag{2.4}$$

---

[25] $X \leftarrow a$ dénote la suite qui résulte après l'insertion de la valeur a dans la suite X, [Abr 96-1].

[26] Dans cette formule, $I \triangleq In \in seq(INT)$ et $I' \triangleq In' \in seq(INT)$. $seq(INT)$ est une suite d'entiers quelconque.

Théorème 2.1. *Implantation d'une spécification transformée avec des variables auxiliaires.*

Si S est raffinée par Si alors S_t est raffinée par Si_t. Formellement :

$$S \sqsubseteq Si \quad \Rightarrow \quad S_t \sqsubseteq Si_t$$

Ce théorème résulte de la définition de la relation de raffinement entre spécifications, définition 1.4, page 22.

Sous les hypothèses de la condition 2.2 et sous l'hypothèse $S \sqsubseteq Si$, nous devons montrer :

$$S_t \sqsubseteq Si_t$$

Nous avons donc les hypothèses suivantes :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow P(x)) \qquad \text{H1}$$
$$\forall_{x,w,x',w'} \cdot (Q_t(x,w,x',w') \Rightarrow Q(x,x')) \qquad \text{H2}$$
$$S \sqsubseteq Si \qquad \text{H3}$$

De H3 et de la définition du raffinement, définition 1.4, page 22, nous obtenons les deux hypothèses suivantes :

$$\forall_{x} \cdot (P(x) \Rightarrow Pi(x)) \qquad \text{H4}$$
$$\forall_{x,x'} \cdot (P(x) \Rightarrow (Qi(x,x') \Rightarrow Q(x,x'))) \qquad \text{H5}$$

D'après la définition du raffinement, définition 1.4, page 22, nous devons montrer les deux formules qui suivent :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow Pi_t(x,w)) \qquad \text{But 1}$$
$$\forall_{x,w,x',w'} \cdot (P_t(x,w) \Rightarrow (Qi_t(x,w,x',w') \Rightarrow Q_t(x,w,x',w'))) \qquad \text{But 2}$$

Pour le but 1 :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow Pi_t(x,w)) \qquad \text{B1}$$

Par définition de Si_t nous avons : $Pi_t(x,w) = Pi(x)$. Nous obtenons le but :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow Pi(x)) \qquad \text{B1.1}$$

Par H1 nous avons :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow P(x))$$

Par H4 nous obtenons :

$$\forall_{x,w} \cdot (P_t(x,w) \Rightarrow Pi(x))$$

Ce qui est le but à montrer.

Pour le but 2 :

$$\forall_{x,w,x',w'} \cdot (P_t(x,w) \Rightarrow (Qi_t(x,w,x',w') \Rightarrow Q_t(x,w,x',w'))) \qquad \text{B2}$$

Par définition de Si_t : $Qi_t(x,w,x',w') = Qi(x,x') \wedge Q_t(x,w,x',w')$. Nous obtenons le but :

$$\forall_{x,w,x',w'} \cdot (P_t(x,w) \Rightarrow (Qi(x,x') \wedge Q_t(x,w,x',w') \;\Rightarrow\; Q_t(x,w,x',w'))) \quad \text{B2.1}$$

La preuve de ce but est trivial.

La transformation indiquée dans la formule 2.4 est justifiée par le fait que la variable w n'est pas implantée. Ceci ne modifie pas l'effet de Si sur les variables implantées car la troisième partie de la condition 2.2 garantit que x' ne dépend ni de w ni de w'. Nous pouvons donc choisir n'importe quelle valeur pour w'. Nous prenons une telle que la formule $Q_t(x,w,x',w')$ soit satisfaite. Nous obtenons ainsi Si_t. Le seul risque est que Si_t devienne non faisable, ce qui n'est pas possible pour les états dénotés par $P_t(x,w)$ à cause de la formule 2.3 qui fait partie de la condition de transformation.

Le théorème 2.1 permet d'implanter la spécification initiale transformée S_t avec l'implantation fournie transformée Si_t car les variables auxiliaires ne sont pas implantées. Toute implantation de la spécification initiale S peut donc être transformée à l'aide de la formule 2.4 en une implantation correcte de la spécification transformée S_t sans altérer le comportement de l'implantation initiale.

Si les deux spécifications S et Si sont définies sur des espaces d'états différents x et $x1$, le théorème est toujours valide. Nous avons donc :

Théorème 2.2. *Implantation d'une spécification élémentaire transformée avec des variables auxiliaires. Cas où il y a changement d'espace.*

Si S est raffiné par Si sous la transformation d'états $J(x,x1)$ alors S_t est raffiné par Si_t sous la même transformation d'états. Formellement :

$$S \;\sqsubseteq_{J(x,x1)}\; Si \quad \Rightarrow \quad S_t \;\sqsubseteq_{J(x,x1)}\; Si_t$$

La démonstration de ce théorème peut se faire d'une façon semblable à la précedente mais en utilisant la relation de raffinement avec changement d'espace, définition 1.6, page 32.

Il existe un danger avec la transformation proposée : la spécification qui en résulte peut être non faisable. Ceci est le cas lorsque la formule $Q_t(x,w,x',w')$ ne peut pas être satisfaite pour un des états dénotés par $P_t(x,w)$. Le rôle de la formule 2.3, page 55, est de garantir que ce cas ne se produit pas. Ceci est formulé dans le théorème suivant.

Théorème 2.3. *Faisabilité d'une spécification transformée avec des variables auxiliaires.*

Si S est faisable alors S_t est faisable. Formellement :

$$\begin{aligned}\forall_{x,w} \cdot ((P(x) \;\Rightarrow\; \exists_{x'} \cdot (Q(x,x'))) \;\Rightarrow \\ (P_t(x,w) \;\Rightarrow\; \exists_{x',w'} \cdot (Q_t(x,w,x',w'))))\end{aligned}$$

La formule 2.3 permet de montrer aisément ce théorème.

Dans notre travail la spécification transformée S_t prendra toujours une des deux formes qui suivent :

1. S a la forme $P(x) \Rightarrow Q(x, x')$. S_t prendra la forme [27] :

$$P(x) \wedge Pa(x, w) \Rightarrow Q(x, x') \wedge Qa(x, w, w') \tag{2.5}$$

2. S a la forme : $P(x) \Rightarrow \exists_a \,.\, Q(x, a, x')$. S_t prendra la forme [28] :

$$P(x) \wedge Pa(x, w) \Rightarrow \exists_a \,.\, (Q(x, a, x') \wedge Qa(x, a, w, w')) \tag{2.5a}$$

La forme de ces deux formules permet de montrer que les parties 1 et 2 de la condition 2.2, page 55, sont satisfaites car :

Partie 1 :

$$P(x) \wedge Pa(x, w) \Rightarrow P(x)$$

Partie 2 :

1. Premier cas :

$$Q(x, x') \wedge Qa(x, w, w') \Rightarrow Q(x, x')$$

2. Deuxième cas :

$$\exists_a \,.\, (Q(x, a, x') \wedge Qa(x, a, w, w')) \Rightarrow \exists_a \,.\, Q(x, a, x')$$

Partie 3 : Cette partie est satisfaite si, sous la pré-condition de S_t, pour toute transition admise par la post-condition de S, il existe un état w' tel que la post-condition de S_t peut être satisfaite. Formellement :

1. Premier cas :

$$\forall_{x,w,x'} \,.\, (P(x) \wedge Pa(x, w) \Rightarrow (Q(x, x') \Rightarrow \exists_{w'} \,.\, Qa(x, w, w'))) \tag{2.6}$$

2. Deuxième cas :

$$\forall_{x,a,w,x'} \,.\, (P(x) \wedge Pa(x, w) \Rightarrow (Q(x, a, x') \Rightarrow \exists_{w'} \,.\, Qa(x, a, w, w'))) \tag{2.6a}$$

Dans notre contexte, la transformation d'une spécification élémentaire S peut donc être effectuée à condition qu'elle prenne la forme indiquée dans une des formules 2.5 ou 2.5a et que la formule correspondante 2.6 ou 2.6a soit satisfaite.

Une question peut se poser ici : *pourquoi ne pas donner directement la bonne spécification ?* c'est à dire, celle qui résulte de la transformation que nous venons d'effectuer. Une première réponse comporte un aspect plutôt pratique :

[27] Un exemple de ce cas est illustré par l'opération *Write* du module $ENV1$, page 62.

[28] Des exemples de ce cas sont illustrés par les opérations *Read* et *More* du module $ENV1$, page 62.

parce que la spécification initiale nous a été donnée ; elle a été fournie pour permettre la communication entre un programme quelconque qui est exécuté dans l'environnement qu'elle décrit et le monde extérieur.

Une autre raison est que la spécification d'un module doit contenir uniquement l'information nécessaire à la description de son comportement, de façon à ce qu'il puisse être utilisé dans la solution de problèmes différents. L'ajout de variables auxiliaires permet de l'adapter à chaque situation particulière.

Il est néanmoins possible de fournir directement une spécification qui contient des variables auxiliaires : $S(x, x', w, w')$. Dans ce cas, elle doit prendre la forme indiquée dans la formule 2.5, ou 2.5a, et doit satisfaire la formule 2.6, ou 2.6a. Nous ferons ceci dans l'exemple du chapitre 5, voir section 5.9.5, page 114.

2.9.2 Transformation d'un Module à l'Aide de Variables Auxiliaires

La transformation d'un module peut se faire comme suit :

- Les variables auxiliaires sont déclarées dans le module. Elles peuvent donc être utilisées par toutes les opérations de celui-ci.
- La spécification d'initialisation est transformée pour initialiser les variables auxiliaires de telle sorte que les parties 2 et 3 de la condition 2.2 soient satisfaites. La pré-condition *true* reste inchangée.
- La spécification de chaque opération transformée doit prendre la forme indiquée dans une des formules 2.5 ou 2.5a et doit satisfaire la formule correspondante 2.6 ou 2.6a.

Dans la pratique, nous allons utiliser des variables auxiliaires pour garder les entrées et les sorties reçues et envoyées pendant l'exécution d'un programme. En conséquence, la formule $Qa(x, w, w')$ prendra la forme d'une égalité qui met en relation l'état avant des variables du module avec l'état après des variables auxiliaires.

Nous allons maintenant transformer la spécification du module ENV en une spécification $ENV1$ qui sera utilisée pour implanter le module MAX. $ENV1$ est enrichi avec trois variables auxiliaires In, Out et Con. Ce sont les mêmes variables qui ont servi à spécifier le module MAX. In permet de tracer les valeurs fournies par l'utilisateur. Out permet de tracer les résultats envoyés à l'utilisateur. Et Con permet de garder la dernière réponse donnée par l'utilisateur. Initialement les variables In et Out contiennent la suite vide, et la variable Con est à $false$. Ceci est effectué dans la spécification d'initialisation [29] :

$$true \Rightarrow In' = \emptyset \wedge Out' = \emptyset \wedge Con' = false \wedge I'$$

Les pré-conditions des opérations *Read* et $More$ sont renforcées avec la formule :

$$Con = false \wedge I$$

[29] Dans les formules qui suivent I représente l'invariant du module à l'état avant, et I' représente l'invariant du module à l'état après.

Ceci interdit l'utilisation de ces opérations après que l'ordre d'arrêt soit donné par l'utilisateur.

La pré-condition de l'opération $Write$ est renforcée avec la formule :

$$Con = true \wedge I$$

Ceci interdit l'utilisation de cette opération tant que l'ordre d'arrêt n'a pas été donné par l'utilisateur.

La post-condition de l'opération $Read$ insère dans la variable In les valeurs reçues de l'utilisateur :

$$In' = In \leftarrow a$$

La post-condition de l'opération $Write$ insère dans la variable Out les valeurs envoyées à l'utilisateur :

$$Out' = Out \leftarrow n$$

La post-condition de l'opération $More$ garde dans la variable Con la dernière réponse donnée par l'utilisateur qui est représentée par la variable a :

$$Con' = a$$

Le module $ENV1$ qui en résulte est [30] :

[30] La notion de variable auxiliaire n'existe pas en B. La spécification correspondante du module $ENV1$ en B est :

MACHINE
ENV1

VARIABLES
In, Out, Con

INVARIANT
$In \in seq(INT) \wedge$
$Out \in seq(INT) \wedge$
$Con \in BOOL$

INITIALIZATION
$In, Out, Con := \emptyset, \emptyset, false$

OPERATIONS
$z \longleftarrow$ Read $\hat{=}$
$PRE \quad Con = false \quad THEN$
$\quad ANY \quad a \quad WHERE \quad a \in INT \quad THEN \quad z, In := a, In \leftarrow a \quad END;$

Write(n) $\hat{=}$
$PRE \quad Con = true \wedge n \in INT \quad THEN \quad Out := Out \leftarrow n \quad END;$

$c \longleftarrow$ More $\hat{=}$
$PRE \quad Con = false \quad THEN$
$\quad ANY \quad a \quad WHERE \quad a \in BOOL \quad THEN \quad Con := a \quad END$

MODULE	
ENV1	Env1 est la transformation de ENV.
VARIABLES AUXILIAIRES	
$\mathbf{In}, \mathbf{Out}, \mathbf{Con}$	In, Out et Con sont des variables auxiliaires.
INVARIANT	
$\mathbf{In} \in \mathbf{seq}(\mathbf{INT}) \wedge$	In est une suite d'entiers qui représente les entiers reçus.
$\mathbf{Out} \in \mathbf{seq}(\mathbf{INT}) \wedge$	Out est une suite d'entiers qui représente les sorties envoyées
$\mathbf{Con} \in \mathbf{BOOL}$	Con est un booléen qui représente la condition d'arrêt du programme.
INITIALIZATION	
$\mathbf{true} \Rightarrow$	Initialement :
$\mathbf{In}' = \emptyset \wedge$	Il n'y a pas d'entrée reçue.
$\mathbf{Out}' = \emptyset \wedge$	Il n'y a pas de sortie envoyée.
$\mathbf{Con}' = \mathbf{false} \wedge$	La condition d'arrêt est $false$.
$\mathbf{I}'$	L'invariant est établit.
OPERATIONS	
$z \longleftarrow$ Read $\hat{=}$	
$\mathbf{Con} = \mathbf{false} \wedge \mathbf{I} \Rightarrow$	L'ordre d'arrêt n'a pas été donnée.
$\exists_a \cdot (a \in INT \wedge z' = a \wedge$	z' ne dépend pas de In, Out et Con.
$\mathbf{In}' = \mathbf{In} \leftarrow \mathbf{a} \wedge \mathbf{I}')$;	La valeur reçue est insérée dans In.
Write(n) $\hat{=}$	
$\mathbf{Con} = \mathbf{true} \wedge \mathbf{I} \wedge n \in INT \Rightarrow$	L'ordre d'arrêt a été donnée.
$\mathbf{Out}' = \mathbf{Out} \leftarrow \mathbf{n} \wedge \mathbf{I}'$;	La valeur envoyée est insérée dans Out.
$c \longleftarrow$ More $\hat{=}$	
$\mathbf{Con} = \mathbf{false} \wedge \mathbf{I} \Rightarrow$	L'ordre d'arrêt n'a pas été donnée.
$\exists_a \cdot (a \in BOOL \wedge c' = a \wedge$	
$\mathbf{Con}' = \mathbf{a} \wedge \mathbf{I}')$	La réponse donnée est gardée dans Con.

A ce point, nous avons la spécification de trois modules : MAX, ENV et $ENV1$. Dans MAX nous avons utilisé les variables abstraites In, Out et Con pour spécifier le comportement du programme souhaité. ENV spécifie le comportement de l'environnement et fournit l'interface que le module MAX doit utiliser pour interagir avec son environnement. $ENV1$ est le résultat de la transformation que nous avons effectuée sur ENV. Il restreint l'utilisation que le module MAX peut faire du module ENV, à l'aide des variables auxiliaires In, Out et Con.

Il est possible de restreindre davantage le module ENV pour interdire d'autres traces d'opérations non souhaitées. Par exemple, pour spécifier qu'une lecture doit être suivie d'une question puis, s'il y a d'autres entiers à évaluer, d'une autre lecture, sinon de l'envoi d'un résultat. Pour cela, il suffit d'utiliser une autre variable auxiliaire qui enregistre la dernière opération appelée et qui permet de contrôler l'ordre des appels aux opérations du module $ENV1$. Avec cette nouvelle variable, nous pouvons renforcer d'avantage les pré-conditions du module $ENV1$ ce qui a pour conséquence la réduction de l'ensemble de traces que ce module admet.

Cette démarche est très semblable au renforcement de gardes dans l'approche par événements que nous présentons dans le chapitre 5, page 97. La différence est qu'ici la réduction de traces est effectuée à l'aide de variables auxiliaires car

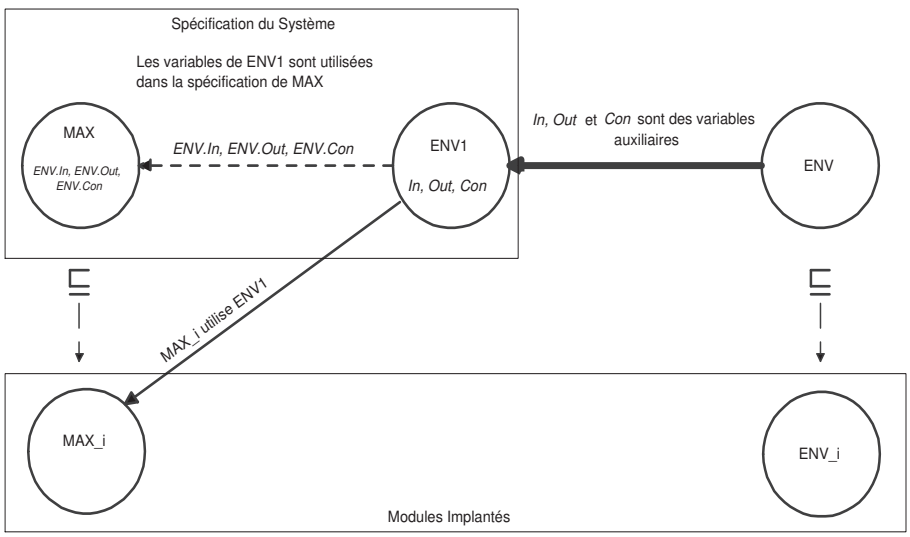

Figure 2.4 – Utilisation de Spécifications de Module Transformées

l'implantation du module nous a déjà été donnée. Dans le cas du chapitre 5 elle est effectuée à l'aide de la relation de raffinement car le but est de construire graduellement la spécification d'un système.

Pour compléter notre spécification, nous devons indiquer le lien qui existe entre les variables des modules MAX et $ENV1$. La spécification de ces liens est importante parcequ'elle permet de différencier les variables qui peuvent être raffinées librement de celles qui doivent être implantées avec les variables d'autres modules.

Pour cela, nous allons indiquer explicitement dans la spécification d'un module les variables qui doivent impérativement être implantées avec les variables d'autres modules. Dans l'exemple, nous devons indiquer que les variables In, Out et Con du module MAX doivent être implantées avec celles du module $ENV1$ qui ont les mêmes noms. Pour donner cette information dans la spécification du module MAX nous écrirons dans la partie $VARIABLES$:
$ENV.In, ENV.Out, ENV.Con$ à la place de In, Out, Con. La figure 2.4 illustre ces idées. Le fait d'imposer l'implantation d'une variable avec celles d'un autre module restreint son raffinement au seul cas qui effectue précisément cette implantation.

Le module MAX est donc implanté avec le module $ENV1$. Mais de quelle façon pouvons nous implanter le module $ENV1$? Le théorème 2.1 fourni la réponse à cette question : Dans le programme final, nous utilisons l'implantation fournie pour le module ENV. Les variables auxiliaires ne sont pas implantées, voir figure 2.4.

2.9.3 Raffinement de Variables Auxiliaires

En théorie, il est possible d'envisager le raffinement de variables auxiliaires à condition de les raffiner ou de les implanter avec d'autres variables auxiliaires. Nous n'utiliserons pas ce type de raffinement.

2.10 Propriétés de Vivacité

Les propriétés que nous pouvons exprimer dans l'invariant d'un module portent sur les états qu'il peut atteindre, mais elles ne font pas intervenir plusieurs états. Ce sont les propriétés de *sûreté*.

Avec une opération, il est possible d'exprimer des propriétés qui font intervenir deux états consécutifs du module : avant et après l'exécution de l'opération.

Il existe d'autres propriétés importantes qui font intervenir plusieurs états d'un module et qui par conséquent ne peuvent être exprimées ni dans l'invariant ni dans les opérations du module. Elles sont appelées de *vivacité* car elles expriment le comportement souhaité du système [Cha,Mis 88, Lam,Sha 90, Man,Pnu 84, Man,Pnu 92, Sch 99].

Dans notre exemple, la post-condition de l'opération $Maximum$ du module MAX : $size(Out') = 1 \wedge Out'(1) = max(ran(In')) \wedge Con' = true$ est une propriété de vivacité du module $ENV1$ car elle spécifie qu'à la fin de l'exécution de l'opération $Maximum$ les variables du module $ENV1$ doivent satisfaire la formule indiquée.

Il est clair que cette propriété n'est pas satisfaite par le module $ENV1$ tel que nous l'avons spécifié, car il y a des traces qu'il admet qui ne la satisfont pas. Par exemple, la trace : $Read, Read, Read, More, Write, Write, Write$ est admise par $ENV1$, ce qui mène à un état où $size(Out') = 3$. La méthode proposée nous permet donc de laisser de côté tous les détails d'implantation. Nous avons juste exprimé la propriété de vivacité souhaitée.

Il serait possible d'ajouter d'autres variables auxiliaires et de renforcer les pré-conditions des opérations de $ENV1$ pour faire en sorte que le module n'admette que les traces que nous souhaitons. Dans ce cas, nous serons amenés à exprimer les propriétés de vivacité souhaitées à l'aide *d'opérateurs temporels*, voir section 5.1, page 98, puis à prouver que le module satisfait la propriété. La preuve de ce type de propriété est souvent très difficile à réaliser car nous devons montrer que toutes les traces que le module admet la satisfont. Ajouter des contraintes supplémentaires dans les pré-conditions des opérations du module $ENV1$ reviendrait indirectement à construire l'algorithme du programme que nous souhaitons implanter : $Maximum$. Cette démarche nous paraît inadaptée car nous souhaitons seulement spécifier le comportement du programme. Nous ne souhaitons pas donner son implantation.

Nous avons adopté une démarche très différente. Nous avons complété le module ENV avec les variables dont nous avons besoin pour exprimer la propriété de vivacité souhaitée. Nous avons obtenu $ENV1$. La propriété de vivacité a été exprimée à l'extérieur du module $ENV1$ dans l'opération $Maximum$ du

module MAX. De cette façon, nous pouvons laisser beaucoup plus de liberté au module $ENV1$ car ce que nous voulons est uniquement qu'il admette les traces souhaitées. La sélection des traces qui satisfont la propriété de vivacité sera effectuée dans l'implantation de l'opération $Maximum$. La preuve que cette propriété est satisfaite sera fournie lors du processus d'implantation de l'opération $Maximum$.

Cette technique apporte une réponse claire à la question de la structuration d'une spécification car elle permet d'exprimer les propriétés d'un système en accord avec les composants utilisés dans son implantation. Il n'est donc pas nécessaire de regrouper la totalité de la spécification en un seul module.

Une question se pose ici : *Existe-t-il des propriétés de vivacité d'un module qui doivent nécessairement être exprimées à l'intérieur de ce même module ?*

Ces propriétés apparaissent quand nous nous intéressons à toutes les suites d'opérations que le module admet. Par exemple, pour exprimer le fait que toute suite d'appels aux opérations du module est nécessairement finie. Elles décrivent des comportements invariants du module. Pour les exprimer il est nécessaire d'introduire des opérateurs temporels [Abr,Mus 98]. Nous ne le ferons pas dans ce travail.

2.11 Opérations Auxiliaires

D'une façon semblable aux variables auxiliaires, il est parfois utile d'inclure dans les spécifications des opérations qui ne seront pas implantées. Nous les appellerons auxiliaires. Leur but est de permettre la modification de variables auxiliaires à des instants précis de l'exécution du programme.

Ces opérations ne peuvent modifier que des variables auxiliaires ; c'est à dire, qu'elles n'affectent pas les variables abstraites et les variables locales du module. Elles ne génèrent pas de code exécutable.

Pour illustrer cette idée nous allons inclure dans la spécification du module MAX la possibilité d'effectuer le calcul pour plusieurs suites d'entiers. Ceci veut dire qu'une fois que le programme a envoyé un résultat, il doit demander à l'utilisateur s'il souhaite calculer le maximum pour une autre suite d'entiers. Si la réponse est "oui", il doit commencer un nouveau cycle de calcul. Si la réponse est "non", il doit s'arrêter. Le programme doit donc adopter un comportement cyclique. Dans chaque cycle, il doit calculer le maximum d'une suite d'entiers.

La spécification d'un cycle de calcul est donnée par l'opération $Maximum$ du module MAX. Nous allons transformer le module $ENV1$ afin de tenir compte de cette nouvelle problématique. Pour cela, nous allons ajouter une opération auxiliaire nommée *BeginCycle*, qui permet d'initialiser les variables In, Out et Con pour chaque nouveau cycle de calcul. Nous allons appeler ce nouveau module $ENV2$. Il sera le résultat d'une transformation correcte du module ENV adaptée à la solution de ce nouveau problème.

Nous avons besoin d'introduire une nouvelle variable auxiliaire : Num, qui permet de forcer l'appel à l'opération *BeginCycle* dans chaque cycle de calcul. Num permet de compter les cycles exécutés. Son type est NAT. Initialement,

elle contient la valeur 0 car au départ aucun cycle n'a été exécuté.

La spécification de l'opération *BeginCycle* initialise les variables auxiliaires pour chaque nouveau cycle, comme suit :

BeginCycle $\hat{=}$	
$I \Rightarrow$	L'opération peut être appelée à tout moment.
$In' = \emptyset \wedge$	Au début du cycle In est vide ; aucune entrée n'a été reçue.
$Out' = \emptyset \wedge$	Au début du cycle Out est vide ; aucune sortie n'a été envoyée.
$Con' = false \wedge$	Au début du cycle Con est $false$; le cycle commence.
$Num' = Num + 1$	Un nouveau cycle est enregistré.

Nous devons prévoir une opération qui permette au programme de questionner l'utilisateur sur son désir de démarrer un autre cycle de calcul. Ceci peut être fait avec une opération semblable à *More* et une autre variable auxiliaire. Nous ne le ferons pas ici. Nous supposons que l'opération $Maximum$ est appelée par un autre programme lorsque l'utilisateur a décidé d'effectuer un nouveau cycle.

La spécification complète du module $ENV2$ est :

MODULE	
ENV2	Env2 est la transformation de ENV pour un programme cyclique.
VARIABLES AUXILIAIRES	
In, Out, Con, Num	In, Out, Con et Num sont des variables auxiliaires.
INVARIANT	
$In \in seq(INT) \wedge$	In est une suite d'entiers qui représente les entiers reçus pendant un cycle.
$Out \in seq(INT) \wedge$	Out est une suite d'entiers qui représente les sorties envoyées pendant un cycle.
$Con \in BOOL \wedge$	Con est un booléen qui représente la condition d'arrêt d'un cycle.
$Num \in NAT$	Num est un naturel qui représente le nombre de cycles exécutés.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Num' = 0 \wedge$	Aucun cycle n'a été exécuté.
$Con' = true \wedge$	**Il n'y a pas de cycle en cours.**
I'	L'invariant est établit.
OPERATIONS	
$z \longleftarrow$ Read $\hat{=}$	
$Con = false \wedge I \Rightarrow$	L'ordre d'arrêt n'a pas été donnée.
$\exists_a \cdot (a \in INT \wedge z' = a \wedge$	z' ne dépend pas de In, Out et Con.
$In' = In \leftarrow a \wedge I')$;	La valeur reçue est insérée dans In.
Write(n) $\hat{=}$	
$Con = true \wedge I \wedge n \in INT \Rightarrow$	L'ordre d'arrêt a été donnée.
$Out' = Out \leftarrow n \wedge I'$;	La valeur envoyée est insérée dans Out.
$c \longleftarrow$ More $\hat{=}$	
$Con = false \wedge I \Rightarrow$	L'ordre d'arrêt n'a pas été donnée.
$\exists_a \cdot (a \in BOOL \wedge c' = a \wedge$	
$Con' = a \wedge I')$	La réponse donnée est gardée dans Con.

OPERATIONS AUXILIAIRES	
BeginCycle $\hat{=}$	
In, **Out**, **Con**, **Num** : (**I** $\Rightarrow$	
In$'$ $= \emptyset \wedge$	Au début du cycle In est vide.
Out$'$ $= \emptyset \wedge$	Au début du cycle Out est vide.
Con$'$ $=$ **false** $\wedge$	Au début du cycle Con est $false$.
Num$'$ $=$ **Num** $+ 1$)	Un nouveau cycle est enregistré.

Le programme MAX doit prendre en compte l'initialisation de chaque cycle de calcul. Pour cela, il suffit d'ajouter dans la post-condition de l'opération *Maximum* la formule $Num' = Num + 1$ qui force l'utilisation de l'opération *BeginCycle* du module $ENV2$ car c'est le seul moyen qui existe pour incrémenter la variable Num. Plus que cela, *BeginCycle* ne peut être utilisée qu'une seule fois pendant un cycle car dans le cas contraire Num serait incrémentée plusieurs fois, ce qui est en désaccord avec la spécification $Num' = Num + 1$. Le module MAX doit donc inclure la variable Num.

La spécification de l'opération $Maximum$ qui en résulte est :

Maximum $\hat{=}$	
$I \Rightarrow$	
$size(Out') = 1 \wedge$	A la fin de chaque cycle :
$Out'(1) = max(ran(In')) \wedge$	Une seule sortie a été envoyée.
	Le résultat envoyé est le maximum des entiers reçus.
$Con' = true \wedge$	La condition d'arrêt est vrai.
Num$'$ $=$ **Num** $+ 1 \wedge$	Un nouveau cycle a été calculé.
I'	L'invariant est rétablit.

Une fois que l'opération *BeginCycle* a été appelée, le programme doit compléter le cycle de calcul puisque la post-condition de l'opération $Maximum$ spécifie que la variable Con doit être à $true$ à la fin, ce qui est possible uniquement en appelant l'opération $More$. A ce point, deux cas sont possibles :

1. l'utilisateur indique qu'il n'y a plus d'entrée. Dans ce cas la variable Con est affectée avec la valeur $true$, ce qui interdit tout autre appel aux opérations $Read$ et $More$. La seule possibilité qui reste est l'appel à l'opération $Write$, ce qui doit être effectué puisque sinon la variable Out contiendrait la séquence vide, ce qui est interdit par la spécification $size(Out) = 1$.
2. l'utilisateur indique qu'il y a d'autres entrées. Dans ce cas la variable Con reste à $false$, et une nouvelle valeur doit être lue.

Une question importante se pose ici : que se passe-t-il si l'utilisateur ne décide jamais de terminer le cycle en cours ? Ce comportement est admissible par la spécification que nous avons donnée. La réponse à cette question est : **le programme ne termine jamais**. Il est donc impossible de montrer la terminaison de l'opération $Maximum$. Nous devons donc faire une hypothèse supplémentaire sur l'environnement : pendant un cycle le nombre d'entiers de la suite à évaluer est fini. Cette hypothèse peut être modélisée à l'aide d'une autre variable auxiliaire, $Size$, qui détermine pour chaque cycle le nombre maximum d'entiers à évaluer. Ce nombre peut être aussi grand que l'on veut, mais fini. Il sert uniquement à faire la preuve de terminaison de l'opération $Maximum$.

Pour prendre en compte cette hypothèse, l'opération *BeginCycle* doit affecter une valeur supérieur à 1 à la variable $Size$, et l'opération *Read* doit la décrémenter de 1. Il sera donc possible d'implanter $Maximum$ avec une boucle dont le corps lit d'abord une valeur, puis la compare au maximum calculé jusqu'au moment de la lecture, et finalement demande à l'utilisateur si la suite est terminée. Le *variant* de la boucle sera précisément $Size$.

2.12 Conclusion

Le comportement d'un programme complexe ne peut pas être spécifié avec une simple spécification élémentaire sans faire d'hypothèse simplificatrice sur ses interactions avec son environnement. Le problème principal vient du fait qu'une spécification élémentaire demande que les entrées soient acquises au début de l'exécution du programme et les sorties envoyées à la fin. Il est donc impossible de spécifier de cette façon des programmes qui doivent interagir avec leur environnement tout au long de leur exécution. La notion de module permet de spécifier ces interactions car un module offre plusieurs opérations qui peuvent être utilisées à différents instants de l'exécution d'un programme.

Nous proposons une approche où la spécification d'un programme est formée de au moins deux modules. Le premier contient une seule opération qui permet de spécifier les propriétés de vivacité du programme. Ce module est spécifié comme un univers clos, c'est à dire que ses opérations n'ont pas de paramètre. Le ou les autres modules spécifient le comportement de l'environnement où le programme doit être exécuté. Ils contiennent toutes les hypothèses qui sont faites sur l'environnement, et fournissent au programme l'interface de communication avec son environnement.

Nous utilisons des variables auxiliaires, non implantables, pour compléter la spécification d'un module. Ces variables permettent de spécifier des propriétés qui ne peuvent pas être exprimées directement avec les variables de la spécification donnée initialement. Elles sont utilisées principalement pour enregistrer les entrées reçues et les sorties envoyées, et pour restreindre l'utilisation qui peut être faite des opérations d'un module fourni et déjà implanté. Ce mécanisme permet de spécifier des propriétés sur des entrées et des sorties abstraites d'un programme. Les variables auxiliaires ne sont pas raffinées. Elles sont simplement ignorées pendant le processus d'implantation de la spécification.

Les opérations auxiliaires élargissent considérablement le cadre d'utilisation des variables auxiliaires car elles permettent de modifier ces variables à des instants précis de l'exécution d'un programme. Les opérations auxiliaires ne génèrent pas de code car elles ne sont pas implantées.

Les spécifications de module contient des restrictions sur la façon d'implanter leurs variables, en indiquant les variables d'autres modules qui doivent impérativement être utilisées pour le faire. Ce mécanisme permet d'établir les liens qui existent entre les différents modules qui forment la spécification du programme.

Les variables auxiliaires et les opérations auxiliaires fournissent une bonne réponse aux problèmes de la réutilisation et de la structuration d'une spécifi-

cation, car elles permettent l'adaptation d'une spécification de propos général dans la solution de problèmes particuliers. Les formules 2.5 et 2.6 fournissent un cadre pratique d'utilisation de ce type de variables dans la solution de problèmes réels.

Il est légitime de se demander si la spécification de l'environnement ne comporte pas de notions de trop bas niveau pour être considérée comme une partie de la spécification du programme. Nous pensons qu'elle est indispensable car la description du comportement souhaité s'appuie sur les observations que nous pouvons faire du système, ce qui est étroitement lié aux interactions du programme avec son environnement.

Chapitre 3

Modules Partagés

Le modules peuvent aussi être utilisés pour spécifier les mécanismes de communication entre programmes ou entre différents composants [1] d'un système. Dans ce chapitre, nous allons développer cette idée.

Il existe une différence fondamentale entre la spécification d'un programme isolé et celle d'un système formé par plusieurs composants. Dans le premier cas, les variables utilisées dans la spécification sont modifiées uniquement par le programme; il n'y a pas d'interférence possible avec d'autres composants. Dans le deuxième cas, ces interférences existent. Nous devons donc les prendre en compte.

Les systèmes comportant plusieurs composants [2] peuvent être regroupées en deux classes : centralisés et distribués. La première classe contient ceux dont les informations sur leur état se trouvent localisées à un même endroit et sont accessibles par tous les composants. Ici la communication est effectuée à l'aide de variables partagées. La deuxième classe contient les systèmes dont les informations sur leur état se trouvent distribuées à plusieurs endroits. Dans ce cas, la communication est effectuée par envoi et réception de messages.

Pour spécifier un système de ce type, nous allons fournir la spécification d'un module qui contient des opérations atomiques qui peuvent être utilisées par tous ses composants. En conséquence, les variables de ce module ne peuvent être modifiées que par une seule opération à la fois [3] . L'entrelacement de ces opérations avec les opérations des modules qui implantent les différents composants du système permet de prendre en compte les aspects concurrentes du système.

[1] Nous utiliseront le mot *composant* pour faire référence à une partie du système qui réalise une fonction bien déterminée. Il peut s'agir d'une partie matériel, par exemple un capteur, ou d'une partie logiciel, par exemple un contrôleur.

[2] Un tel système est souvent formé par plusieurs agents qui doivent interagir entre eux soit par partage de variables soit par échange de messages.

[3] Cette idée correspond à la notion de moniteur d'Hoare.

3.1 Un Système de Transfert de Fichiers

Pour présenter ces idées, nous allons étudier et spécifier un système distribué qui permet de transférer un fichier Fa d'un site A à un site B. Ce système est formé par un processus émetteur, un processus récepteur, et un mécanisme de transfert d'information entre les deux sites : le canal.

Le processus émetteur est spécifié avec deux modules : un qui définit sa propriété de vivacité : *Sender*, et un autre qui décrit l'environnement qui lui est fourni pour effectuer le transfert : *EnvS*.

Le processus récepteur est spécifié également avec deux modules : un qui définit sa propriété de vivacité : *Receiver*, et un autre qui décrit l'environnement qui lui est fourni pour recevoir les messages envoyés par l'émetteur : *EnvR*. La figure 3.1 illustre cette spécification.

Les variables utilisées dans $EnvS$ sont différentes de celles utilisées dans $EnvR$. Il n'y a donc pas d'interférence possible entre ces deux modules. Néanmoins, il existe bien une relation entre les informations envoyées par l'émetteur et celles reçues par le récepteur. Ce lien ne peut être établi qu'en modélisant le canal de communication. Pour cela, nous allons fusionner les modules *EnvS* et *EnvR* en un seul module *Channel* qui sera partagé par les deux processus : émetteur et récepteur. Le module qui en résulte sera complété avec les propriétés du canal. La figure 3.2 illustre cette idée.

3.1.1 Spécification de l'Emetteur

Le fichier à envoyer est représenté par une suite de messages. L'émetteur doit d'abord prendre le premier message et l'envoyer accompagné d'une information de contrôle qui sert à indiquer au récepteur l'état du transfert : terminé ou non. $false$ indique que le transfert n'est pas terminé et $true$ indique qu'il est terminé. Si l'envoi n'est pas terminé, l'émetteur doit envoyer le message suivant accompagné du message de contrôle correspondant. Le processus se termine quand tous les messages ont été envoyés. Le dernier message de contrôle doit contenir $true$. Tous les autres doivent contenir $false$.

La spécification du module $Sender$ utilise trois variables abstraites : Fa, Fs et Cs. Fa représente le fichier à envoyer. Son type est $seq(MES)$ [4] . Fs représente la suite de messages envoyés. Son type est aussi $seq(MES)$. Cs représente la suite de messages de contrôle envoyés. Son type est $seq(BOOL)$. Fs et Cs sont des variables du module $ENVs$ qui sont utilisées ici pour spécifier l'opération de transfert.

Dans l'invariant nous allons exprimer que le fichier à envoyer n'est pas vide : $size(Fa) > 0$, et que le nombre de messages envoyés est égal au nombre de messages de contrôle envoyés : $size(Cs) = size(Fs)$.

$Sender$ a une seule opération, $Transfert$, qui effectue le transfert du fichier du côté émetteur. La pré-condition est l'invariant car le transfert peut être déclenché à tout moment. Dans la post-condition nous allons exprimer :

[4] Nous supposons que MES est un type préalablement défini qui représente les messages qui peuvent être envoyés.

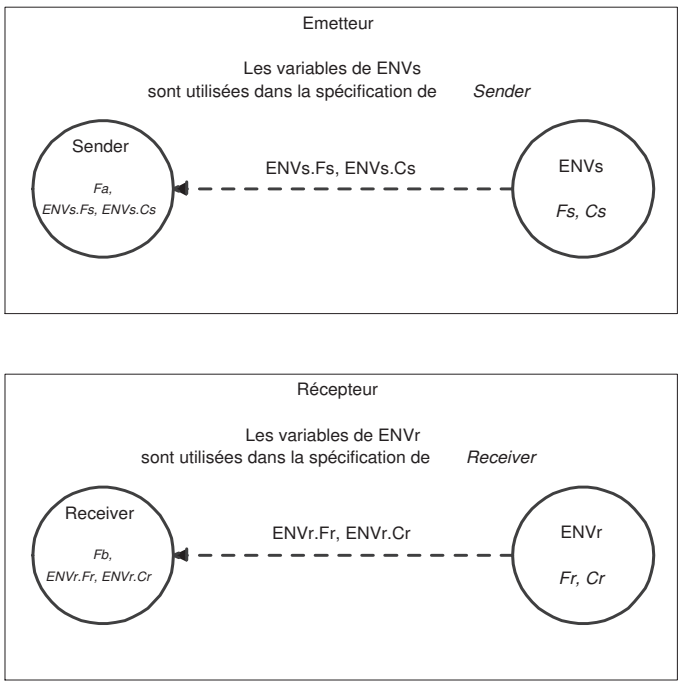

Figure 3.1 – Spécification d'un Système de Transfert de Fichiers

- que le fichier d'origine a été entièrement envoyé : $Fs' = Fa$,
- que le fichier d'origine n'est pas modifié : $Fa' = Fa$,
- que seulement le dernier message de contrôle contient $true$ [5] :
  $Cs' \uparrow size(Fa) - 1 \in seq(\{false\}) \wedge last(Cs') = true$.

L'opération *Transfert* effectue l'envoi du fichier Fa d'un seul coup. Les détails sur la façon de le faire apparaissent lors de son implantation sous la forme d'appels aux opérations du module $EnvS$. La spécification qui en résulte est [6] :

MODULE	
Sender	Processus émetteur.
VARIABLES	
Fa,	Fichier à envoyer.
$ENVs.Fs$,	Suite de messages envoyés.
$ENVs.Cs$	Suite de messages de contrôle envoyés.

[5] L'expression $S \uparrow n$ dénote la suite formée par les premiers n éléments de la suite S pris dans l'ordre [Abr 96-1].
L'expression $last(Cs')$ dénote le dernier élément de la suite Cs'.

[6] La spécification correspondante en B est :

INVARIANT	
$Fa \in seq(MES) \wedge$	Fa est une suite finie de messages.
$size(Fa) > 0 \wedge$	Fa est une suite non vide.
$ENVs.Fs \in seq(MES) \wedge$	Fs est la suite de messages envoyés.
$ENVs.Cs \in seq(BOOL) \wedge$	Cs est la suite de messages de contrôle.
$size(ENVs.Cs) = size(ENVs.Fs)$	Nombre de messages de données = Nombre de messages de contrôle.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Fa' \in seq(MES) \wedge$	Fa est une suite finie de messages.
$ENVs.Fs' = \emptyset \wedge$	Aucun message n'a été envoyé.
$ENVs.Cs' = \emptyset \wedge$	Aucun message de contrôle
I'	n'a été envoyé. L'invariant est établit.
OPERATIONS	
Transfert $\hat{=}$	
$I \Rightarrow$	A la fin :
$ENVs.Fs' = Fa \wedge$	Le fichier initial a été envoyé.
$Fa' = Fa \wedge$	Le fichier initial n'est pas modifié.
$ENVs.Cs' \uparrow size(Fa) - 1 \in seq(\{false\}) \wedge$	Uniquement le dernier message
$last(ENVs.Cs') =$true$\wedge$	de contrôle contient $true$.
I'	L'invariant est rétablit.

Le module $EnvS$ fournit une opération $Send(m, c)$ qui permet d'envoyer un message au récepteur [7]. Elle prend deux paramètres : m qui contient le message à envoyer et c qui contient le message de contrôle. Le type de m est MES et le type de c est $BOOL$.

Le module contient deux variables auxiliaires Fs et Cs qui permettent d'enregistrer les messages envoyés. Les variables Fs et Cs du module $Sender$ doivent être implantées avec les variables Fs et Cs du module $EnvS$. Ceci est illustré dans la figure 3.1, page 73. La variable Fa du module $Sender$ doit être implantée avec la variable Fa d'un autre module qui permet d'accéder au fichier de messages à copier. Nous ne donnons pas ici cette spécification.

Le module $EnvS$ peut être vu comme une librairie d'entrée-sortie de mes-

MACHINE
Sender

SETS
MES

VARIABLES
Fa, Fs, Cs

INVARIANT
$Fa \in seq(MES) \wedge size(Fa) > 0 \wedge Fs \in seq(MES) \wedge$
$Cs \in seq(BOOL) \wedge size(Cs) = size(Fs)$

INITIALIZATION
$Fa, Fs, Cs : (Fa \in seq(MES) \wedge Fs = \emptyset \wedge Cs = \emptyset)$

OPERATIONS
Transfert $\hat{=}$
$Fa, Fs, Cs : (Fs = Fa\_0 \wedge Fa = Fa\_0 \wedge$
$Cs \uparrow size(Fa) - 1 \in seq(\{false\}) \wedge last(Cs') = true)$

[7] Dans les spécification de protocoles de communication, il est classique d'utiliser une opération d'envoi de messages $Send$ et des opérations de réception de messages Get et $More$.

sages. Il peut être utilisée dans la solution d'autres problèmes.

La spécification de *EnvS* est : [8] :

MODULE	
EnvS	Interface de communication fournie au module *Sender*.
VARIABLES AUXILIAIRES	
Fs,	Suite de messages envoyés.
Cs	Suite de messages de contrôle envoyés.
INVARIANT	
$Fs \in seq(MES) \wedge$	Fs est une suite de messages.
$Cs \in seq(BOOL)$	Cs est une suite de booléens.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Fs' = \emptyset \wedge$	Aucun message n'a été envoyé.
$Cs' = \emptyset \wedge$	Aucun message de contrôle n'a été envoyé.
I'	L'invariant est établit.
OPERATIONS	
Send$(m, c) \triangleq$	
$m \in MES \wedge c \in BOOL \Rightarrow$	
$Fs' = Fs \leftarrow m \wedge$	Un message est envoyé.
$Cs' = Cs \leftarrow c \wedge I'$	Un message de contrôle est envoyé.
End	

3.1.2 Spécification du Récepteur

Du côté récepteur, le fichier est reçu message par message. D'abord il faut lire le premier message avec l'information de contrôle correspondante. Si l'envoi n'est pas terminé, le récepteur doit lire le message suivant accompagné du message de contrôle. Le processus se termine quand tous les messages ont été lus. C'est à dire quand un message de contrôle contient *true*. Ce comportement impose qu'au moins un message soit lu.

La spécification du module *Receiver* utilise trois variables abstraites : Fb, Fr et Cr. Fb représente la copie du fichier reçu. Son type est $seq(MES)$. Fr re-

[8] La spécification du module *EnvS* en B est :

MACHINE
EnvS

SETS
MES

VARIABLES
Fs, Cs

INVARIANT
$Fs \in seq(MES) \wedge Cs \in seq(BOOL)$

INITIALIZATION
$Fs, Cs := \emptyset, \emptyset$

OPERATIONS
Send$(m, c) \triangleq$
$PRE \quad m \in MES \wedge c \in BOOL \quad THEN \quad Fs, Cs := Fs \leftarrow m, Cs \leftarrow c \quad END$

présente la suite de messages reçus. Son type est aussi $seq(MES)$. Cr représente la suite de messages de contrôle reçus. Son type est $seq(BOOL)$. Fr et Cr sont des variables du module $ENVr$ qui sont utilisées ici pour spécifier l'opération de réception.

Dans l'invariant nous allons exprimer que la *copie* créée du côté récepteur est identique au fichier reçu : $Fb = Fr$. Nous ne pouvons pas exprimer ici la propriété souhaitée pour l'ensemble du système, c'est à dire que le fichier créé du côté récepteur est identique au fichier d'origine qui se trouve du côté émetteur, car nous n'avons pas une vision d'ensemble du système. Nous feront ceci dans la section suivante.

Receiver a une seule opération, *Reception*, qui reçoit le fichier envoyé par l'émetteur. La pré-condition est l'invariant car la réception peut être déclenchée à tout moment. Dans la post-condition nous allons exprimer :

- qu'au moins un message est arrivé : $size(Cr') > 0$,
- qu'uniquement le dernier message de contrôle contient *true*
  $Cr' \uparrow size(Cr') - 1 \in seq(\{false\}) \wedge last(Cr') = true$.

L'opération *Receiver* effectue la réception du fichier Fr d'un seul coup. Les détails sur la façon de le faire apparaissent lors de son implantation sous la forme d'appels aux opérations du module $EnvR$.

La spécification est [9] :

MODULE	
Receiver	Processus récepteur.
VARIABLES	
Fb,	Copie du fichier reçu.
$ENVr.Fr$,	Fichier reçu.
$ENVr.Cr$,	Suite de messages de contrôle reçus.
INVARIANT	
$Fb \in seq(MES) \wedge$	Fb est une suite de messages.
$ENVr.Fr \in seq(MES) \wedge$	Fr est une suite de messages.
$ENVr.Cr \in seq(BOOL) \wedge$	Cr est une suite de messages de contrôle.
$Fb = ENVr.Fr$	**Propriété 1.** La copie créée est identique au fichier reçu.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Fb' = \emptyset \wedge$	La copie n'a pas commencée.
$ENVr.Fr' = \emptyset \wedge$	Aucun message n'a été reçu.
$ENVr.Cr' = \emptyset \wedge$	Aucun message de contrôle n'a été reçu.
I'	L'invariant est établit.
OPERATIONS	
Reception $\hat{=}$	
$I \Rightarrow$	A la fin :
$size(ENVr.Cr') > 0 \wedge$	**Au moins un message est arrivé.**
$(ENVr.Cr' \uparrow size(ENVr.Cr') - 1 \in seq(\{false\})) \wedge$	Uniquement le dernier message de contrôle contient $true$.
$last(ENVr.Cr') =$ true $\wedge$	
I'	L'invariant est rétablit.

Le module $EnvR$ fournit deux opérations : $More$ et Get.

[9] La spécification en B de ce module est très semblable à celle du module $Sender$. Nous ne la donnons pas ici.

- $More$ permet de vérifier si des nouveaux messages sont arrivés. Elle rend le résultat r qui indique si des messages sont en attente de lecture : $false$ indique qu'il n'y a pas de message en attente, $true$ indique qu'il y a des messages en attente.
- Get permet de recevoir un message. Elle peut être appelée uniquement lorsque des messages sont en attente de lecture. Elle rend deux résultats : m qui contient le message reçu et c qui contient l'information de contrôle qui indique l'état du transfert : terminé ou non. Le type de m est MES et le type de c est $BOOL$.

Ce module contient trois variables auxiliaires. Fr et Cr permettent d'enregistrer les messages reçus par le récepteur. Nm sert à indiquer que des nouveaux messages sont arrivés. Cette variable est utilisée pour écrire la pré-condition de l'opération Get afin d'interdire son appel lorsqu'il n'y a pas de message en attente. Cette technique évite d'adopter un style *défensif* dans la spécification de l'opération Get. Ceci illustre l'apport de la preuve dans le processus de construction de programmes. Sans cette variable, il serait nécessaire de prévoir les cas de mauvaise utilisation pour l'opération Get.

La spécification de $EnvR$ est [10] :

MODULE	
EnvR	Interface de communication fournie au récepteur.
VARIABLES AUXILIAIRES	
Fr,	Suite de messages reçus.
Cr,	Suite de messages de contrôle reçus.
Nm	Sert à indiquer si des messages sont en attente.
INVARIANT	
$Fr \in seq(MES) \wedge$	Fr est une suite de messages.
$Cr \in seq(BOOL) \wedge$	Cr est une suite de booléens.
$Nm \in BOOL$	Nm est un booléen.
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Fr' = \emptyset \wedge$	Aucun message n'a été reçu.
$Cr' = \emptyset \wedge$	Aucun message de contrôle n'a été reçu.
$Nm' = false \wedge$	Aucun message n'est en attente de lecture.
I'	L'invariant est établit.
OPERATIONS	
$m, c \leftarrow$ Get $\hat{=}$	
$I \wedge Nm = true \Rightarrow$	Il y a des messages en attente.
$\exists_{am,ac} \cdot (am \in MES \wedge$	A la fin :
$ac \in BOOL \wedge$	
$m' = am \wedge c' = ac \wedge$	Le message reçu est envoyé au programme.
$Fr' = Fr \leftarrow am \wedge$	Le message reçu est enregistré dans Fr.
$Cr' = Cr \leftarrow ac \wedge$	Le message de contrôle est enregistré dans Cr.
$Nm' = false \wedge I')$	Un message a été lu.
$r \leftarrow$ More $\hat{=}$	
$I \Rightarrow$	
$\exists_{ar} \cdot (ar \in BOOL \wedge$	
$r' = ar \wedge$	La réponse est envoyée au programme.
$Nm' = ar \wedge I')$	La réponse est enregistrée dans Nm.

[10] La spécification en B de ce module est très semblable à celle du module $EnvS$. Nous ne la donnons pas ici.

Les variables Fr et Cr du module *Receiver* doivent être implantées avec les variables Fr et Cr du module *EnvR*. Ceci est illustré dans la figure 3.1, page 73. La variable Fb du module *Receiver* doit être implantée avec la variable Fb d'un autre module qui permet d'accéder au fichier de messages copié. Nous ne donnons pas ici cette spécification.

Nous avons utilisé la technique présentée dans le chapitre 2 pour spécifier séparément l'émetteur et le récepteur. Le module *EnvS* correspond à l'interface fournie au module *Sender* pour utiliser le canal de communication. Ses variables auxiliaires permettent d'exprimer la propriété de bon comportement souhaitée du côté émetteur, ce qui est fait dans la post-condition de l'opération $Transfert$. Ces deux modules forment la spécification du côté émetteur. Le module *EnvR* correspond à l'interface fournie au module *Receiver* pour utiliser le canal de communication à la réception. Ses variables auxiliaires permettent d'exprimer la propriété de bon comportement souhaitée du côté récepteur, ce qui est fait dans la post-condition de l'opération *Reception*. Ces deux modules forment la spécification du côté récepteur.

3.1.3 Spécification du Canal de Communication

Avec les quatre spécifications de module données, il n'est pas possible d'exprimer des propriétés sur le canal car pour cela il est nécessaire d'utiliser des informations du côté émetteur et du côté récepteur simultanément, voir figure 3.1, page 73. Il est néanmoins important de pouvoir exprimer ces propriétés parce qu'elles correspondent à des *hypothèses* que nous souhaitons faire sur le canal de communication. Pour résoudre ce problème, nous allons introduire la notion de module partagé.

Les modules *EnvS* et *EnvR* sont définis sur des domaines différents : *ils agissent sur des variables différentes qui dénotent des ensembles disjoints.* Ils peuvent donc être vus comme un seul module qui contient leurs variables et leurs opérations, Ce qui permet de supposer que les opérations sont exécutées l'une après l'autre puisqu'elles appartiennent à un même module. A premier abord, il est difficile d'accepter ce raisonnement car notre intuition nous fait penser qu'au contraire ces opérations peuvent être exécutées d'une façon simultanée : dans la réalité rien ne s'y oppose. Néanmoins, si nous tenons compte du fait que les variables modifiées par l'opération *Send* sont différentes de celles modifiées par les opérations *Get* et $More$, que *Sender* utilise uniquement l'opération *Send*, et que *Receiver* utilise uniquement les opérations *Get* et $More$, il est possible de s'intéresser seulement à la fin de l'exécution de chaque opération sans tenir compte de ce qui se passe entre le début et la fin. Ainsi, les opérations sont observées l'une après l'autre. Dans le cas extrême où deux opérations terminent exactement au même moment, nous pouvons encore considérer qu'elles ont été exécutées l'une après l'autre car l'ordre choisi n'affecte pas l'état final du système. Le module ainsi obtenu permet de spécifier le canal de communication dans son ensemble en ajoutant les autres propriétés qui expriment les hypothèses que nous souhaitons faire sur lui.

Nous allons donc *composer* les modules *EnvS* et *EnvR* en un seul module.

Pour cela nous allons définir la notion de composition de modules.

3.2 Composition de Modules

La composition de deux modules $M1$ et $M2$ définis sur deux espaces d'états différents est un module qui est défini sur le produit cartésien de ces deux espaces et qui satisfait la condition qui suit :

Condition 3.1. Composition de Modules.

1. La spécification d'initialisation est celle qui résulte de la composition parallèle des spécifications élémentaires d'initialisation de $M1$ et de $M2$:

$$O1_0 \,||\, O2_0$$

2. Les opérations qu'il offre sont celles de $M1$ et de $M2$. Les pré-conditions restent inchangées. Les post-conditions n'affectent pas les variables apportées par l'autre module.

Définition 3.1. Composition de Modules.
Soit $M1 \triangleq (x, x : O1_0, x : O1_1 \ldots x : O1_n)$ **et** $M2 \triangleq (w, w : O2_0, w : O2_1 \ldots w : O2_m)$ **les spécifications de deux modules définis sur deux espaces d'états différents. Le composition de** $M1$ **avec** $M2$ **est un nouveau module dont l'espace d'états est le produit cartésien des espaces de** $M1$ **et** $M2$ **et qui est défini comme suit :**

$$\begin{array}{lll} M1 \,||_M\, M2 & \triangleq & ((x, w), \\ & & (x : O1_0 \,||\, w : O2_0), \\ & & (x : O1_1 \ldots x : O1_n, w : O2_1 \ldots w : O2_m)) \end{array}$$

Le résultat de la composition est une spécification de module qui contient les variables x, w, qui est initialisé avec la composition parallèle des spécifications d'initialisation, et qui offre les opérations des deux modules composés.

Le module *Channel* de l'exemple peut être défini comme la composition de *EnvS* avec *EnvR* :

$$Channel = EnvS \,||_M\, EnvR$$

Ce nouveau module est *partagé* par les processus *Sender* et *Receiver*. Ceci veut dire que pour implanter le module *Sender* nous allons utiliser $Channel$ a la place de $EnvS$, et que pour implanter le module *Receiver* nous allons utiliser le même module $Channel$ à la place de $EnvR$. Ceci est illustré dans la figure 3.2.

Nous allons maintenant compléter le module $Channel$ avec d'autres hypothèses que nous souhaitons faire sur le canal. Nous supposons que le canal est totalement fiable et que les messages arrivent dans l'ordre d'envoi. Il n'y a donc pas de perte de message ni d'altération d'information dans le canal.

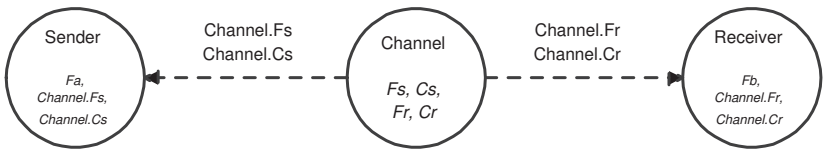

Figure 3.2 – Utilisation de Modules Partagés

Pour simplifier la modélisation de ces hypothèses, nous allons introduire deux nouvelles variables auxiliaires dans la spécification du module *Channel* : Ns et Nr.

- Ns contient le nombre de messages envoyés. Son type est NAT. Nous avons donc $Ns = size(Fs)$.
- Nr contient le nombre de messages reçus. Son type est NAT. Nous avons donc $Nr = size(Fr)$ [11] .

L'opération *Send* doit incrémenter la variable Ns puisqu'elle envoie un nouveau message sur le canal. Sa post-condition doit donc contenir la formule :

$$Ns' = Ns + 1$$

L'opération *More* doit être modifiée pour tenir compte du fait que les messages qui arrivent ont bien été envoyés, c'est à dire qu'il n'y a pas de faux messages. Cette hypothèse est exprimée dans la post-condition comme suit :

$$ar = true \ \Rightarrow \ Ns > Nr$$

Cette formule exprime le fait que si un nouveau message est en attente de lecture : $ar = true$, alors il a été envoyé : $Ns > Nr$.

L'opération *Get* doit être modifiée pour tenir compte du fait que les messages arrivent dans l'ordre et qu'un nouveau message a été lu. Ceci peut être exprimé avec la formule :

$$am = Fs(Nr + 1) \wedge ac = Cs(Nr + 1) \wedge Nr' = Nr + 1$$

Les deux dernières formules mettent en relation des informations du côté émetteur et du côté récepteur. L'apport du module *channel* est clairement illustré ici. Ces formules modifient la spécification donnée au départ pour les opérations *More* et *Get* car elles font dépendre les variables ar,am et ac des variables auxiliaires. Ceci viole la partie 3 de la condition 2.2 qui dit que les variables concrètes ne peuvent pas dépendre des variables auxiliaires. *Nous acceptons ceci car ce sont des hypothèses que nous faisons sur l'environnement*, ce qui signifie que nous supposons que les implantations qui seront fournies pour les modules *EnvS* et *EnvR* satisfont ces propriétés [12] .

[11] Ns et Nr ne sont pas nécessaires à l'expression des nouvelles propriétés. Nous les incluons pour simplifier leur expression.

[12] Une autre possibilité est de considérer que ces hypothèses sont des propriétés du système qui doivent être implantées. Dans ce cas, les variables Fs, Cs, Ns, Fr, Cr et Nr doivent être des variables abstraites plutôt qu'auxiliaires. Il serait donc nécessaire de les implanter.

Dans l'invariant du module *Channel* nous incluons d'autres propriétés supplémentaires qui portent sur le comportement du canal. Elles seront utiles pour effectuer les preuves générées lors de l'implantation des modules *Sender* et *Receiver*.

- La suite de messages reçus est le début de la suite de messages envoyés sur le canal :

  $$Fr = Fs \uparrow Nr$$

  Ceci a pour conséquence :

  $$Nr = Ns \;\Rightarrow\; Fr = Fs$$

  et

  $$Nr \leq Ns$$

- La suite de messages de contrôle reçus est le début de la suite de messages de contrôle envoyés sur le canal :

  $$Cr = Cs \uparrow Nr$$

  Ceci a pour conséquence :

  $$Nr = Ns \;\Rightarrow\; Cr = Cs$$

- Seulement le dernier message de contrôle peut contenir *true* :

  $$Ns > 0 \;\Rightarrow\; Cs \uparrow (Ns - 1) \in seq(\{false\})$$

  Une conséquence de ceci est que lorsque le dernier message de contrôle reçu contient *true* alors $Nr = Ns$. Formellement :

  $$Cr(Nr) = true \;\Rightarrow\; Ns = Nr$$

  Ce qui permet de conclure que lorsque le message de contrôle *true* est reçu par le récepteur la suite de messages reçus est identique à la suite de messages envoyés. Formellement :

  $$Cr(Nr) = true \;\Rightarrow\; Fr = Fs$$

Cette dernière propriété est fondamentale dans notre système car elle garantit que le canal n'a pas de défaut. Elle permet donc de montrer que le transfert est effectué correctement.

Dans la réalité, un tel canal n'existe pas. Nous devrons donc admettre que les messages peuvent être altérés. La prise en compte de ce type de comportement est à l'origine de la complexité des protocoles de communication industriels. Dans le chapitre 5, page 97, nous présenterons une méthode descendante de construction de spécifications pour des systèmes plus complexes. Elle permet de prendre en compte ce type de problématique.

Nous allons inclure toutes ces propriétés dans l'invariant du module *Channel*. Nous obtenons la spécification qui suit [13] :

MODULE	
Channel	Interface de communication fournie à l'émetteur et au récepteur.
VARIABLES AUXILIAIRES	
Fs,	Messages envoyés.
Cs,	Messages de contrôle envoyés.
Ns,	Numéro de Messages envoyés.
Fr,	Messages reçus.
Cr,	Messages de contrôle reçus.
Nr,	Numéro de Messages reçus.
Nm	Indique si des messages sont en attente.
INVARIANT	
$Fs \in seq(MES) \wedge$	Fs est une suite de messages.
$Cs \in seq(BOOL) \wedge$	Cs est une suite de booléens.
$\mathbf{Ns} \in \mathbf{NAT} \wedge \mathbf{Ns} = \mathbf{size}(\mathbf{Fs}) \wedge$	Ns est le nombre de messages envoyés.
$Fr \in seq(MES) \wedge$	Fr est une suite de messages.
$Cr \in seq(BOOL) \wedge$	Cr est une suite de booléens.
$\mathbf{Nr} \in \mathbf{NAT} \wedge \mathbf{Nr} = \mathbf{size}(\mathbf{Fr}) \wedge$	Nr est le nombre de messages reçus.
$Nm \in BOOL \wedge$	Nm est un booléen.
$\mathbf{Nr} \leq \mathbf{Ns} \wedge$	Le nombre de messages reçus est $\leq$ au nombre de messages envoyés.
$\mathbf{Fr} = \mathbf{Fs} \uparrow \mathbf{Nr} \wedge$	Les messages sont reçus dans l'ordre.
$\mathbf{Cr} = \mathbf{Cs} \uparrow \mathbf{Nr} \wedge$	Les messages sont reçus dans l'ordre.
$(\mathbf{Nr} = \mathbf{Ns} \Rightarrow \mathbf{Fr} = \mathbf{Fs} \wedge$	
$(\mathbf{Nr} = \mathbf{Ns} \Rightarrow \mathbf{Cr} = \mathbf{Cs}) \wedge$	
$(\mathbf{Ns} > \mathbf{0} \Rightarrow$	Seulement le dernier message de contrôle
$\mathbf{Cs} \uparrow \mathbf{Ns} - \mathbf{1} \in \mathbf{seq}(\{\mathbf{false}\})) \wedge$	peut contenir $true$.
$(\mathbf{Cr}(\mathbf{Nr}) = \mathbf{true} \Rightarrow \mathbf{Ns} = \mathbf{Nr}) \wedge$	**Propriété fondamentale 1.**
$(\mathbf{Cr}(\mathbf{Nr}) = \mathbf{true} \Rightarrow \mathbf{Fr} = \mathbf{Fs})$	**Propriété fondamentale 2.**
INITIALIZATION	
$true \Rightarrow$	Initialement :
$Fs' = \emptyset \wedge$	Aucun message n'a été envoyé.
$Cs' = \emptyset \wedge$	
$\mathbf{Ns}' = \mathbf{0} \wedge$	
$Fr' = \emptyset \wedge$	Aucun message n'a été reçu.
$Cr' = \emptyset \wedge$	
$\mathbf{Nr}' = \mathbf{0} \wedge$	
$Nm' = false \wedge I'$	Aucun message n'est en attente de lecture.
OPERATIONS	
Send$(m, c) \triangleq$	
$Fs, Cs, Ns : ($	A la fin :
$m \in MES \wedge c \in BOOL \Rightarrow$	
$Fs' = Fs \leftarrow m \wedge$	Le message est envoyé.
$Cs' = Cs \leftarrow c \wedge$	Le message de contrôle est envoyé.
$\mathbf{Ns}' = \mathbf{Ns} + \mathbf{1} \wedge I')$	Un message est comptabilisé.
$r \leftarrow$ More $\triangleq$	
$r, Nm : ($	A la fin :
$I \Rightarrow$	
$\exists_{ar} \cdot (ar \in BOOL \wedge$	
$(\mathbf{ar} = \mathbf{true} \Rightarrow \mathbf{Ns} > \mathbf{Nr}) \wedge$	**Hypothèse 1**.
$r' = ar \wedge$	La réponse est envoyée au programme.
$Nm' = ar \wedge I'))$	La réponse est enregistrée dans Nm.

[13] La spécification correspondante en B de ce module est donnée dans le chapitre 4, page 94.

$m, c \leftarrow \text{Get} \mathrel{\hat{=}}$	
$m, c, Fr, Cr, Nr, Nm : ($	A la fin :
$I \wedge Nm = true \Rightarrow$	Il y a des messages en attente.
$\exists_{am,ac} \cdot (am \in MES \wedge$	
$ac \in BOOL \wedge$	
$\mathbf{am = Fs(Nr + 1)} \wedge$	**Hypothèse 2**.
	Le message est reçu dans l'ordre.
$\mathbf{ac = Cs(Nr + 1)} \wedge$	**Hypothèse 3**.
	La condition est reçue dans l'ordre.
$m' = am \wedge c' = ac \wedge$	Le message reçu est envoyé au programme.
$Fr' = Fr \leftarrow am \wedge$	Le message reçu est enregistré.
$Cr' = Cr \leftarrow ac \wedge$	Le message de contrôle est enregistré.
$\mathbf{Nr' = Nr + 1} \wedge$	Un message reçu est comptabilisé.
$Nm' = false \wedge I'))$	Un message a été lu.

3.3 Utilisation de Modules Partagés

Il existe une limitation dans l'utilisation qui peut être faite des variables d'un module partagé pour spécifier les composants d'un système. La difficulté vient du fait que les variables du module partagé peuvent être modifiées par plusieurs composants. Il devient donc impossible de garantir que l'invariant d'un composant est préservé et que les implantations fournies sont correctes. Par exemple, la spécification du module *Receiver* utilise les variables Fr et Cr qui doivent être implantées avec celles du module *Channel*. Nous savons cependant que les variables Fr et Cr sont modifiées uniquement par l'opération *Get* qui est utilisée uniquement par le module *Receiver*. Ainsi, du point de vue du module *Receiver* les variables Fr et Cr sont 'stables'.

Cette analyse nous permet de formuler une condition suffisante pour garantir que les propriétés exprimées dans les spécifications des composants sont préservées quand ces spécifications sont implantées avec des modules partagés.

Condition 3.2. Utilisation des variables d'un module partagé dans la spécification des composants d'un système : *les variables d'un module partagé qui peuvent être utilisées dans la spécification d'un composant sont celles qui sont modifiées uniquement par les opérations du module partagé qui sont appelées exclusivement par ce composant. Une variable qui peut être modifiée par un autre composant ne peut pas être utilisée.*

Nous dirons qu'une variable est *stable* vis-à-vis de la spécification d'un module si elle satisfait la condition 3.2.

Nous devons donc indiquer pour chaque composant les opérations du module partagé qu'il peut utiliser dans son implantation. Avec cette information, il est facile de déterminer les variables qui peuvent être utilisées dans la spécification d'autres composants.

Dans l'exemple, les variables Fs, Cs et Ns peuvent être utilisées dans la spécification du module *Sender* puisqu'elles sont modifiées uniquement par l'opération *Send* qui est utilisée uniquement dans l'implantation du module *Sender*. Les variables Fr, Cr, Nr et Nm peuvent être utilisées dans la spécification du module *Receiver* puisqu'elles sont modifiées exclusivement par les opérations *Get* et *More* qui sont appelées uniquement dans l'implantation du module

Receiver.

L'introduction de modules partagés donne lieu à la question suivante :

Est-il toujours possible d'utiliser la relation de raffinement entre spécifications de module pour implanter un module qui partage un autre module ?

La réponse est oui car si nous respectons la condition 3.2 les variables du module partagé sont stables vis-à-vis des modules qui les utilisent. Toutes les notions théoriques restent donc valides.

3.4 Implantation de Modules Partagés

Le processus d'implantation de la spécification d'un module partagé est différent selon le type de système qu'elle décrit : centralisé ou distribué.

Dans le cas d'un système centralisé, le module doit être implanté en adoptant la même démarche que pour un module non partagé. Les variables auxiliaires peuvent donc être laissées de côté. Des mécanismes de protection pour interdire l'exécution simultanée d'opérations doivent être disponibles. Une telle implantation produit un *monitor* qui peut être partagé par plusieurs processus concurrents.

Dans le cas d'un système distribué, l'implantation est formée par les implantations des parties correspondant à chaque site. Nous devons procéder comme suit :

- Toutes les variables auxiliaires doivent être éliminées ainsi que toutes les hypothèses faites sur le système de communication. De cette façon, nous obtenons un module qui contient uniquement des variables abstraites ou locales. Dans notre exemple, nous éliminons toutes les variables auxiliaires du module $Channel$.
- Le module doit être ensuite décomposé en plusieurs modules, un pour chaque site, de telle sorte que chacun d'entre eux contiennent les opérations qui sont réservées à l'implantation du composant du site correspondant. Dans notre exemple, nous décomposons le module $Channel$ en deux. Nous obtenons les modules $EnvS$ et $EnvR$ dépourvus de leurs variables auxiliaires. La composition des modules obtenus doit être égale au module du départ.
- Les modules ainsi obtenus sont implantés séparément.

3.5 Propriétés de Vivacité

Dans l'exemple, nous souhaitons dire qu'une fois que le transfert a commencé il est effectué complètement. Cette propriété détermine la vivacité de ce système. Elle ne peut pas être exprimée à l'intérieur du module $Channel$ car avec ce seul module il n'y a aucun moyen de prévoir le comportement que les modules $Sender$ et $Receiver$ adopteront.

Cette propriété est exprimée en trois parties. La première est exprimée dans l'opération $Transfert$ du module $Sender$: $Fs' = Fa$. Ceci dit qu'une fois que le

transfert a commencé alors tout le fichier sera émis. La deuxième est exprimée dans l'invariant du module *Receiver* : $Fb = Fr$. Ceci exprime le fait que la copie produite est identique au fichier reçu. Une troisième partie est exprimée dans l'invariant du module *Channel* : $(Cr(Nr) = true \Rightarrow Fr = Fs)$. Cette formule dit que si un message de contrôle contenant *true* est lu du côté récepteur alors le fichier reçu est égal au fichier envoyé. Ces trois propriétés permettent de conclure que lorsque les deux processus terminent alors $Fb = Fa$.

3.6 Terminaison et Equité

Un autre aspect important d'un système concerne les problèmes d'équité et de terminaison. Dans notre contexte, ceci peut être formulé avec la question qui suit :

Est-il possible de montrer la terminaison d'un système formé par plusieurs composants qui utilisent un ou plusieurs modules partagés ?

Dans notre exemple, il n'y a pas de difficulté particulière pour montrer la terminaison de l'opération $Transfert$, puisque la variable Fa est une suite finie de messages. Il sera donc possible d'implanter cette opération avec une boucle dont le corps appelle l'opération *Send* du module *Channel* jusqu'à ce que le dernier message soit envoyé. La variant de la boucle est $(size(Fa) - Ns)$.

Par contre, il n'est pas possible de montrer la terminaison de l'opération *Reception* car pour cela nous avons besoin de connaître la taille du fichier à recevoir ce qui est totalement inconnu du côté récepteur. Même en utilisant le module *Channel* tel que nous l'avons spécifié cette information n'est pas disponible. Plus que cela, le récepteur n'a aucun moyen de connaître le comportement de l'émetteur ; il lui est donc impossible de savoir quand est-ce qu'il enverra le prochain message. Le récepteur doit juste attendre que des nouveaux messages arrivent.

Pour montrer la terminaison de l'opération *Reception* nous devons faire deux hypothèses supplémentaires :

- La première porte sur le temps qui s'écoule entre la réception d'un message et l'arrivée du suivant. Tant que le message de contrôle indiquant la fin du transfert n'a pas été envoyé, l'émetteur enverra un autre message au bout d'un temps fini. Cette hypothèse peut être exprimée à l'aide d'une nouvelle variable auxiliaire, De, qui sera décrémentée à chaque fois que l'opération $More$ est appelée. Elle dénote le temps maximum qui peut s'écouler avant la réception du prochain message. La variable est modifiée aussi par l'opération *Get* à chaque fois qu'un nouveau message est reçu, avec une valeur supérieure au temps nécessaire pour recevoir le prochain message. Cette valeur peut être aussi grande que l'on veut, mais finie. Dans la preuve de terminaison de l'opération *Reception* nous avons besoin seulement de savoir que ce temps est fini. A l'aide de cette variable l'opération *Reception* peut être implantée avec une boucle dont le corps appelle l'opération $More$ d'une façon répétée jusqu'à ce qu'un nouveau message soit en attente de lecture. Le variant de cette boucle est justement la variable auxiliaire De

qui est décrémenté par l'opération *More*.
L'hypothèse que nous avons ajoutée est une hypothèse d'équité [Fra 86, Sch 99] qui exprime le fait que l'opération *Send* sera utilisée au bout d'un temps fini si sa pré-condition est satisfaite, ce qui est le cas tant que le fichier d'origine n'a pas été complètement envoyé. Nous traiterons systématiquement le problème de l'équité de la façon indiquée dans les paragraphes précédents [14] .

– La deuxième porte sur la taille du fichier à recevoir, ce qui est totalement inconnu du côté récepteur. Le module *Channel* ne dispose pas non plus de cette information. Nous supposons donc que le nombre de messages à recevoir est fini. Cette hypothèse peut aussi être exprimée à l'aide d'une nouvelle variable auxiliaire, *Size*, qui sera décrémentée à chaque fois que l'opération *Get* est appelée. Elle dénote la taille du fichier à recevoir. Cette valeur peut être aussi grande que l'on veut, mais finie.

Cette deuxième hypothèse peut être évitée si nous modélisons le fichier à envoyer à l'intérieur du module *Channel* car dans ce cas nous disposons de l'information dont nous avons besoin pour effectuer la preuve de terminaison.

3.7 Conclusion

Les modules partagés peuvent être utilisés pour spécifier des systèmes formés par plusieurs composants. Ils fournissent un moyen de communication entre composants. Ils permettent donc de prendre en compte les aspects concurrents d'un système.

Une restriction est nécessaire dans l'utilisation des variables d'un module partagé lors de la spécification des composants du système car dans le cas général les invariants ne sont pas préservés et les implantations d'opérations ne sont plus correctes. La vérification de la condition 3.2 suffit pour garantir la préservation d'invariants et la correction des implantations.

Deux modules indépendants peuvent être composés pour former un seul module partagé qui pourra être enrichi avec des hypothèses faites sur le système de communication. Le module obtenu peut être utilisé pour montrer la correction des implantations des composants du système. Ceci est une démarche ascendante car nous partons des spécifications déjà existantes, puis nous construisons un module qui permet d'exprimer les propriétés souhaitées du système dans son ensemble.

Dans le chapitre 5, page 97, nous allons étudier une méthode descendante qui permet de construire la spécification d'un système par petits pas. Celle-ci est une démarche inverse car nous allons d'abord dériver la spécification d'un module partagé qui servira dans l'implantation des composants du système, puis nous allons spécifier chaque composant séparément.

L'implantation d'un module partagé est différente suivant le type de système spécifié : centralisé ou distribué. Dans le premier cas le processus d'implantation est semblable à celui d'un module non partagé. Le résultat correspond à

[14] Les notions d'équité faible et d'équité forte sont traitées de la même façon [Fra 86].

un monitor qui peut être utilisé pas plusieurs processus du système. Dans le second cas il est nécessaire d'éliminer toutes les variables auxiliaires et toutes les hypothèses faites sur le dispositif de communication, puis il faut décomposer le module partagé en plusieurs modules qui seront implantés séparément. Nous obtenons ainsi les modules qui serviront à implanter les composants de chaque site.

La technique présentée dans ce chapitre fournie une réponse claire au besoin de structuration d'une spécification d'un système complexe, car les composants du système sont spécifiés et implantés séparément. Le système de communication est aussi spécifié et implanté séparément sous la forme d'un module partagé. En plus, les implantations sont effectuées à l'aide des techniques connues de dérivation de programmes par raffinements successifs. La difficulté qui apparaît est de pouvoir maîtriser la construction de la spécification du système de communication. Ce problème est traité dans le chapitre 5.

Chapitre 4

Utilisation de la Méthode B

Dans ce chapitre nous allons établir une correspondance entre les idées théoriques telles que nous les avons présentées et la méthode B.

4.1 Méthodes Formelles

L'utilisation industrielle de méthodes mathématiques dans la construction de systèmes dépend fortement de la maturité des outils associés : générateur d'obligations de preuve, prouveurs automatiques et interactifs, gestion de projet, traducteurs vers des langages de programmation, etc. Au début des années 80 ont commencé à apparaître des méthodes qui proposent un langage de spécification, un système de preuve, et en ensemble d'outils associés : VDM [Jon 86], Z [Spivey 92], B [Abr 96-1].

La méthode B permet de produire le code d'un programme à partir d'une spécification abstraite sans interrompre à aucun moment le processus de développement formel. Le code est produit en langage C ou ADA et doit être compilé pour obtenir le code exécutable final.

La méthode B a été utilisée avec succès dans la construction de systèmes industriels [Beh,Ben,Fai,Mey 99]. En particulier, pour développer des programmes de contrôle sur lesquels repose la sécurité des usagers.

4.2 Spécifications Elémentaires en B

Une spécification élémentaire en B prend la forme d'une substitution. Plutôt que d'utiliser des prédicats avant-après, B propose la notion de substitution généralisée, qui est une extension de la notion de substitution de variables en logique. La méthode associe à chaque instruction du langage une substitution et un axiome qui permet de transformer toute formule écrite dans le langage de substitutions en une formule de la logique usuelle. Par exemple, la spécification d'une opération qui incrémente de 1 une variable interne x est écrite : $x := x+1$. Ce qui correspond à la spécification avant-après $true \Rightarrow x' = x + 1$.

Les axiomes des principales substitutions généralisées sont définis comme suit :

Soit $P(x)$ un prédicat.

soit E une expression d'un terme.

Soient $S(x), S_1(x), S_2(x)$ des substitutions généralisées qui affectent la variable x.

Soit $R(x)$ une post-condition à établir.

Instruction	Substitution	Axiome	
Sans effet :	$skip$	$[skip]R$	$\Leftrightarrow\ R$
Affectation	$x := E$	$[x := E]R$	$\Leftrightarrow\ R[x \leftarrow E]$
Pré-condtion	$P \mid S$	$[P \mid S]R$	$\Leftrightarrow\ P \wedge [S]R$
Séquence	$S_1; S_2$	$[S_1; S_2]R$	$\Leftrightarrow\ [S_1][S_2]R$
Garde	$P \Longrightarrow S$	$[P \Longrightarrow S]R$	$\Leftrightarrow\ P \Rightarrow [S]R$
Choix	$S_1 \,[]\, S_2$	$[S_1 \,[]\, S_2]R$	$\Leftrightarrow\ [S_1]R \wedge [S_2]R$
Choix non borné	$@z \cdot S$	$[@z \cdot S]R$	$\Leftrightarrow\ \forall_z \cdot [S]R$

Pour plus de détail sur la séquence voir [Abr 96-1], page 375. Pour la boucle et la composition parallèle voir [Abr 96-1], 377 et 308 respectivement.

4.2.1 Forme Normale des Substitutions Généralisées

Il existe une forme normale pour chaque substitution généralisée. Soit $P(x)$ et $Q(x, x')$ deux prédicats. La forme normale d'une substitution $S(x)$ a la structure qui suit [1] :

$$S(x) \ = \ P(x) | @_{x'} \cdot (Q(x, x') \ \Longrightarrow \ x := x') \tag{4.1}$$

La méthode associe trois prédicats distincts à chaque substitution généralisée S qui la caractérisent : $trm(S), fis(S), prd_x(S)$ [2] .

$trm(S)$ définit la condition de terminaison de S. Pour la forme normale, nous avons [3] :

$$trm(P(x) | @_{x'} \cdot (Q(x, x') \ \Longrightarrow \ x := x')) \ \Leftrightarrow \ P(x) \tag{4.2}$$

Ce qui est exactement le rôle de la pré-condition d'une spécification élémentaire, voir la formule 1.1, page 19.

$prd_x(S)$ est le prédicat avant-après associé à la substitution généralisée S. Pour la forme normale, nous avons :

$$prd_x(P(x) | @_{x'} \cdot (Q(x, x') \ \Longrightarrow \ x := x')) \ \Leftrightarrow \ P(x) \ \Rightarrow \ Q(x, x') \tag{4.3}$$

[1] Voir [Abr 96-1], Théorème 6.1.1, pages 283 à 284.

[2] Voir [Abr 96-1], pages 288 à 294.

[3] Voir [Abr 96-1], propriété 6.3.2, page 290.

Ici $Q(x, x')$ correspond à la post-condition d'une spécification élémentaire, voir formule 1.1, page 19.

$fis(S)$ détermine la faisabilité de S. Pour la forme normale, nous avons [4] :

$$fis(P(x)|@_{x'} \cdot (Q(x, x') \implies x := x')) \Leftrightarrow P(x) \Rightarrow \exists_{x'} \cdot Q(x, x') \qquad (4.4)$$

Ceci correspond exactement à la définition de faisabilité que nous avons donnée, voir définition 1.3, page 19.

A chaque substitution généralisée S correspond une spécification élémentaire qui prend la forme :

$$trm(S) \Rightarrow prd_x(S) \qquad (4.5)$$

Cette formule permet de vérifier que la sémantique que nous avons donnée à chaque instruction du langage présenté dans la section 1.5.5, page 22, à l'aide de la fonction Sp, est la même que celle donnée en B.

Il existe un théorème qui permet de mettre une substitution généralisée quelconque S en sa forme normale à l'aide des prédicats $trm(S)$ et $prd_x(S)$ [5] .

$$S = trm(S)|@_{x'} \cdot (prd_x(S) \implies x := x')$$

Nous allons analyser trois substitutions généralisées qui jouent un rôle important en tant qu'outil de spécification.

L'instruction gardée

$SELECT \quad G(x) \quad THEN \quad S(x) \quad END$ est une substitution qui peut s'écrire [6] :

$$G(x) \implies S(x)$$

Son prédicat de terminaison est :

$$trm(G(x) \implies S(x)) \Leftrightarrow (G(x) \Rightarrow trm(S(x))$$

D'après la forme normale de $S(x)$ nous avons : $trm(S(x)) = P(x)$. Nous avons donc :

$$trm(G(x) \implies S(x)) \Leftrightarrow (G(x) \Rightarrow P(x))$$

Son prédicat avant après est :

$$prd_x(G(x) \implies S(x)) \Leftrightarrow (G(x) \wedge prd_x(S(x))$$

D'après la forme normale de $S(x)$ nous avons :

$$prd_x(S(x)) = P(x) \Rightarrow Q(x, x')$$

[4] Voir [Abr 96-1], théorème 6.3.5, page 294.
[5] Voir [Abr 96-1], théorème 6.3.1, page 294.
[6] Voir [Abr 96-1], pages 266 à 270.

Nous avons donc :

$$prd_x(G(x) \implies S(x)) \Leftrightarrow (G(x) \wedge (P(x) \Rightarrow Q(x,x'))$$

En utilisant la formule 4.5, nous obtenons la spécification qui suit :

$$(G(x) \Rightarrow P(x)) \Rightarrow (G(x) \wedge (P(x) \Rightarrow Q(x,x'))$$

Ce qui est équivalent à :

$$(G(x) \Rightarrow P(x)) \Rightarrow (G(x) \wedge Q(x,x')) \tag{4.6}$$

Cette formule correspond exactement à la définition que nous avons donnée de la fonction $Sp(G(x) \implies Q(x,x'))$, voir formule 1.2, page 23.

Un cas particulier de la formule 4.6 que nous utiliserons dans le chapitre 5 pour spécifier des événements est obtenu lorsque $P(x) = true$. Ceci produit la spécification élémentaire :

$$true \Rightarrow (G(x) \wedge Q(x,x')) \tag{4.6a}$$

Le choix non borné

$ANY \quad a \quad WHERE \quad G(x,a) \quad THEN \quad S(x,a) \quad END$ est une substitution qui peut s'écrire [7] :

$$@_a \cdot (G(x,a) \implies S(x,a))$$

Son prédicat de terminaison est :

$$trm(@_a \cdot (G(x,a) \implies S(x,a))) \Leftrightarrow \forall_a \cdot trm(G(x,a) \implies S(x,a))$$

D'après la forme normale de $S(x,a)$ nous avons :

$$trm(G(x,a) \implies S(x,a)) \quad \triangleq \quad G(x,a) \Rightarrow P(x,a)$$

Nous avons donc :

$$trm(@_a \cdot (G(x,a) \implies S(x,a))) \Leftrightarrow \forall_a \cdot (G(x,a) \Rightarrow P(x,a))$$

Son prédicat avant après est :

$$prd_x(@_a \cdot (G(x,a) \implies S(x,a))) \Leftrightarrow \exists_a \cdot (prd_x(G(x,a) \implies S(x,a)))$$

D'après la forme normale de $S(x)$ nous avons :

$$prd_x(G(x,a) \implies S(x,a)) = G(x,a) \wedge (P(x,a) \Rightarrow Q(x,a,x'))$$

Nous avons donc :

$$prd_x(@_a \cdot (G(x,a) \implies S(x,a))) \Leftrightarrow \exists_a \cdot (G(x,a) \wedge (P(x,a) \Rightarrow Q(x,a,x')))$$

[7] Voir [Abr 96-1], pages 266 à 270.

En utilisant la formule 4.5, nous obtenons la spécification qui suit :

$$\forall_a \cdot (G(x,a) \Rightarrow P(x,a)) \Rightarrow \exists_a \cdot (G(x,a) \wedge (P(x,a) \Rightarrow Q(x,a,x')))$$

Ce qui donne la spécification élémentaire :

$$\forall_a \cdot (G(x,a) \Rightarrow P(x,a)) \Rightarrow \exists_a \cdot (G(x,a) \wedge Q(x,a,x')) \tag{4.7}$$

Un cas particulier de cette formule que nous utiliserons dans le chapitre 5 pour spécifier des événements est obtenu lorsque $P(x,a) = true$. Ceci produit la spécification élémentaire :

$$true \Rightarrow \exists_a \cdot (G(x,a) \wedge Q(x,a,x')) \tag{4.7a}$$

L'instruction de spécification

$x : Q(x_0, x)$ est une substitution dont le prédicat de terminaison est :

$$trm(x : Q(x_0,x)) \Leftrightarrow true$$

Son prédicat avant après est :

$$prd_x(x : Q(x_0,x)) \Leftrightarrow Q(x_0,x)[x_0, x \leftarrow x, x']$$

En appliquant la substitution nous obtenons :

$$prd_x(x : Q(x_0,x)) \Leftrightarrow Q(x,x')$$

En utilisant la formule 4.5, nous obtenons la spécification qui suit :

$$true \Rightarrow Q(x,x') \tag{4.8}$$

4.3 Modules

Un module en B prend la forme d'une machine abstraite [8] . Une machine abstraite est formée par un ensemble de spécifications élémentaires, données sous la forme de substitutions généralisées, qui partagent une ou plusieurs variables qui déterminent l'état du système [9] . Ceci correspond exactement à la définition de module donnée dans le chapitre 2, voir section 2.1, page 38.

La spécification d'une machine abstraite contient plusieurs parties :
- $MACHINE$: permet de donner un non au module.
- $SETS$: permet de définir les types utilisées dans la spécification.
- $ABSTRACT\_VARIABLES$: permet de déclarer les variables abstraites.
- $CONCRET\_VARIABLES$: permet de déclarer les variables locales.

[8] Voir [Abr 96-1], chapitre 4, page 227.

[9] Les machines abstraites peuvent avoir des paramètres. Nous n'utilisons pas cette possibilité.

- $ABSTRACT\_CONSTANTS$: permet de déclarer les constantes abstraites.
- $CONCRET\_CONSTANTS$: permet de déclarer les constantes locales.
- $INVARIANT$: permet de donner l'invariant du module.
- $INITIALIZATION$: permet de donner la spécification d'initialisation.
- $OPERATIONS$: permet de donner les spécifications élémentaires du module.

Toutes ces notions ont été introduites dans le chapitre 2.

L'invariant de la machine n'est pas inclus dans les spécifications élémentaires. Il est donc nécessaire de montrer séparément que l'initialisation établit l'invariant et que chaque opération le rétablit.

Le notion de variable auxiliaire n'existe pas en B, voir section 2.9, page 54. En conséquence, la séparation entre variables implantables et non implantables ne peut pas être faite d'une façon formelle.

Dans les chapitres 2 et 3, nous avons donné des exemples de machines abstraites qui correspondent aux spécifications des modules que nous avons écrit. Nous allons donner ici à titre d'exemple la machine qui correspond à la spécification du module partagé $Channel$ donné dans la section 3.2, page 79.

MACHINE
Channel

SETS
MES

VARIABLES
$Fs, Cs, Ns, Fr, Cr, Nr, Nm$

INVARIANT
$Fs \in seq(MES) \wedge Cs \in seq(BOOL) \wedge$
$Ns \in NAT \wedge Ns = size(Fs) \wedge$
$Fr \in seq(MES) \wedge Cr \in seq(BOOL) \wedge$
$Nr \in NAT \wedge Nr = size(Fr) \wedge$
$Nm \in BOOL \wedge Nr \leq Ns \wedge$
$Fr = Fs \uparrow Nr \wedge Cr = Cs \uparrow Nr \wedge$
$(Nr = Ns \Rightarrow Fr = Fs) \wedge (Nr = Ns \Rightarrow Cr = Cs) \wedge$
$(Ns > 0 \Rightarrow Cs \uparrow Ns - 1 \in seq(\{false\})) \wedge$
$(Cr(Nr) = true \Rightarrow Ns = Nr) \wedge$
$(Cr(Nr) = true \Rightarrow Fr = Fs)$

INITIALIZATION
$Fs, Cs, Ns, Fr, Cr, Nr, Nm = \emptyset, \emptyset, 0, \emptyset, \emptyset, 0, False$

OPERATIONS
Send$(m, c) \triangleq$
$PRE \quad m \in MES \wedge c \in BOOL \quad THEN$
$Fs, Cs, Ns := Fs \leftarrow m, Cs \leftarrow c, Ns + 1$
END;

$r \leftarrow$ More $\triangleq$
$ANY \quad ar \quad WHERE \quad ar \in BOOL \wedge (ar = true \Rightarrow Ns > Nr) \quad THEN$
$r, Nm := ar, ar$
END;

$$
\begin{array}{l}
m, c \leftarrow \text{Get} \mathrel{\hat{=}} \\
\quad PRE \quad Nm = true \quad THEN \\
\quad\quad ANY \quad am, ac \\
\quad\quad\quad WHERE \quad am \in MES \wedge ac \in BOOL \wedge am = Fs(Nr+1) \wedge ac = Cs(Nr+1) \\
\quad\quad\quad THEN \quad m, c, Fr, Cr, Nr, Nm := am, ac, Fr \leftarrow am, Cr \leftarrow ac, Nr+1, false \\
\quad\quad END \\
\quad END
\end{array}
$$

4.4 Implantations

La spécification d'un programme en B prend la forme d'une machine abstraite. L'outil qui permet de conduire le processus d'implantation est la relation de raffinement [10] dont la définition correspond à celle que nous avons présenté dans le chapitre 2.

La méthode permet d'utiliser d'autres machines dans l'implantation de la spécification d'un machine. Ceci doit être fait dans un type de machine particulier appelé *IMPLEMENTATION*. La déclaration des machines utilisées est faite dans une partie appelée *IMPORTS* [11] . Il est donc possible d'implanter séparément les machines déclarées dans la partie *IMPORTS*.

Il y a deux différences significatives entre la démarche méthodologique proposée dans B et la notre :

1. Lors du raffinement d'une opération en B, il est possible de remplacer des instructions déjà introduites. Ceci engendre la nécessité de faire intervenir dans chaque raffinement tous les invariants de collage utilisés préalablement. Dans notre démarche, nous pouvons modifier seulement les feuilles de l'arbre d'une opération, voir section 2.5, page 42.
2. Dans une implantation en B, il est nécessaire de déclarer tous les modules utilisés d'un seul coup et d'implanter toutes les opérations du module raffiné en un seul pas de raffinement à l'aide d'opérations fournies par les modules importés. La démarche que nous proposons permet de déclarer graduellement les modules utilisés, c'est à dire en plusieurs pas de raffinement, et permet aussi d'implanter les opérations en plusieurs pas, voir section 2.6, page 43.

La notion de module partagé n'existe pas en B, voir chapitre 3, page 71. La spécification d'un module peut être utilisée dans plusieurs implantations mais ce sont des modules différents à chaque fois. La méthode ne permet donc pas de prendre en compte la notion de module partagé telle que nous l'avons définie.

Dans la section 2.8, page 47, nous avons dit que la spécification d'un système complexe est formée par au moins deux spécifications de module reliées entre elles par des indications sur la façon d'implanter les variables abstraites d'une d'entre elles avec celles de l'autre. Ceci n'est pas pris en compte dans B. Il n'est donc pas possible de donner ces indications pour relier deux machines abstraites.

[10] Voir [Abr 96-1], chapitre 11, page 501.
[11] Voir [Abr 96-1], chapitre 12, page 551.

Dans la suite, nous utiliserons la méthode B en nous rapprochant le plus possible de notre démarche méthodologique. Nous donneront des indications à chaque fois qu'une notion non prise en compte par la méthode sera utilisée.

4.5 Conclusion

Une spécification élémentaire en B prend la forme d'une substitution généralisée. La formule $trm(S) \Rightarrow prd_x(S)$ établit le lien qui existe entre une substitution généralisée et une spécification donnée sous la forme d'un prédicat avant-après. Un module en B prend la forme d'une machine abstraite.

Les notions de variable auxiliaire, d'opération auxiliaire et de module partagé n'existent pas en B. Nous pensons que la méthode B peut être étendue de façon à prendre en compte ces notions de la façon suivante :

- *Variables Auxiliaires* : Une partie dédiée à la déclaration de ce type de variable peut être ajoutée dans la description des machines abstraites.
  Pour implanter une machine qui contient des variables auxiliaires la méthode doit forcer le développeur à fournir une autre machine où toutes ces variables ont été enlevées. Il serait alors possible de générer les obligations de preuve correspondant à la condition 2.2, page 55. Une fois ces obligations déchargées, l'implantation de la nouvelle machine peut être effectuée normalement.
  La méthode peut aussi permettre de compléter avec des variables auxiliaires une machine abstraite qui n'en contient pas afin de l'utiliser dans la solution d'un problème particulier. Dans ce cas, elle doit assurer la validité de la transformation effectuée en générant les obligations de preuve correspondant à la condition 2.2.
- *Opérations Auxiliaires* : Une partie dédiée à la déclaration de ce type d'opération peut être ajoutée dans la description des machines abstraites. Il serait alors possible de générer pour chaque opération auxiliaire une obligation de preuve qui garantit qu'elle no modifie pas les autres variables de la machine abstraite.
  Les opérations auxiliaires ne sont pas implantées. Elles peuvent donc être ignorées lors de l'implantation de la machine abstraite.
- *Modules Partagés* : La description d'une machine abstraite qui sera partagée peut être complétée avec les informations des machines qui la partagent. Pour chacune d'entre elles, il est nécessaire de préciser son *Nom*, les *Noms des Variables* de la machine partagée qu'elle utilise, et les *Noms des Opérations* de la machine partagée qu'elle utilise. Avec ces informations, il est possible de vérifier la condition 3.2, page 83, pour chacune des machines qui partagent la machine décrite.
  L'implantation d'une machine partagée devra être effectuée de la façon indiquée dans la section 3.4, page 84.

Chapitre 5

Spécification de Systèmes

Les mécanismes que nous avons introduit jusqu'à présent suffissent pour construire la spécification d'un système complexe et pour produire une implantation correcte de celui-ci. Les variables auxiliaires permettent la réutilisation de modules déjà existants et les modules partagés permettent de prendre en compte la communication entre composants et donc les aspects concurrents d'un système.

Il y a cependant une difficulté qui apparaît lorsque nous essayons de spécifier un système réel : nous sommes très vite confrontés au problème de la maîtrise de la spécification elle même. Ce problème est généré par le nombre important d'idées qu'elle doit contenir. Nous devons être capables de juger si chaque propriété exprimée est effectivement souhaitée et si d'autres propriétés importantes n'ont pas été oubliées.

Dans [Lop 96-1, Lop 96-2] nous proposons une démarche méthodologique qui permet de construire graduellement la spécification d'un composant d'un système. Les propriétés sont d'abord écrites dans une langue naturelle, puis elles sont introduites dans la spécification formelle peu à peu, ce qui permet de les valider une à une et par niveaux d'abstraction. Toutes les propriétés exprimées d'une façon informelle sont prises en compte soit parce qu'elles sont exprimées dans la spécification formelle, soit parce que nous avons constaté l'impossibilité de le faire. Dans ce dernier cas, nous devons apporter une solution au problème rencontré. Cette démarche est bien adaptée à la construction graduelle de la spécification d'un composant, mais elle ne permet pas la spécification d'un système formé par plusieurs composants, ni la prise en compte des interactions avec l'environnement.

Dans ce chapitre, nous allons présenter une démarche méthodologique qui répond à ces deux problèmes. Pour cela nous allons adopter une approche par événements.

5.1 Systèmes Concurrents : Evolution

L'utilisation de langages formels et de systèmes de preuve dans l'analyse et la construction de systèmes concurrents a nécessité dès le début de regarder le comportement d'un tel système comme la succession d'un ensemble d'événements indivisibles ou atomiques [Dij 65].

Très tôt, les systèmes concurrents ont été modélisés par des systèmes état-transition, où la notion de module joue un rôle primordial. Deux classes distinctes de propriétés ont été identifiées : d'invariance et de vivacité, voir section 3.5, page 84. Les propriétés d'invariance sont toujours satisfaites, c'est à dire que tous les états que le système peut atteindre les satisfont ; elles sont aussi appelées de sûreté. Les opérateurs de la logique usuelle suffisent pour les exprimer. les propriétés de vivacité font intervenir plusieurs états du système. Elles expriment des notions où le temps joue un rôle important.

La nécessité d'exprimer des propriétés temporelles a motivé l'utilisation de logiques temporelles. Pnueli a été un des premiers à la proposer et à l'utiliser pour raisonner formellement sur les propriétés de systèmes concurrents [Pnu 77, Pnu 79]. D'autres travaux où la logique temporelle est utilisée ont suivi [Hai 82, Hai,Owi 83, Lamp 80, Owi,Lam 82, Sch,Mel 82].

Deux approches méthodologiques distinctes sont apparues. La première permet de décrire d'une façon abstraite les comportements possibles du système. Ici la notion de comportement est privilégiée et l'expression de la synchronisation entre processus joue un rôle important. La notion d'état est secondaire. Cette approche rend difficile la structuration d'une spécification, car l'implantation d'une opération abstraite avec des opérations plus élémentaires qui peuvent s'entrelacer avec d'autres opérations n'est pas simple à réaliser. Deux exemples de ce type d'approche sont CCS [Mil 80] et CSP [Hoa 85] [1] .

La deuxième approche permet de spécifier un système par les propriétés qu'il doit satisfaire. Il s'agit d'une méthode axiomatique où les notions d'état et de transition jouent un rôle primordial. Cette approche permet la structuration de spécifications et leur raffinement par étapes successives. La difficulté ici est d'arriver à exprimer les propriétés du système d'une façon claire et simple. Plusieurs méthodes axiomatiques de spécification basées sur la notion d'état sont apparues.

Manna et Pnueli ont proposé d'écrire des formules sur les états du système et de les combiner avec des opérateurs logiques temporels pour former des formules temporelles [Man,Pnu 92]. Ils proposent un système de preuve qui permet de vérifier formellement que la spécification du système satisfait les propriétés souhaitées. Lamport a proposé une méthode qui utilise des notions semblables mais qui permet en plus d'exprimer des formules sur les actions du système :TLA, logique temporelle d'actions [Lamp 80, Lamp 83, Lamp 91].

D'autres formalismes basés sur des idées semblables ont été proposés. Certains utilisent des opérateurs temporels d'autres utilisent seulement la logique usuelle [Cha,Mis 88, Bac,Kur 83, Lyn,Tut 87, Lam,Sha 90].

[1] La théorie utilisée dans ce type d'approche est basée sur les algèbres de processus.

Plus récemment, les techniques de raffinement ont commencé à être utilisées pour conduire le processus de construction de la spécification formelle d'un système concurrent [Abr 96-2, Abr,Mus 97, Abr,Mus 98, Bac 89, Bac 90, Bel,Dar,Jul,Kou 2000, But 96, Lop 99, Lop,Sim,Don 2000]. L'apport du raffinement ici est de permettre l'introduction d'événements pas à pas. Il est donc possible de partir d'une spécification abstraite qui contient peu d'événements, puis d'introduire des nouveaux événements à chaque pas de raffinement.

Le problème de l'implantation d'une spécification événementielle est souvent laissé de côté, ce qui casse le processus de raisonnement formel. Nous allons aborder ce problème dans ce chapitre.

5.2 Observation d'un Système

Une technique très utilisée pour décrire le comportement de systèmes concurrents consiste à spécifier les événements qui peuvent être observés lorsque le système est vu de l'extérieur. Cette technique permet d'éliminer complètement les détails d'implantation, et permet aussi de construire une spécification d'une façon très intuitive car ce que nous décrivons sont les observations que le système doit admettre.

La démarche est différente à celle présentée dans les premiers chapitres. Plutôt que de fournir les spécifications des composants qui implantent le système, nous spécifions les événements qu'il permet d'observer. L'état du système est vu dans son ensemble. Les événements le font évoluer.

5.3 Spécification d'Evénements

La spécification d'un événement est formée par deux formules logiques : la garde $G(x)$ et l'action $A(x, x')$. x est une variable, ou une liste de variables, libre dans les deux formules. Elle représente l'état du système observé. *La garde* $G(x)$ dénote les états qui permettent le déclenchement de l'événement. *L'action* $A(x, x')$ dénote les transitions d'états que l'événement permet d'observer.

Nous utiliserons deux classes de formules pour spécifier un événement. La première prend la forme :

$$G(x) \wedge A(x, x') \tag{5.1}$$

Cette formule capture l'idée intuitive que nous avons quand un événement est observé : la garde et l'action sont satisfaites.

La deuxième prend la forme :

$$\exists_a \cdot (G(x, a) \wedge A(x, a, x')) \tag{5.2}$$

Cette formule permet de spécifier le choix non déterministe a que le système peut effectuer lorsque l'événement est observé. La formule capture bien cette idée car elle spécifie que lorsque l'événement est observé, il existe un objet a tel que la garde et l'action sont satisfaites.

La spécification d'un événement peut être donnée aussi sous la forme d'une spécification élémentaire, pré-post, comme suit [2] :

$$true \Rightarrow G(x) \wedge A(x, x') \tag{5.3}$$

et

$$true \Rightarrow \exists_a \cdot (G(x, a) \wedge A(x, a, x')) \tag{5.4}$$

La pré-condition *true* n'ajoute aucune condition supplémentaire pour permettre l'observation de l'événement. L'avantage de cette dernière forme est qu'elle permet d'utiliser toute la théorie du raffinement introduite dans les premiers chapitres dans ce nouveau contexte. Nous allons donc adopter la forme indiquée dans les formules 5.3 et 5.4.

Puisque les spécifications que nous allons donner dans la suite seront écrites directement dans le formalisme de la méthode B, nous allons utiliser les substitutions généralisées correspondantes, voir section 4.2, page 89 :

$$SELECT \quad G(x) \quad THEN \quad S(x) \quad END$$

et

$$ANY \quad a \quad WHERE \quad G(x, a) \quad THEN \quad S(x, a) \quad END$$

5.4 Raffinement d'Evénements

Le raffinement d'événements permet de remplacer un événement par un autre. La condition nécessaire pour pouvoir effectuer ce remplacement est que tout ce qui est observable avec l'événement concret soit aussi observable avec l'abstrait. L'événement concret permet de réduire le non déterminisme interne de l'abstrait. Ceci peut être exprimé formellement comme suit :

Propriété 5.1. *Réduction du non déterminisme interne d'un événement.*
Soit E_1 et E_2 deux spécifications d'événement. Si E_1 est raffiné par E_2 alors tout ce que E_2 permet d'observer peut être observé avec E_1. Formellement :

Pour la première forme, formule 5.1 :

$$E_1 \sqsubseteq E_2 \quad \Rightarrow \quad \forall_{x,x'} \cdot (G_2(x) \wedge A_2(x, x') \Rightarrow G_1(x) \wedge A_1(x, x'))$$

Pour la deuxième forme, formule 5.2 :

$$E_1 \sqsubseteq E_2 \quad \Rightarrow$$
$$\forall_{x,x'} \cdot (\exists_a \cdot (G_2(x, a) \wedge A_2(x, a, x')) \Rightarrow \exists_a \cdot (G_1(x, a) \wedge A_1(x, a, x')))$$

[2] Voir [Abr 96-2, But 96].

Il est facile de vérifier que la relation de raffinement entre spécifications élémentaires satisfait cette propriété, voir définition 1.4, page 22.

Si le raffinement effectué modifie l'espace de définition de variables, nous obtenons la propriété qui suit :

Propriété 5.2. *Raffinement d'événements avec changement d'espace.*
Soit $E_1(x, x')$ et $E_2(w, w')$ deux spécifications d'événements définies sur des espaces d'états différents. Soit $J(x, w)$ une relation totale définie entre ces deux espaces d'états, et dont $J(x, w)^{-1}$ est aussi une relation totale. Si E_1 est raffiné par E_2 sous la transformation J alors tout ce que E_2 permet d'observer peut être observé avec E_1. Formellement :

Pour la première forme :

$$\begin{array}{l} E_1 \sqsubseteq_J E_2 \quad \Rightarrow \\ \quad \forall_{x,w,w'} \cdot (J(x,w) \Rightarrow \quad G_2(w) \wedge A_2(w,w') \Rightarrow \\ \qquad\qquad\qquad\qquad\quad \exists_{x'} \cdot (G_1(x) \wedge A_1(x,x') \wedge J(x',w'))) \end{array}$$

Pour la deuxième forme :

$$\begin{array}{l} E_1 \sqsubseteq_J E_2 \quad \Rightarrow \\ \quad \forall_{x,w,w'} \cdot (J(x,w) \Rightarrow \quad \exists_a \cdot (G_2(w,a) \wedge A_2(w,a,w')) \Rightarrow \\ \qquad\qquad\qquad\qquad\quad \exists_{a,x'} \cdot (G_1(x,a) \wedge A_1(x,a,x') \wedge J(x',w'))) \end{array}$$

Il est facile de vérifier que la relation de raffinement entre spécifications élémentaires avec changement d'espace satisfait cette propriété, voir définition 1.6, page 32.

La relation J^{-1} doit être totale car à toute transition observable avec E_2 doit correspondre au moins une transition observable avec E_1.

5.5 Spécification Evénementielle d'un Système

Nous allons utiliser la notion de module pour donner la spécification événementielle d'un système, voir section 2.1, page 38. Nous adoptons exactement la même démarche que dans [Abr 96-2, But 96]. Les variables du module dénotent l'état du système. Les spécifications d'événements déterminent les observations que le système admet. L'observation d'un événement est atomique. C'est à dire qu'il n'y pas d'observation simultanée. Les événements sont observés l'un après l'autre.

Le module est initialisé avec une spécification de la forme :

$$true \Rightarrow A(x') \tag{5.5}$$

La substitution généralisée correspondante est, voir section 4.2, page 89 :

$$x : A(x)$$

Les spécifications d'événements prennent la forme des formules 5.3 et 5.4. Nous appelleront ce type de spécification $SYSTEME$ car elle correspond à la spécification d'un système dans son ensemble.

5.6 Raffinement de Spécifications Système

Une spécification événementielle $ME1(x)$ est raffinée par une autre $ME2(w)$ sous la transformation de l'espace d'états $J(x, w)$ si, voir définition 2.1, page 40 :

$$ME1(x) \sqsubseteq_{J(x,w)} ME2(w)$$

Ceci signifie que l'initialisation de $ME2$ raffine celle de $ME1$, et que chaque événement i de $ME2$ raffine l'événement correspondant de $ME1$. Les propriétés 5.1 et 5.2, page 100, permettent de conclure que pour toute trace d'événements observée avec $ME2$ il existe une trace identique qui peut être observe avec $ME1$.

Il y a deux cas particuliers de raffinement que nous allons admettre car ils sont très utiles dans la pratique. Le premier permet d'introduire des nouveaux événements dans la spécification. Le deuxième permet de décomposer un événement en plusieurs. Ceci est une démarche purement méthodologique qui ne demande pas de modification de la définition de la relation de raffinement que nous avons donnée. Toutes les notions théoriques restent valides.

5.6.1 Introduction de Nouveaux Evénements

La démarche que nous allons suivre est identique à celle de [Abr 96-2, But 96]. Pour ajouter un événement dans un pas de raffinement, nous devons supposer qu'il raffine un événement qui existe déjà dans la spécification raffinée, mais qui n'a aucun effet sur les variables de celle-ci. La spécification d'un tel événement est :

$$true \Rightarrow skip$$

La substitution généralisée correspondante est $SKIP$.

Lors de la preuve du raffinement, nous devrons donc montrer que la spécification du nouvel événement raffine $SKIP$. C'est à dire que le nouvel événement ne modifie pas les variables de la spécification raffinée. Cette technique est très utile car elle permet d'introduire des nouveaux événements dans une spécification qui peuvent s'entrelacer avec ceux déjà existents. Cette démarche nous oblige à introduire dans le même pas de raffinement tous les événements qui peuvent modifier les variables introduites dans le pas. Le système $FWS2$ donné plus loin illustre cette idée. Il ajoute plusieurs événements au système $FWS1$, voir section 5.9.4, page 111.

Avec l'introduction de nouveaux événements, nous créons le risque de bloquer les événements de la spécification raffinée car les événements introduits peuvent prendre le contrôle indéfiniment. Dans [Abr 96-2, Bac 90, But 96], il est nécessaire de fournir un variant à chaque raffinement qui permet de montrer que ce genre de situations ne se présente pas.

Nous pensons cependant qu'il existe des cas de spécification où nous souhaitons précisément bloquer le système ou une partie du système lorsque une situation non désirée se produit. Nous laissons donc le choix au concepteur de demander ou non cette preuve.

5.6.2 Décomposition d'un Evénement

Nous allons aussi permettre la décomposition d'un événement en plusieurs. Une telle décomposition permet d'observer plus finement l'événement décomposé. Supposons par exemple, que nous observons la ligne météor du métro parisien dans son ensemble. Supposons en plus que nous avons spécifié un événement qui permet d'observer l'ouverture d'une porte quelconque de n'importe quel quai de la ligne. Cet événement est utile pour décrire le système avec un haut niveau d'abstraction, mais il ne permet pas de faire la différence entre l'ouverture d'une porte auxiliaire et l'ouverture d'une porte palière. Nous souhaitons donc pouvoir le décomposer en deux événements différents. Le cas d'étude du chapitre 6 contient des exemples de ce type de décomposition, voir section B.2.7, page 173, événement $Transfert$.

Pour effectuer la décomposition, nous allons supposer que nous avons autant de spécifications identiques de l'événement à décomposer dans la spécification à raffiner que d'événements qui résultent de la décomposition, mais que nous ne les distinguons pas dans la spécification à raffiner. Chaque observation d'un de ces événements est considérée comme une observation de l'événement à décomposer.

Cette analyse nous pouvons formuler la règle pratique qui suit : *toutes les spécifications d'événement qui résultent de la décomposition d'un événement doivent raffiner la spécification de l'événement décomposé.*

5.7 Implantation d'une Spécification Système

Le problème de l'implantation d'une spécification événementielle est très peu abordé dans la littérature. Nous pensons que ce problème est important car il est indispensable de préserver le raisonnement formel tout au long du développement d'un système. Dissocier l'implantation de la spécification serait une source d'erreurs importante, ce qui est contraire à notre démarche de développement formel.

Nous proposons ici de transformer la spécification événementielle finale en la spécification d'un module partagé qui peut être implanté de la façon indiquée dans la section 3.4, page 84. Un avantage significatif de cette approche est que pour effectuer l'implantation souhaitée, il est nécessaire d'utiliser uniquement des techniques connues de développement de programmes par raffinements. Les spécifications événementielles que nous allons construire peuvent ainsi être transformées en spécifications de composants, données sous la forme de spécifications de module. La figure 5.1 illustre cette idée.

L'idée directrice est d'inclure dans la spécification événementielle uniquement les propriétés qui concernent les mécanismes de communication du système avec son environnement et les mécanismes de communication entre composants. La spécification produite doit permettre l'observation de tous les échanges d'information que le système admet. Chaque événement de la spécification finale permet d'observer une opération atomique d'échange d'information.

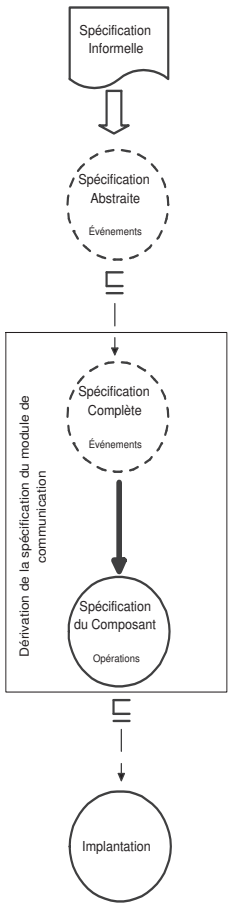

Figure 5.1 – Dérivation de la Spécification du Module de Communication

5.7.1 Dérivation du Module de Communication

Soit Ms la spécification d'un module dont l'initialisation est O_0 et qui contient N opérations $O_1 \dots O_N$. Soit x la variable ou la liste de variables de Ms. x représente toutes les variables du module : abstraites, auxiliaires, locales et constantes. Soit I_m l'invariant de Ms.

Soit Es la spécification événementielle d'un système dont l'initialisation est E_0 et qui contient N événements $E_1 \dots E_N$. Soit x la variable ou la liste de variables de Es. Soit I_s l'invariant de Es.

Nous dirons que le comportement du module Ms peut être observé avec la spécification événementielle Es si la condition qui suit est satisfaite :

Condition 5.1. Observation du comportement d'un module avec une spécification événementielle.

1. Les variables de Ms sont exactement les mêmes que celles de Es.
2. $I_m \quad \Leftrightarrow \quad I_s$. L'invariant de Ms est équivalent à celui de Es.
3. $O_0(x') \;\Rightarrow\; E_0(x')$. L'initialisation de Ms implique l'initialisation de Es. L'implication signifie que l'initialisation du module peut être plus déterministe que celle de la spécification événementielle.
4. Pour chaque opération O_i de Ms l'événement correspondant E_i de Es satisfait la formule qui suit :
   Cas 1 : l'événement E_i prend la forme : $G(x) \wedge A(x, x')$. :

$$\forall_{x,x'} \cdot (P_i(x) \wedge Q_i(x, x') \;\Rightarrow\; G_i(x) \wedge A_i(x, x')) \tag{5.6}$$

   Cas 2 : l'événement E_i prend la forme : $\exists_a \cdot (G(x, a) \wedge A(x, a, x'))$.

$$\forall_{x,a,x'} \cdot (P_i(x, a) \wedge Q_i(x, a, x') \;\Rightarrow\; G_i(x, a) \wedge A_i(x, a, x')) \tag{5.7}$$

   Cette dernière partie de la condition garantit que toute opération i de Ms qui est utilisée sous sa pré-condition peut être observée par l'événement i correspondant de Es. Par conséquent, toute transition admise par Ms est aussi admise par Es.

Pour dériver la spécification du module correspondant à une spécification événementielle, nous allons procéder comme suit :

- Les constantes de la spécification du module sont celles de la spécification événementielle.
- Les variables de la spécification du module sont celles de la spécification événementielle. Dans la spécification du module nous pouvons séparer les variables en locales, abstraites et auxiliaires. Ce choix est fait en fonction de l'implantation que nous souhaitons pour le module dérivé. Pour chaque opération dérivée, les variables auxiliaires doivent respecter la condition 2.2 et donc les formules 2.5 et 2.6.
- L'invariant du module est celui de la spécification événementielle.
- L'initialisation du module est une spécification élémentaire telle que la formule $Q_0(x') \;\Rightarrow\; A_0(x')$ est satisfaite.
- Chaque événement de la spécification événementielle est transformé en la spécification d'une opération de la façon indiquée plus bas dans la section 5.7.2. Chaque opération dérivée doit satisfaire soit la formule 5.6 soit la formule 5.7 suivant la forme de l'événement transformé.

Cette démarche est illustrée dans la section 5.9.5, page 114, où le module $FWS\_ENVIRONMENT$ est le résultat de la transformation de la spécification événementielle $FWS2$.

5.7.2 Transformation d'un Evénement en une Opération

La transformation d'un événement en une opération est effectuée suivant la forme de la spécification de l'événement. Nous avons deux cas :

1. L'événement prend la forme de la formule 5.1 : $G(x) \wedge A(x, x')$. Nous associons une opération dont la spécification prend la forme $P(x) \Rightarrow Q(x, x', z')$. C'est à dire, une opération sans paramètre d'entrée mais qui peut avoir des paramètres de sortie. En plus, la formule qui suit doit être satisfaite :

$$\forall_{x,x',z'} \cdot ((P(x) \Leftrightarrow G(x)) \wedge (Q(x, x', z') \Rightarrow A(x, x'))) \qquad (5.8)$$

   Le fait que la pré-condition soit équivalente à la garde garantit que les propriétés de la spécification événementielle qui concernent uniquement des gardes sont satisfaites aussi par le module dérivé. Par exemple, l'absence d'interblocage. La post-condition implique l'action parce que l'opération peut être plus déterministe que l'événement. Il est facile de vérifier que la formule 5.8 implique la formule 5.6.

   La transformation de l'événement $BeginCycle1$ dans l'exemple de la section 5.9.5, page 114, illustre ce premier cas.

2. L'événement prend la forme de la formule 5.2 : $\exists_a \cdot (G(x, a) \wedge A(x, a, x'))$. Nous associons une opération dont la spécification prend la forme $P(x, a) \Rightarrow Q(x, a, x', z')$. C'est à dire, une opération avec paramètres d'entrée et paramètres de sortie. La formule qui suit doit être satisfaite :

$$\forall_{x,a,x',z'} \cdot ((P(x, a) \Leftrightarrow G(x, a)) \wedge (Q(x, a, x', z') \Rightarrow A(x, a, x'))) \qquad (5.9)$$

   Le paramètre d'entrée a doit être fourni lors de l'appel de l'opération. Il est donc le témoin qui permet de satisfaire la formule 5.2, page 99. Il est facile de vérifier que la formule 5.9 implique la formule 5.7.

   La transformation de l'événement $ExamineWarning2$ dans l'exemple de la section 5.9.5, page 114, illustre ce deuxième cas.

5.8 Spécification des Composants du Système

Une fois que nous avons dérivé le module de communication, nous pouvons spécifier les composants du système de la façon indiquée dans le chapitre 3, page 71. Nous obtenons ainsi des spécifications qui peuvent être implantées à l'aide du module de communication dérivé de la spécification événementielle. La figure 5.2 donne une idée d'ensemble de la méthode de spécification que nous proposons.

5.9 Exemple : Le Système de Contrôle d'Alarmes

Pour illustrer les idées présentées dans ce chapitre, nous allons utiliser un extrait du système spécifié dans [Lop 96-2]. Il s'agit du système de contrôle d'alarmes embarqué dans les Airbus A340 : Flight Warning System (FWS). Ce cas d'étude à été fourni par la Société Aérospatiale. Une spécification informelle complète de ce système est donnée dans [Lop 96-2].

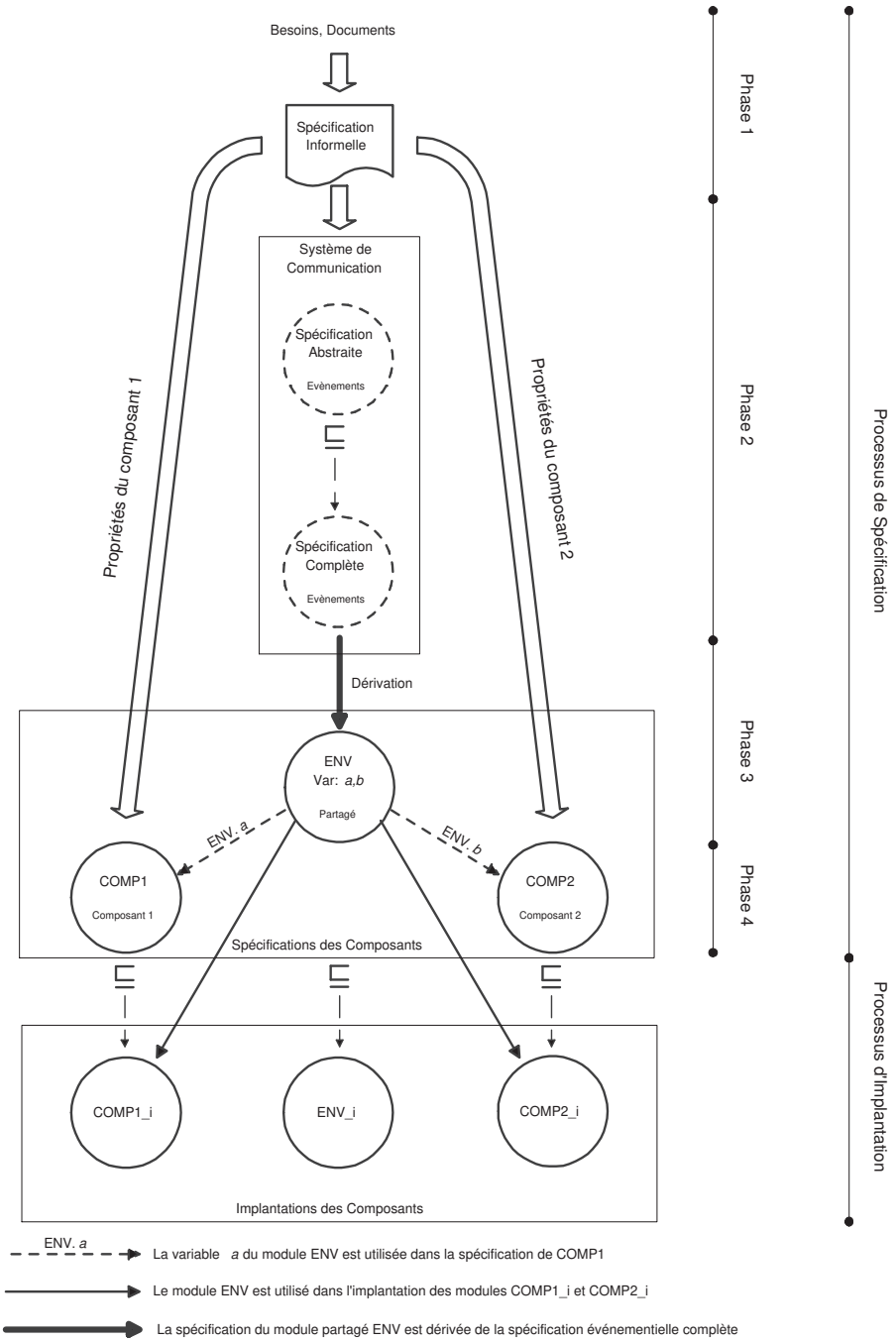

Figure 5.2 – Méthode de Spécification Proposée

La spécification formelle fournie dans [Lop 96-2] ne tient pas compte des aspects concurrents du système ni des interactions avec l'environnement. La méthode que nous présentons ici fournit une réponse aux difficultés que nous avons rencontrées lors de cette première expérience. Ces difficultés sont présentées en détail dans [Lop,Sim,Don 2000].

La fonction principale du FWS est du surveiller en permanence l'état de l'appareil. Lorsqu'une situation anormale se produit, il doit informer l'équipage à l'aide de signaux d'alarme. Une difficulté importante est engendrée par le fait que plusieurs signaux d'alarme peuvent être en compétition pour être émis et que le système dispose d'un nombre limité de ressources pour le faire.

5.9.1 Description du système

Pour décrire le comportement du système il est utile d'introduire quatre notions fondamentales : alarme, signal d'alarme, phase de vol et commande de contrôle.

Alarme

Une alarme est un état particulier de l'appareil qui requière l'attention de l'équipage. Par exemple un réacteur qui est en feu ou la détection de fumée dans les toilettes.

Lorsque la condition associée à une alarme est satisfaite nous dirons qu'elle est présente. Dans le cas contraire nous dirons qu'elle est absente. Avant d'envoyer des signaux d'alarme un temps prédéfini doit s'écouler à partir du moment ou l'alarme devient présente. Après l'écoulement de ce temps nous dirons que l'alarme est confirmée.

Une alarme confirmée peut être soit active soit inhibée suivant la phase de vol où se trouve l'appareil.

Signaux

A chaque alarme sont associés des signaux d'alarme qui prennent la forme de messages textuels, sonores, lumineux et numériques. Ces signaux peuvent être émis uniquement si l'alarme est active.

Un même signal peut être associé à plusieurs alarmes. Par exemple, un même son peut être utilisé pour alerter l'équipage sur plusieurs situations de danger différentes. Nous allons différencier les signaux même si leur caractéristiques externes sont identiques.

Nous dirons qu'un signal est absent si son alarme associée n'est pas active. Dans le cas contraire nous dirons qu'il est présent. Un signal présent peut être dans un des trois états qui suivent : actif, effacé ou annulé.

Phase de Vol

Un vol est découpé en phases : décollage, 1500 pieds, atterrissage, etc. La phase de vol est le seul critère qui détermine si une alarme doit être inhibée ou

non. A chaque alarme est associé un ensemble de phases qui l'inhibent.

Commande de Contrôle

L'équipage dispose d'un nombre réduit de commandes qui permettent de contrôler l'émission de signaux d'alarme.

5.9.2 Comportement du Système

Fws doit calculer périodiquement les signaux à émettre. Pour cela il doit :
- Obtenir de l'environnement l'information sur l'état des dispositifs surveillés et sur les commandes envoyées par l'équipage.
- Activer les signaux associés aux alarmes actives.
- Calculer les signaux qui doivent être émis en tenant compte des commandes envoyées par l'équipage, et composer les messages à émettre.
- Envoyer ces messages aux dispositifs de sortie.

5.9.3 Propriétés du Système

Nous présentons les propriétés les plus abstraites du système.

Propriétés de bon fonctionnement

Elles expriment le comportement désiré du système.

Propriété FWS 5.1. *Les alarmes présentes dont la période de confirmation est écoulée sont des alarmes confirmées.*

Propriété FWS 5.2. *Les alarmes confirmées et non inhibées par la phase de vol courante sont des alarmes actives.*

Propriété FWS 5.3. *Les signaux d'alarme non annulés et qui sont associés à des alarmes actives sont des signaux actifs.*

Propriétés de sécurité

Ces propriétés garantissent que le système n'émet pas de signal erroné, c'est à dire des signaux qui ne reflètent pas l'état de l'appareil. Nous les appelons de sécurité plutôt que de sûreté car elles ont un impact direct sur la sécurité des voyageurs.

Propriété FWS 5.4. *Les alarmes confirmées sont des alarmes présentes dont la période de confirmation est écoulée.*

Propriété FWS 5.5. *Les alarmes actives sont des alarmes confirmées.*

Propriété FWS 5.6. *Les signaux actifs sont associés à des alarmes actives.*

Propriété FWS 5.7. *Les signaux émis sont des signaux actifs.*

Propriété FWS 5.8. *L'équipage est toujours en mesure de déterminer avec précision la situation d'alarme qui a généré l'émission d'un signal.*

5.9.4 Specification Formelle

Pour construire la spécification formelle nous allons adopter l'approche proposée dans [Lop 99]. Nous partirons d'une spécification abstraite qui contient très peu d'événements, puis nous allons introduire d'autres événements graduellement à l'aide de la relation de raffinement. Le cas d'étude du chapitre 6, page 119, est un exemple plus détaillé de cette approche.

Premier Model. FWS1

Cette spécification permet d'observer l'apparition de situations d'alarme et l'émission de signaux d'alarme. Ces observations correspondent à des événements réels.

Nous avons besoin de deux ensembles : Ww et Ss.

Nom	Description	Propriétés
Ww	Ensemble de toutes les alarmes	$Ww \neq \emptyset$
Ss	Ensemble de tous les signaux	$Ss \neq \emptyset$

Nous avons besoin de deux variables : wp et se.

Nom	Description	Propriétés
wp	Alarmes présentes	$wp \subseteq Ww$
se	Signaux émis	$se \subseteq Ss$

Au départ il n'y a pas d'alarme présente et aucun signal n'est émis. Le système est donc initialisé comme suit : $wp, se := \emptyset, \emptyset$

La spécification contient quatre événements : $NewWarning$, $EndWarning$, $NewSignal$, $EndSignal$. La spécification en B est :

SYSTEM FWS1	Spécification abstraite.
SETS Ww, Ss	Ensemble d'alarmes. Ensemble de signaux.
VARIABLES wp, se	Alarmes présentes. Signaux émis.
INVARIANT $wp \subseteq Ww \wedge$ $se \subseteq Ss$	
INITIALIZATION $wp, se := \emptyset, \emptyset$	
EVENTS NewWarning $\hat{=}$ ANY wx WHERE $wx \in Ww \wedge wx \notin wp$ THEN $wp := wp \cup \{wx\}$ END ;	Une nouvelle alarme aparaît.
EndWarning $\hat{=}$ ANY wx WHERE $wx \in wp$ THEN $wp := wp - \{wx\}$ END ;	Une situation d'alarme disparaît.
EmittedSignal $\hat{=}$ ANY sx WHERE $sx \in Ss \wedge sx \notin se$ THEN $se := se \cup \{sx\}$ END ;	Un signal est émis.
EndSignal $\hat{=}$ ANY sx WHERE $sx \in se$ THEN $se := se - \{sx\}$ END	L'émission d'un signal s'arrête.

Premier raffinement : FWS2

Dans ce raffinement nous faisons un choix de conception : nous allons utiliser deux processus cycliques asynchrones pour implanter le système. [3]. Le premier est chargé de confirmer les alarmes présentes et de calculer les paramètres du système. Par exemple la phase de vol. Le deuxième doit calculer et composer les signaux à émettre. Nous ne faisons aucune hypothèse sur le temps de calcul de chaque cycle. La spécification contient toutes les variables et tous les événements nécessaires à la description des interactions entre les deux processus et entre le système et son environnement. Nous avons besoin de deux nouveaux ensembles Cc et Pp :

Nom	Description	Propriétés
Cc	Ensemble de toutes les commandes	$Cc \neq \emptyset$
Pp	Ensemble de toutes les phases de vol	$Pp \neq \emptyset$

Nous laissons inchangées les variables du premier modèle et nous introduisons d'autres variables nécessaires à l'expression des nouvelles propriétés.

wc dénote les alarmes confirmées. Cette variable est partagée par les deux processus. Le premier doit la modifier lorsqu'il confirme une alarme et le deuxième doit la lire pour connaître les alarmes confirmées.

[3] Cette contrainte faisait partie de la description du système qui nous a été fournie par le donneur d'ordre.

pc dénote la phase de vol courante. Cette variable est partagée par les deux processus. Le premier doit la modifier lorsqu'il a calculée une nouvelle phase et le deuxième doit la lire à chaque début de cycle pour effectuer le calcul des signaux à émettre.

Les variables qui suivent permettent de spécifier le comportement de chaque processus. Lors de la transformation de la spécification événementielle en la spécification d'un module elles deviendront auxiliaires. Elles ne seront donc pas implantées.

$Wp1$ dénote les alarmes présentes vues par le processus 1. Au début du cycle $Wp1$ est vide.

$Wc2$ dénote les alarmes confirmées vues par le processus 2.

$Pc2$ dénote la phase de vol courante utilisée par le processus 2. Nous avons besoin de cette variable car pc peut changer pendant un cycle de calcul du processus 2.

$Cc2$ dénote les commandes à prendre en compte par le processus 2.

$We1$ dénote les alarmes déjà examinées par le processus 1.

$We2$ dénote les alarmes déjà examinées par le processus 2.

$Pe1$ détermine si la phase de vol a déjà été calculée par le processus 1.

$Pe2$ détermine si la phase de vol a déjà été lue par le processus 2.

$Ce2$ détermine si les commandes données par l'équipage ont déjà été lues par le processus 2.

$Num1$ Permet de compter les cycles calculés par le processus 1.

$Num2$ Permet de compter les cycles calculés par le processus 2.

En résumé, la spécification contient les variables suivantes :

Nom	Description	Propriété
wp	Alarmes Présentes.	$wp \subseteq Ww$
se	Signaux émis.	$se \subseteq Ss$
pc	Phase de vol courante. Partagée.	$pc \in Pp$
wc	Alarmes confirmées. Partagée.	$wc \subseteq Ww$
$Num1$	Nombre de cycles. P1.	$Num1 \in NAT$
$Wp1$	Alarmes présentes. P1.	$Wp1 \subseteq Ww$
$We1$	Alarmes examinées. P1.	$We1 \subseteq Ww$
$Pe1$	Phase de vol calculée : oui ou non. P1.	$Pe1 \in BOOL$
$Num2$	Nombre de cycles. P2.	$Num2 \in NAT$
$Wc2$	Alarmes confirmées. P2.	$Wc2 \subseteq Ww$
$We2$	Alarmes examinées. P2.	$We2 \subseteq Ww$
$Cc2$	Commandes. P2.	$Cc2 \subseteq Cc$
$Ce2$	Commandes lues. Oui ou non. P2	$Ce2 \in BOOL$
$Pc2$	Phase de vol courante. P2.	$Pc2 \in Pp$
$Pe2$	Phase de vol courante lue. Oui ou non. P2.	$Pe2 \in BOOL$

Ces propriétés forment l'invariant du système.

Nous ajoutons 9 nouveaux événements de la façon indiquée dans la section 5.6.1, page 102. Ils permettent d'observer les interactions des deux processus entre eux et les interactions du système avec l'environnement. La spécification de ces interactions constitue notre guide principal pour introduire les nouveaux événements. Le modèle B complet est :

SYSTEM
FWS2 — FWS2 raffine FWS1.

SETS
Ww, Ss, $\mathbf{Cc}, \mathbf{Pp}$

VARIABLES
wp, se, pc, wc,
$\mathbf{wp}, \mathbf{se}, \mathbf{pc}, \mathbf{wc}, \mathbf{Num1}, \mathbf{Wp1}, \mathbf{We1}, \mathbf{Pe1},$
$\mathbf{Wc2}, \mathbf{We2}, \mathbf{Cc2}, \mathbf{Ce2}, \mathbf{Pc2}, \mathbf{Pe2}, \mathbf{Num2}$

INVARIANT
$wp \subseteq Ww \wedge se \subseteq Ss \wedge$
$\mathbf{pc} \in \mathbf{Pp} \wedge \mathbf{wc} \subseteq \mathbf{Ww} \wedge \mathbf{Num1} \in \mathbf{NAT} \wedge$
$\mathbf{Wp1} \subseteq \mathbf{Ww} \wedge \mathbf{We1} \subseteq \mathbf{Ww} \wedge \mathbf{Pe1} \in \mathbf{BOOL} \wedge$
$\mathbf{Wc2} \subseteq \mathbf{Ww} \wedge \mathbf{We2} \subseteq \mathbf{Ww} \wedge \mathbf{Cc2} \subseteq \mathbf{Cc} \wedge$
$\mathbf{Ce2} \in \mathbf{BOOL} \wedge \mathbf{Pc2} \in \mathbf{Pp} \wedge$
$\mathbf{Pe2} \in \mathbf{BOOL} \wedge \mathbf{Num2} \in \mathbf{NAT}$

INITIALIZATION
wp, se, pc, wc, $Num1$, $Wp1$, $We1$, $Pe1$,
$Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$,$Num2$:
$(wp = \emptyset \wedge se = \emptyset \wedge \mathbf{pc} \in \mathbf{Pp} \wedge$
$\mathbf{wc} = \emptyset \wedge \mathbf{Num1} = \mathbf{0} \wedge \mathbf{Wp1} = \emptyset \wedge$
$\mathbf{We1} = \emptyset \wedge \mathbf{Pe1} = \mathbf{TRUE} \wedge \mathbf{Wc2} = \emptyset \wedge$
$\mathbf{We2} = \mathbf{Ww} \wedge \mathbf{Cc2} = \emptyset \wedge \mathbf{Ce2} = \mathbf{TRUE} \wedge$
$\mathbf{Pc2} = \mathbf{pc} \wedge \mathbf{Pe2} = \mathbf{TRUE} \wedge \mathbf{Num2} = \mathbf{0})$

Nous utilisons ici la forme $x : P(x, x_0)$.
Voir page 93.

EVENTS

NewWarning $\hat{=}$ — Nouvelle alarme présente.
ANY wx WHERE $wx \in Ww \wedge wx \notin wp$
THEN $wp := wp \cup \{wx\}$ END ;

EndWarning $\hat{=}$ — Fin d'une alarme.
ANY wx WHERE $wx \in wp$
THEN $wp := p - \{wx\}$ END ;

EmittedSignal $\hat{=}$ — Nouveau signal émis.
ANY sx WHERE $sx \in Ss \wedge sx \notin se$
THEN $se := se \cup \{sx\}$ END ;

EndSignal $\hat{=}$ — Fin d'émission d'un signal.
ANY sx WHERE $sx \in se$
THEN $se := se - \{sx\}$ END ;

ExamineWarning1 $\hat{=}$ — P1 examine l'état d'une alarme.
ANY wx WHERE $wx \in Ww \wedge wx \notin We1$ — wx n'a pas été examinée : $wx \notin We1$.
THEN
IF $wx \in wp$ THEN — L'alarme est présente.
$Wp1, We1 := Wp1 \cup \{wx\}, We1 \cup \{wx\}$
ELSE $We1 := We1 \cup \{wx\}$ END — L'alarme est absente.
END ;

ConfirmWarning1 $\hat{=}$ — P1 confirme une alarme.
ANY wx WHERE $wx \in Ww \wedge wx \notin wc$
THEN $wc := wc \cup \{wx\}$ END ;

AbsentWarning1 $\hat{=}$ — P1 indique qu'une alarme est absente.
ANY wx WHERE $wx \in Ww$
THEN $wc := wc - \{wx\}$ END ;

CurrentFlightPhase1 $\hat{=}$ — P1 calcule une nouvelle phase de vol.
ANY px WHERE $px \in Pp \wedge Pe1 = FALSE$
THEN $pc, Pe1 := px, TRUE$ END ;

BeginCycle1 $\hat{=}$	P1 démarre un nouveau cycle.
SELECT $We1 = Ww \wedge Pe1 = TRUE$	
THEN $We1, Wp1, Pe1, Num1 :=$	Toutes les alarmes seront examinées,
$\emptyset, \emptyset, FALSE, Num1 + 1$	une nouvelle phase de vol sera calculée.
END ;	Un nouveau cycle est comptabilisé.
ExamineWarning2 $\hat{=}$	P2 examine une alarme.
ANY wx WHERE $wx \in Ww \wedge wx \notin We2$	wx n'a pas été examinée : $wx \notin We2$.
THEN	
IF $wx \in wc$ THEN	
$Wc2, We2 := Wc2 \cup \{wx\}, We2 \cup \{wx\}$	L'alarme est confirmée.
ELSE $We2 := We2 \cup \{wx\}$ END	L'alarme n'est pas confirmée.
END ;	
CurrentFlightPhase2 $\hat{=}$	P2 lit la phase de vol.
SELECT $Pe2 = FALSE$	
THEN $Pc2, Pe2 := pc, TRUE$ END ;	
CrewCommands2 $\hat{=}$	P2 lit les commandes données
SELECT $Ce2 = FALSE$	par l'équipage.
THEN ANY Cx WHERE $Cx \subseteq Cc$	
THEN $Cc2, Ce2 := Cx, TRUE$ END	
END ;	
BeginCycle2 $\hat{=}$	P2 démarre un nouveau cycle.
SELECT	
$We2 = Ww \wedge Pe2 = TRUE \wedge Ce2 = TRUE$	
THEN	Toutes les alarmes seront examinées.
$We2, Wc2, Pe2, Ce2, Num2 :=$	La phase de vol et les commandes
$\emptyset, \emptyset, FALSE, FALSE, Num2 + 1$	données seront lues.
END ;	Un nouveau cycle est comptabilisé.

5.9.5 Dérivation du Module de Communication

Nous allons maintenant dériver le module de communication en utilisant la méthode décrite dans la section 5.7.1, page 104. Chaque événement sera transformé en une opération de la façon indiquée dans la section 5.7.2, page 105. Les événements $BeginCycle1$ et $BeginCycle2$ sont transformés en opérations auxiliaires car ils modifient uniquement des variables auxiliaires.

Les parties SETS, CONSTANTS, INVARIANT et INITIALIZATION sont exactement les mêmes que dans $FWS2$. Les variables sont divisées en deux groupes : abstraites et auxiliaires. Toutes les opérations dérivées prennent la forme de la formule 2.5 et satisfont la formule 2.6, voir page 60. Ainsi, la condition 2.2 est satisfaite. Nous obtenons la machine B suivante :

MACHINE	
FWS_ENVIRONMENT	Ce module est dérivé directement
	de la spécification FWS2.
VARIABLES	
wp, se, pc, wc	
VARIABLES AUXILIAIRES	
$Num1$, $Wp1$, $We1$, $Pe1$,	
$Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$	

OPERATIONS

NewWarning(wx) $\hat{=}$ — Action externe.
  PRE $wx \in Ww \wedge wx \notin wp$
  THEN $wp := wp \cup \{wx\}$ END ;

EndWarning(wx) $\hat{=}$ — Action externe.
  PRE $wx \in wp$
  THEN $wp := wp - \{wx\}$ END ;

EmittedSignal(sx) $\hat{=}$ — P2
  PRE $sx \in Ss \wedge sx \notin se$
  THEN $se := se \cup \{sx\}$ END ;

EndSignal(sx) $\hat{=}$ — P2
  PRE $sx \in Se$
  THEN $se := se - \{sx\}$ END ;

$bres \leftarrow-$ ExamineWarning1(wx) $\hat{=}$ — P1
  PRE $wx \in Ww \wedge wx \notin We1$
  THEN
    IF $wx \in wp$
    THEN $Wc1, We1, bres :=$
        $Wc1 \cup \{wx\}, We1 \cup \{wx\}, TRUE$
    ELSE $We1, bres := We1 \cup \{wx\}, FALSE$ END
  END ;

ConfirmWarning1(wx) $\hat{=}$ — P1
  PRE $wx \in Ww \wedge wx \notin wc$
  THEN $wc := wc \cup \{wx\}$ END ;

AbsentWarning1(wx) $\hat{=}$ — P1
  PRE $wx \in Ww$
  THEN $wc := wc - \{wx\}$ END ;

CurrentFlightPhase1(px) $\hat{=}$ — P1
  PRE $px \in Pp \wedge Pe1 = FALSE$
  THEN $pc, Pe1 := px, TRUE$ END ;

$bres \leftarrow-$ ExamineWarning2(wx) $\hat{=}$ — P2
  PRE $wx \in Ww \wedge wx \notin We2$
  THEN
    IF $wx \in wc$
    THEN $Wc2, We2, bres :=$
        $Wc2 \cup \{wx\}, We2 \cup \{wx\}, TRUE$
    ELSE $We2, bres := We2 \cup \{wx\}, FALSE$ END
  END ;

$pres \leftarrow-$ CurrentFlightPhase2 $\hat{=}$ — P2
  PRE $Pe2 = FALSE$
  THEN $Pc2, Pe2, pres := pc, TRUE, pc$ END ;

$cres \leftarrow-$ CrewCommands2 $\hat{=}$ — P2
  PRE $Ce2 = FALSE$
  THEN
    ANY Cx WHERE $Cx \subseteq Cc$
    THEN $Cc2, Ce2, cres := Cx, TRUE, Cx$ END ;
  END ;

OPERATIONS AUXILIAIRES

BeginCycle1 $\hat{=}$
  PRE $We1 = Ww \wedge Pe1 = TRUE$ — P1
  THEN $We1, Wp1, Pe1, Num1 := \emptyset, \emptyset, FALSE, Num1 + 1$ END ;

BeginCycle2 $\hat{=}$
  PRE $We2 = Ww \wedge Pe2 = TRUE \wedge Ce2 = TRUE$ — P2
  THEN $We2, Wc2, Pe2, Ce2, Num2 :=$
      $\emptyset, \emptyset, FALSE, FALSE, Num2 + 1$ END

Nous avons fait une simplification dans l'opération $CrewCommands2$ car *cres* est un ensemble. Cette variable n'est donc pas directement implantable. Ce problème peut être résolu en raffinant davantage FWS2. Nous ne le ferons pas ici.

5.9.6 Spécification de Composants

Nous pouvons spécifier maintenant séparément les processus P1 et P2 qui doivent coopérer à l'aide du module $FWS\_ENVIRONMENT$. Ils sont spécifiés de la façon indiquée dans le chapitre 3, page 71, en tant que modules séparés qui utilisent les variables du module partagé.

A titre d'exemple, nous allons spécifier le processus 2. La spécification du processus 1 peut être écrite d'une façon semblable. Nous ne le feront pas ici.

Les opérations utilisées exclusivement par le processus 2 sont : *Examine-Warning2*, *CurrentFlightPhase2*, *CrewCommands2*, *BeginCycle2*, *EmittedSignal* and *EndSignal*. Ces opérations modifient les variables suivantes : se, $Wc2$, $We2$, $Pc2$, $Pe2$, $Cc2$, $Ce2$ et $Num2$. Il est facile de vérifier que les autres opérations du module partagé ne modifient pas ces variables. Elles sont stables vis-à-vis du processus 2. Nous pouvons donc les utiliser dans la spécification de ce composant, voir condition 3.2, page 83.

Le processus 2 prend en compte les propriétés 2,3 et 5 à 8 du FWS données dans la section 5.9.1, page 108. Les propriétés 1 et 4 sont prises en compte par le processus 1. Nous insistons ici sur le fait que ces propriétés n'interviennent pas dans la spécification du module partagé ce qui simplifie considérablement la spécification de ce module. Pour exprimer les propriétés du processus 2, nous avons besoin des variables qui suivent [4] :

s_w est une fonction totale qui associe une alarme à chaque signal.

c_w est une relation entre les commandes et les alarmes.

wa dénote les alarmes actives.

wni dénote les alarmes non inhibées par la phase de vol.

$wnis$ dénote les alarmes non isolées.

sa dénote les signaux actifs.

snc dénote les alarmes annulées.

Nous utilisons les mêmes ensembles que dans $FWS\_ENVIRONMENT$: Ww, Ss, Cc et Pp. La spécification du processus 2 est :

[4] Ces variables sont présentées en détail dans [Lop 96-1, Lop 96-2].

MACHINE FWS_PROCESS2	Processus 2.
SETS Ww, Ss, Cc, Pp	
CONSTANTS s_w, c_w	
PROPERTIES $s\_w \in Ss \rightarrow Ww \wedge c\_w \in Cc \leftrightarrow Ww$	
VARIABLES se, $Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$, wa, wni, $wnis$, sa, snc	
INVARIANT $se \subseteq Ss \wedge Wc2 \subseteq Ww \wedge We2 \subseteq Ww \wedge$ $Cc2 \subseteq Cc \wedge Ce2 \in BOOL \wedge Pc2 \in Pp \wedge$ $Pe2 \in BOOL \wedge Num2 \in NAT \wedge wa \subseteq Wc2 \wedge$ $wni \subseteq Ww \wedge wnis \subseteq wa \wedge sa \subseteq Ss \wedge$ $se \subseteq sa \wedge snc \subseteq Ss \wedge$	
$Wc2 \cap wni \subseteq wa \wedge$	**(Propriété FWS 2.)**
$s\_w^{-1}[wa] \cap snc \subseteq sa \wedge$	**(Propriété FWS 3.)**
$wa \subseteq Wc2 \wedge$	**(Propriété FWS 5.)**
$s\_w[sa] \subseteq wa \wedge$	**(Propriété FWS 6.)**
$se \subseteq sa \wedge$	**(Propriété FWS 7.)**
$wnis \subseteq ran(c\_w) \wedge$	**(Propriété FWS 8.)**
$We2 = Ww \wedge Pe2 = TRUE \wedge Ce2 = TRUE$	**Restriction sur le module de communication.**
INITIALIZATION se, $Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$ wa, wni, $wnis$, sa, $snc : INVARIANT$	
OPERATIONS Cycle2 $\hat{=}$ se, $Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$ wa, wni, $wnis$, sa, snc : $(INVARIANT \wedge Num2 = Num2\_0 + 1)$	

Une comparaison entre la première spécification du FWS que nous avons écrite dans [Lop 96-1, Lop 96-2] et celle que nous donnons ici est faite dans [Lop,Sim,Don 2000].

5.10 Conclusion

La construction de la spécification formelle d'un système réel est difficile à maîtriser à cause de la quantité d'information qu'elle doit contenir. L'approche de spécification par événements combinée avec la notion de raffinement permet de construire graduellement une telle spécification ce qui favorise la maîtrise de la complexité.

Dans une spécification événementielle le système est observé dans son ensemble sans distinguer les composants qui le forment. Ceci permet d'exprimer des propriétés très abstraites qui portent sur le système dans sa globalité. Il est cependant très important de savoir implanter une spécification de ce type sans rompre le processus de raisonnement formel. Nous proposons une méthode qui permet de transformer la spécification événementielle en la spécification d'un

module partagé qui peut être implantée de la façon indiquée dans la section 3.4, page 84. La condition 5.1, page 104, est un moyen pratique qui permet d'effectuer cette transformation.

Le module partagé dérivé fournit les mécanismes de communication entre les composants du système et avec l'environnement. Les composants sont spécifiés séparément de la façon indique dans le chapitre 3, page 71. Ceci permet de structurer la spécification car la spécification du module partagé ne contient que les propriétés globales du système. Les propriétés d'un composant sont exprimées séparément dans la spécification du composant.

La méthode proposée permet de faire le lien entre la spécification d'un système et son implantation. Le raisonnement formel est préservé pendant tout le processus de construction du système.

Chapitre 6

Cas d'Etude Industriel

Pour effectuer notre travail de recherche nous avons jugé indispensable d'étudier un système industriel réel car nous pensons que des nombreux problèmes théoriques et pratiques apparaissent précisément lorsque nous sommes confrontés à des cas réels. Nous avons donc cherché un système dont le traitement par des méthodes formelles intéressait l'industrie.

Le cas d'étude a été fourni par les Sociétés RATP et Matra Transport International dans le cadre d'une étude de faisabilité d'utilisation de la méthode B très en amont dans le processus de construction d'un système. Nous devions étudier et modéliser la fonction assurer la protection des voyageurs de la ligne météor du métro Parisien. Le but final était de construire un modèle formelle qui exprime les propriétés qui garantissent que la ligne possède le niveau de sécurité requis.

Nous avions deux objectifs principaux :
- Evaluer la difficulté pour construire le modèle d'un tel sysème.
- Evaluer la possibilité d'utiliser le modèle obtenu pour dériver les spécifications formelles des composants du système.

Les spécifications informelle et formelle que nous présentons en annexe sont le résultat de ce travail qui a duré environ 10 mois. Plusieurs personnes du Cnam et des Sociétés RATP et MTI ont participé directement à leur élaboration et à leur validation. Ce travail est donc le résultat d'un effort commun entre industriels et le Cnam. La cohérence des spécifications formelles présentées et la correction des raffinements effectués ont été vérifiés avec l'atelier B dans le cadre de la préparation d'un mémoire de DESS au Cnam [1] .

Dans ce chapitre nous décrivons et analysons le processus de construction de ces spécifications.

[1] La vérification a été effectuée par Aymeric Burlureaux dans le cadre d'un stage de DESS réalisé à la RATP.

6.1 Construction de la Spécification Informelle

Nous avons suivi l'approche proposée dans [Lop 96-1, Lop 96-2]. Nous sommes partis des documents informels fournis par la RATP et MTI. Nous avons d'abord étudié en détail ces documents, puis nous avons réécrit la spécification informelle.

Nous avons créé un Comité chargé de valider la spécification produite. Il était formé par des spécialistes dans la spécification et la construction de ce type de système, et par des spécialistes dans l'utilisation des méthodes formelles. Ce Comité devait vérifier une à une toutes les propriétés exprimées dans la spécification, puis les approuver ou le rejeter. Il devait par ailleurs valider la spécification dans son ensemble. Nous pensons que ce rôle est primordial car le Comité est le mécanisme qui permet de certifier la bonne qualité de la spécification produite.

La réécriture de la spécification informelle a pris plus de six mois. Il s'agit d'un travail long et laborieux parce qu'il est nécessaire d'abord de bien comprendre le système dans son ensemble, puis de connaître et comprendre ses détails. Ceci est indispensable pour lui donner une bonne structure, ce qui est très important dans le travail ultérieur de modélisation.

Nous l'avons structurée en 6 parties comme suit :

- *Notions fondamentales* : Présente les notions les plus générales qui sont nécessaires à la compréhension du système et à l'expression de ses propriétés. Dans notre cas d'étude, elle donne les principes sur lesquels est conçue la protection des voyageurs de la ligne. Elle présente aussi les différents types de transfert qui peuvent être effectués, voir section A.1, page 129.
- *Eléments du Système* : Sert essentiellement à introduire le vocabulaire et les définitions qui sont nécessaires à l'expression des propriétés.
- *Description du Système* : Décrit le fonctionnement du système et donne au lecteur une vision d'ensemble sur son comportement.
- *Propriétés* : Donne une à une les propriétés que le système doit satisfaire. Elles déterminent le comportement souhaité. Les propriétés sont introduites dans un ordre qui suit le niveau d'abstraction des idées qu'elles expriment. Cet aspect doit aussi être vérifié et validé par le Comité.
- *Détails* : Donne d'autres propriétés qui introduisent un premier niveau de détail du système.
- *Composants* : Présente les composants logiciel et matériel qui seront utilisés pour implanter le système de contrôle de la ligne. Les interfaces de ces sous-systèmes sont décrites en détail.

Nous pensons que cette structure peut être utilisée d'un façon systématique dans la spécification de systèmes.

La principale difficulté que nous avons rencontré dans l'élaboration de la spécification informelle a été d'acquérir un niveau de compréhension du système suffisant pour identifier ses propriétés et pour les classer par groupes suivant leur niveau d'abstraction. A notre avis, deux causes sont à l'origine de cette difficulté : la quantité importante d'information contenue dans les documents qui nous ont été fournis, plus de 1.000 pages ; et la complexité du système étudié.

6.2 Construction de la Spécification Formelle

La construction de la spécification formelle est fortement guidée par les propriétés exprimées dans la spécification informelle. L'idée principale est de les modéliser une à une en respectant leur ordre de présentation : commencer par les propriétés les plus abstraites, puis introduire un premier niveau de détail, ensuite un deuxième, etc. Les propriétés qui concernent un seul composant sont laissées de côté. Elles seront introduites plus tard lors de la spécification du composant. Nous avons procédé en trois étapes :

- *Analyse de la description générale du système donnée dans les sections A.1, A.2 et A.3, page 129* : Le but est de relever les notions fondamentales et de les classer par niveau d'abstraction. Nous espérions par la suite pouvoir les introduire dans l'ordre défini par ce classement.
- *Analyse des propriétés fondamentales du système données dans la section A.4, page 139* : Le but est d'identifier les propriétés qui doivent être exprimées dans la spécification système et de les classer par groupes suivant leur niveau d'abstraction. Ce travail est facilité par la structure déjà donnée à la spécification informelle.
  Le premier groupe de propriétés que nous avons identifié est formé par les quatre principes fondamentaux sur lesquels est fondée la sécurité des voyageurs. Ce sont les propriétés les plus abstraites du système.
- *Expression formelle des propriétés de chaque groupe* : Nous écrivons ici les formules qui expriment les propriétés à modéliser. L'annexe B contient la spécification formelle du premier groupe de propriétés : les principes fondamentaux.

Le processus de spécification formelle se termine lorsque nous avons introduit toutes les propriétés qui déterminent les mécanismes de communication et de synchronisation du système.

Pour introduire les propriétés de chaque groupe nous procédons d'une façon très systématique, comme suit :

- *Analyse des propriétés à modéliser* : D'abord, nous cherchons à relever les notions importantes utilisées dans les propriétés du groupe.
- *Définition d'objets mathématiques* : Ensuite, nous introduisons les objets mathématiques qui serviront à exprimer ces propriétés : variables, constantes, ensembles, relations, fonctions, etc. Chaque objet introduit est justifié.
- *Introduction des propriétés de typage* : Nous complétons la phase précédente avec des formules qui précisent le type des objets utilisées dans la spécification. Elles sont tirées de la connaissance globale du système. Ces formules font partie de l'invariant du module.
- *Introduction d'invariants* : Après, nous écrivons les formules invariantes

qui correspondent aux contraintes statiques exprimées dans les propriétés à modéliser.

- *Introduction des événements* : Puis, nous identifions les événements que le système permet d'observer. Nous sommes guidés ici par les interactions du système avec son environnement : chaque événement de la spécification événementielle finale sera transformé en une opération de communication du système.

L'introduction d'objets mathématiques est une phase particulièrement importante car le choix des ces objets impacte directement l'écriture des formules que la spécification contiendra. L'introduction d'événements est réalisée graduellement à l'aide de la relation de raffinement. Les principes fondamentaux ont été modélisés en quatre étapes. A chaque étape correspond une spécification.

La première spécification, page 168, contient 5 événements. Un qui **permet d'observer tous les transferts effectués dans la ligne** : *Transfert*. Il est observé chaque fois qu'un transfert est réalisé. Nous devons inclure aussi les événements qui prennent en compte l'apparition de **situations dangereuses dans la ligne** : *TrainDanger, QuaiDanger, TrainSur, QuaiSur*.

La deuxième spécification, qui correspond au premier raffinement, page 173, **permet d'observer les détails d'un transfert normal de voyageurs**. L'événement *Transfert* est décomposé en deux : *TransfertVoy* qui permet d'observer la fin d'un transfert normal de voyageurs, et *Transfert1* qui permet d'observer les autres types de transfert, voir section A.1.1, page 130. De cette façon nous pouvons introduire des précisions sur les observations que le système doit admettre. Nous incluons aussi un événement pour observer le début d'un transfert normal de voyageurs : *TransfertVoyDeb*.

Cette deuxième spécification contient quatre autres événements : *TrainImmobil* et *TrainMobil* permettent de prendre en compte la notion d'immobilisation d'un train, voir section A.3.2, page 136. *ArriveeQuai* et *DepartQuai* permettent de prendre en compte la notion de bon positionnement d'un train. La deuxième spécification contient en total 11 événements.

La troisième spécification, qui correspond au deuxième raffinement, page 178, **permet d'observer l'ouverture et la fermeture des portes des trains et des quais de station**. Les événements *TrainOuvert* et *TrainFerme* prennent en compte la notion d'ouverture d'un train. Les événements *QuaiOuvert* et *QuaiFerme* prennent en compte la notion d'ouverture d'un quai. Les événements *ZoneQuaiSec* et *ZoneQuaiNor* prennent en compte la notion de mise en sécurité d'un quai. Toutes ces notions sont présentées dans la spécification informelle. La troisième spécification contient en total 17 événements.

La quatrième spécification, qui correspond au troisième raffinement, page 182, **permet d'observer les détails d'une évacuation d'urgence**, ce qui correspond à un autre type de transfert. L'événement *Transfert1* est décomposé en deux : *Evacuation* qui permet d'observer la fin d'une évacuation d'urgence, et *Transfert2* qui permet d'observer les autres types de transfert, voir section A.1.1, page 130. Nous incluons aussi un événement pour observer le début d'une

évacuation d'urgence : *EvacuationDeb*.

Cette quatrième spécification contient quatre autres événements : *EntreeZoneEva* et *SortieZoneEva* prennent en compte la notion d'entrée d'un train dans une zone d'évacuation. Les événements *ZoneEvacSec* et *ZoneEvacNor* prennent en compte la notion de mise en sécurité d'une zone d'évacuation. Toutes ces notions sont présentées dans la spécification informelle et sont indispensables pour exprimer les quatre principes fondamentaux.

Cette Dernière spécification est difficile à appréhender car elle contient, de notre point de vue, un nombre très important d'événements : 23 en total. Utiliser les gardes de ses événements pour introduire d'autres contraintes du système nous semblait être un travail très difficile à faire et surtout à valider par le Comité de spécialistes.

Nous avons donc décidé d'analyser cette spécification afin de mieux identifier les difficultés qu'elle engendrait. Nous avons trouvé les trois problèmes présentés dans l'introduction du document :

- Le nombre important d'événements qu'il faut inclure.
- L'implantation et donc la dérivation des spécifications des composants du système.
- L'implantation des variables utilisées qui sont toutes très abstraites.

Nous avons arrêté à ce point le processus d'expression formelle de propriétés et nous nous sommes concentrés sur la recherche de solutions aux problèmes énoncés, ce qui a aboutit aux résultats présentés dans les chapitres précédents.

Ce cas d'étude comporte une difficulté supplémentaire qui est due au fait que nous voulions étudier uniquement une fonction du système : la sécurité des voyageurs pendant les transferts. Le problème est que pour étudier cette partie du système il est indispensable de prendre en compte beaucoup d'autres notions qui n'interviennent pas directement dans la sécurité pendant les transferts mais qui le font indirectement.

Nous pensons que pour valider par la pratique les méthodes et techniques que nous proposons, il est préférable d'étudier un système plus petit mais dans sa totalité.

Chapitre 7

Conclusion

Sujets Abordés

Notre travail a été consacré à l'étude de la construction de spécifications formelles de systèmes. Ce sujet n'est pas nouveau. Il existe un bon nombre de méthodes et d'approches qui ont été proposées pour le faire. Il y a cependant un aspect essentiel qui est laissé de coté dans la plus part des cas : *l'implantation de la spécification produite*. Nous pensons que ce point est fondamental dans le processus de construction d'un système car dissocier une spécification de son implantation serait une source d'erreur importante.

Nous avons tenté de répondre à la question suivante : *Comment pouvons-nous construire la spécification formelle d'un système de telle sorte qu'elle permette de dériver les spécifications formelles des composants nécessaires à l'implantation du système ?*

Nous voulions en plus utiliser les techniques connues de développement formelle de programmes pour implanter les spécifications obtenues, et permettre la réutilisation de composants existents.

Nous nous sommes placés dans le cadre de la construction de systèmes par raffinements successifs et nous avons utilisé la méthode B en tant qu'outil de spécification et de développement.

Difficultés Rencontrées

Pour effectuer nos recherches nous avons décidé d'étudier un système réel fourni par l'industrie que nous avons spécifié suivant une approche par événements. Nous avons rencontré trois difficultés principales :

1. Le nombre d'événements qu'il faut inclure dans la spécification croît rapidement avec l'introduction de détails du système. La spécification devient alors très difficile à maîtriser.
2. Le problème de l'implantation de la spécification obtenue et de la dérivation des spécifications de composants n'est pas abordé. L'implantation est déconnectée de la spécification.

3. La spécification contient des variables abstraites qui permettent d'enregistrer l'évolution du système. Ces variables sont très utiles dans la phase de spécification mais elles doivent disparaître pendant le processus d'implantation.

La suite du travail a été dédiée à l'étude de ces problèmes.

Solutions Proposées

Pour traiter le premier problème, nous proposons une méthodologie pour construire graduellement la spécification événementielle d'un système. Nous incluons dans la spécification système uniquement les événements qui permettent d'observer les échanges d'information entre le système et son environnement, et entre ses composants. Les composants du système sont spécifiés séparément.

Pour traiter le deuxième problème, nous proposons la notion de module partagé. Cette notion est utilisée en programmation pour prendre en compte les aspects concurrents d'un système. Nous dérivons de la spécification événementielle la spécification d'un module partagé qui décrit tous les mécanismes de communication. Elle servira à implanter les composants du système. Les spécifications obtenues sont implantées à l'aide de techniques connues de raffinement.

Pour traiter le troisième problème, nous utilisons la notion de variable auxiliaire. Ces variables servent soit à exprimer des propriétés qui font intervenir le comportement du système, soit à montrer la correction des implantations effectuées. Elles ne sont pas implantées.

Apport du Travail

Le modèle relationnel que nous présentons dans le chapitre 1 permet d'exprimer la théorie du raffinement d'une façon simple et concise. Les idées qu'il contient son bien connues, mais nous pensons que ce modèle aide à mieux les comprendre.

La méthode de spécification proposée dans le chapitre 2 fournit une réponse adéquate au problème de la spécification des interactions d'un système avec son environnement. Elle permet de spécifier finement les comportements observables que le système doit admettre, ce qui à notre sens est le but principal de la spécification d'un système.

L'utilisation que nous faisons de variables auxiliaires et d'opérations auxiliaires répond bien au problème de la réutilisation de spécifications. Ce résultat nous parait important car il permet de partir d'une spécification générale d'un dispositif et de la compléter ensuite avec des variables auxiliaires afin de l'utiliser dans la solution de problèmes particuliers. L'implantation fournie pour la spécification du départ est réutilisée dans chaque cas particulier. Le cadre d'utilisation de variables auxiliaires et d'opérations auxiliaires, que nous définissons à travers les formules 2.5, 2.5a, 2.6 et 2.6a peut être appliqué dans la construction de systèmes industriels réels.

La notion de module partagé présentée dans le chapitre 3 permet de prendre en compte les aspects concurrents d'un système et répond bien au besoin de

structuration de la spécification. Nous donnons des conditions d'utilisation, condition 3.2, page 83, qui peuvent être appliquées dans la pratique pour construire des systèmes concurrents réels.

La méthodologie de construction de la spécification formelle d'un système fournie dans le chapitre 5 peut être utilisée d'un façon systématique. Le cas d'étude présenté dans le chapitre 6 est un exemple d'un système industriel spécifié suivant notre approche.

La transformation d'une spécification événementielle en une spécification pré-post permet de relier le modèle global d'un système aux spécifications des modules nécessaires à son implantation. Cette technique apporte une réponse claire au problème de la séparation entre un modèle système et son implantation : le raisonnement formel est préservé durant tout le processus de conception et de développement. En plus, elle rend possible l'utilisation de techniques de dérivation de programmes connues. La condition 5.1, page 104, fournit un moyen pratique d'utilisation de cette méthode de transformation.

Travaux Futurs et Perspectives

Pour pouvoir utiliser dans l'industrie cette méthodologie, il est nécessaire de développer des outils qui prennent en compte les notions de variable auxiliaire, d'opération auxiliaire et de module partagé. Nous pensons que ces notions peuvent être introduites dans la méthode B.

D'un autre coté, Il est indispensable de valider les techniques proposées avec le développement complet de systèmes réels de petite taille. Ceci permettrait de déterminer leur apport dans un contexte industriel et aussi leur limites.

Annexe A

Cas d'Etude : Spécification Informelle

La spécification informelle de la fonction assurer la protection des voyageurs de la ligne Météor du métro parisien est formée par six parties. La première introduit les notions fondamentales utilisées dans la description de cette fonction du système. La deuxième décrit les éléments qui forment la ligne météor. La troisième décrit le comportement général attendu du système de contrôle. La quatrième contient les propriétés fondamentales les plus abstraites. La cinquième introduit un premier niveau de détail sur le comportement attendu du système. La sixième contient un premier découpage du système de contrôle suivant les équipements utilisés dans son implantation.

A.1 Notions fondamentales

Le but de la ligne Météor est de transporter en toute sécurité les voyageurs de la ligne. La protection des voyageurs et du personnel d'exploitation est conçue en s'appuyant sur les quatre principes fondamentaux qui suivent :

Principe A.1. *Les voyageurs se trouvant à bord des trains et à quai ne sont protégés que dans la mesure où ces deux univers sont hermétiquement clos vis-à-vis du site dans lequel les trains sont en mouvement.*

Principe A.2. *Il est nécessaire d'assurer la protection des voyageurs pendant le transfert entre le quai et le train lors des arrêts en station en assurant l'immobilité du train.*

Principe A.3. *Les voyageurs se trouvant à bord d'un train ne sont en sécurité que si aucune situation dangereuse n'apparaît à bord ou si, en cas de situation dangereuse (incendie par exemple), ils ont en dernier recours la possibilité de*

quitter le train en urgence, où qu'il soit (évacuation), et de regagner la sortie de secours la plus proche (station) dans un environnement rendu sûr.

Principe A.4. *D'autre part, le système doit autoriser avec un minimum de perturbation du trafic certaines interventions du personnel d'exploitation et de maintenance. En particulier, la pénétration de personnel habilité dans les zones où les trains circulent doit être possible avec un maximum de garanties de sécurité.*

Les personnes peuvent accéder ou quitter les trains seulement dans un des quatre cas qui suivent :

- Le transfert de voyageurs.
- L'évacuation d'urgence d'un train.
- Le transfert de service.
- Les tests de fonctionnement de portes.

A.1.1 Le transfert de voyageurs

Ce cas se présente lorsqu'un train en conditions normales d'exploitation doit desservir une station. Le train doit s'arrêter dans la station et un transfert de voyageurs doit avoir lieu.

A.1.2 L'évacuation d'urgence d'un train

Lorsqu'une anomalie apparaît ou lorsqu'une situation dangereuse se présente dans un train, il peut être nécessaire d'effectuer une évacuation d'urgence du train. Dans ce cas, le système doit permettre l'évacuation en assurant la sécurité des voyageurs soit en station, soit en inter-station (entre deux stations).

A.1.3 Le transfert de service

Lorsqu'un train se trouve dans une zone de garage ou de maintenance, le personnel de maintenance et d'exploitation doit pouvoir accéder et quitter le train.

A.1.4 Les tests de fonctionnement de portes

Avant de rendre un train disponible pour entrer en exploitation, le bon fonctionnement de ses portes doit être vérifié. Dans ce cas, le personnel d'exploitation doit pouvoir accéder et quitter le train.

A.2 Eléments qui forment la ligne météor

Dans le contexte du problème traité ici, la ligne Météor est formée par :

- Un ensemble de voies sur lesquelles peuvent circuler des trains.
- Un ensemble de trains, en principe sans conducteur, qui peuvent circuler sur les voies.
- Un ensemble de stations où peuvent être effectués des transferts de voyageurs entre les trains et les quais. Ces transferts peuvent être effectués seulement sous certaines conditions.
- Des zones de garage et de maintenance de trains.
- Un poste de commande centralisé (PCC) qui permet au personnel d'exploitation de la ligne d'agir sur les éléments du système (trains, quais de station, etc.).
- Un système qui contrôle et commande automatiquement l'exploitation de la ligne.

L'exploitation de la ligne doit être effectuée d'une façon automatique, comme suit : Un opérateur doit pouvoir indiquer un programme d'exploitation au système de contrôle. Celui-ci doit ensuite décider sur les actions qui doivent être effectuées afin d'accomplir le programme indiqué. Il doit en plus envoyer les commandes nécessaires à la réalisation de ces actions.

Lorsque des conditions anormales surviennent, des interventions humaines doivent être possibles. Par exemple lorsqu'un train est en panne il doit être possible de l'actionner en conduite manuelle.

A.2.1 Les voies où peuvent circuler les trains

La ligne météor comporte deux types de voie : principales et secondaires.

Sur les voies principales circulent les trains en exploitation. Elles traversent toutes les stations desservies par la ligne. Les voies secondaires relient les voies principales avec d'autres voies ou avec les zones de garage et de maintenance.

Sectionnement de la voie

L'alimentation en énergie électrique de chaque voie est effectuée par section de voie. L'alimentation de chaque section peut être contrôlée d'une façon indépendante. Il est par conséquent possible d'alimenter partiellement une voie.

Lorsqu'une situation dangereuse se produit dans un train, il peut s'avérer nécessaire d'isoler une partie de la voie. Dans ce cas, la coupure sera effectuée sur l'ensemble de segments qui alimentent la partie à isoler. Cette isolation doit prévenir contre le risque d'électrocution dû à la libération d'énergie des trains qui se trouvent sur la voie et qui effectuent un freinage. Ce phénomène est appelé *freinage par récupération*.

Découpage en zones de la voie

Une voie peut être découpée logiquement en zones. Un découpage en zones obéit à un critère spécifique. Dans le cas de l'évacuation d'urgence d'un train

en inter-station, un découpage en zones est effectué afin de garantir la sécurité des personnes sur la voie, en maintenant une relative disponibilité de la ligne.

A.2.2 Les trains

Un train est formé par un groupe de voitures. Chaque voiture possède un ensemble de portes dont le but est de permettre aux voyageurs d'accéder au train et de le quitter. Ces portes sont localisées des deux cotés du train.

Les portes des trains

Les portes d'un train sont commandés automatiquement par le système de contrôle. Lorsque des situations anormales se produisent, elles peuvent être actionnées soit par les voyageurs soit par le personnel d'exploitation de la ligne.

Lorsque les mécanismes de traction d'une porte exercent une pression sur celle-ci, nous dirons que la porte est non libre. Lorsque aucune pression n'est exercée sur la porte, nous dirons qu'elle est libre.

Une porte peut se trouver dans un des états qui suivent :

- *Non verrouillée.* Une porte non verrouillée peut être soit libre soit non libre.
- *Verrouillée.*

Chaque porte est dotée d'un mécanisme de verrouillage dont le but est d'empêcher l'ouverture intempestive de la porte. Une porte verrouillée ne peut pas être ouverte. Une porte non verrouillée peut être fermée. Dans ce cas, si la porte est libre une action motrice est nécessaire afin de l'ouvrir.

Les poignées de déverrouillage

Chaque porte est dotée d'une poignée de déverrouillage (KSA) qui permet, en cas de danger, une évacuation d'urgence. Ces poignées peuvent être actionnées seulement de l'intérieur du train.

L'actionnement d'une poignée de déverrouillage a pour vocation d'arriver à un déverrouillage des portes localisées du coté opposé à la voie lorsque le train est arrêté. Si le train est en mouvement, le système de contrôle attendra l'arrêt du train avant d'effectuer ce déverrouillage. Juste après le déverrouillage, les portes fermées ne sont pas ouvertes. Une action manuelle est nécessaire afin de les ouvrir.

Les poignées de déverrouillage de service

Les portes des extrémités des trains sont dotées d'une poignée de déverrouillage de service (KSE) qui permet au personnel d'exploitation de la ligne d'accéder au train et de le quitter, par exemple pour une reprise du train en conduite manuelle. Ces poignées peuvent être actionnées de l'extérieur et de l'intérieur du train.

L'actionnement d'une poignée de déverrouillage de service a pour vocation d'arriver à un déverrouillage de la porte concernée. Ceci permet au personnel d'exploitation d'accéder au train. Ce déverrouillage n'est effectué que lorsque le train est arrêté.

Les boutons d'alarme voyageur embarqués

Sur chaque voiture, il existe plusieurs boutons mis à la disposition des voyageurs qui permettent d'envoyer un signal d'alarme au poste de commande centralisé lorsqu'une situation dangereuse se produit. Ces boutons sont appelés *boutons d'alarme voyageur*.

A.2.3 Les quais de station

Afin d'assurer la sécurité des usagers, chaque quai de station est complètement isolé de la voie de circulation de trains. Pour permettre les transferts de voyageurs et les interventions du personnel d'exploitation de la ligne chaque quai est doté de :

- Un ensemble de portes palières qui permettent les échanges de voyageurs. Il existe au moins autant de portes palières sur un quai que de portes sur un train.
- Deux portillons d'extrémité de quai qui permettent au personnel d'exploitation de la ligne d'accéder à la voie de circulation des trains.
- Un ensemble de portes de secours disposées en alternance des portes palières. Ces portes peuvent être ouvertes seulement depuis la voie.
- Un bouton d'alarme, placé au milieu du quai, qui permet aux voyageurs d'entrer en communication avec le personnel d'exploitation.
- Un poste de contrôle qui permet au personnel d'exploitation d'agir sur les éléments localisés dans la station (portes palières, portillons d'extrémité de quai, portes de secours). Ce poste de contrôle est appelé pupitre manuel de station.

Les portes palières

Les portes palières sont commandées automatiquement par le système de contrôle. Lorsque des situations anormales se produisent, elles peuvent être actionnées par le personnel d'exploitation de la ligne.

Une porte palière peut se trouver dans un des états qui suivent :

- *Non verrouillée.* Une porte palière non verrouillée peut être soit libre soit non libre (voir section A.2.2, page 132. Les portes des trains).
- *Verrouillée.*

Chaque porte est dotée d'un mécanisme de verrouillage dont le but est d'empêcher l'ouverture intempestive de la porte. Une porte verrouillée ne peut pas être ouverte. Par contre, une porte non verrouillée peut être fermée; dans ce cas, si la porte est libre une action motrice est nécessaire afin de l'ouvrir.

Chaque porte est dotée d'un dispositif de déverrouillage qui peut être actionné du coté voie.

Les portillons d'extrémité de quai

Les portillons d'extrémité de quai sont fermés et verrouillés pour empêcher l'intrusion de personnes dans la voie.

Un portillon peut se trouver dans un des deux états qui suivent :

- *Non verrouillé.*
- *Verrouillé.*

Chaque portillon est doté d'un dispositif de déverrouillage. Un portillon verrouillé ne peut pas être ouvert.

Les portes de secours

Les portes de secours ne sont pas commandées par le système de contrôle. Lorsque des situations dangereuses se produisent, elles peuvent être actionnées par le personnel d'exploitation de la ligne.

Une porte de secours peut se trouver dans un des deux états qui suivent :

- *Ouverte.*
- *Fermée.*

Chaque porte de secours est dotée d'un dispositif d'ouverture anti-panique situé du coté voie.

Le bouton d'alarme voyageur à quai

Sur chaque quai de station, il existe un bouton qui permet aux usagers d'entrer en contact avec le personnel d'exploitation.

Le pupitre manuel de station

Chaque station est doté d'un dispositif de contrôle qui permet aux agents de station d'agir sur les éléments localisées dans la station. Ce dispositif est appelé pupitre manuel de station (PMS).

Le PMS permet de :

- Commander l'ouverture et la fermeture des portes palières.

- Inhiber les contrôles de fermeture et de verrouillage des portes constituant la façade du quai.
- Autoriser le départ du train arrêté dans la station.
- Provoquer un arrêt d'urgence des trains circulant sur la zone et de couper la haute tension sur les segments de voie associés à la station.

A.2.4 Les zones de garage et de maintenance

La ligne dispose de zones de garage dans lesquelles peuvent être garés les trains qui ne sont pas en exploitation.

La ligne dispose en plus de zones de maintenance de trains. Les opérateurs de la ligne ont la possibilité de changer une zone d'exploitation en zone de garage et vice-versa.

A.2.5 Le poste de contrôle centralisé (PCC)

La ligne dispose d'un poste de contrôle centralisé qui permet aux opérateurs d'agir sur les éléments qui la forment.

A.2.6 Le système de contrôle et de commande automatique

L'exploitation de la ligne Météor est assurée par un système de contrôle qui permet en conditions normales de fonctionnement, l'automatisation intégrale de cette exploitation. Ce système de contrôle est décrit en détail dans la section suivante.

A.3 Le système de contrôle de météor (SAET)

Le système SAET (système d'automatisation de l'exploitation des trains) a pour but d'assurer l'exploitation automatique de la ligne et de permettre au personnel d'exploitation d'agir sur les éléments qui la forment (trains, quais, etc.).

Ce système est formé par les éléments qui suivent :

- Dans le poste de contrôle centralisé sont installés des équipements qui permettent de surveiller la ligne tout entière et d'agir sur ses éléments.
- Dans chaque train sont installés des équipements sur lesquels fonctionne le sous-système chargé de surveiller et contrôler les éléments du train (portes du train, mécanismes de traction, etc.). Ce sous-système communique avec le poste de contrôle centralisé.
- Dans certaines stations sont installés des équipements sur lesquels fonctionne le sous-système chargé de surveiller et contrôler les éléments des

quais (portes palières, portes de secours, etc.). Ce sous-système communique avec le poste de contrôle centralisé.

A.3.1 Fonctionnement général du SAET

Pour exploiter la ligne, le personnel d'exploitation doit fournir au système un programme d'exploitation. Le système doit alors décider sur les actions nécessaires à sa réalisation. D'un point de vue très général, le système SAET doit réaliser les tâches qui suivent :

- Décider sur les trains qui doivent être mis en exploitation, et envoyer les commandes nécessaires à la mise en exploitation effective de ces trains.
- Décider sur les trains qui doivent être mis en zone de garage ou de maintenance, et envoyer les commandes nécessaires à la réalisation effective de ces actions.
- Définir pour chaque train la mission qui lui est assignée et la lui communiquer.
- Surveiller en permanence l'exploitation de la ligne et déclencher les actions nécessaires lorsque des situations anormales ou dangereuses se produisent.

Le SAET intègre une fonction de contrôle sécuritaire qui délivre, entre autres, les autorisations d'ouverture et de fermeture des portes et de démarrage des trains.

A.3.2 Le transfert de voyageurs

Les trains circulant sur la ligne doivent s'arrêter dans les stations indiquées dans la mission de chaque train, pour permettre aux voyageurs d'aborder les trains et de les quitter.

En conditions normales, un transfert de voyageurs est effectué en plusieurs phases en séquence, comme suit :

- Le train arrive à la station et s'arrête sur le quai de telle sorte que les portes du train soient en face à face avec les portes palières du quai.
- L'autorisation d'ouverture des portes du train est donnée par la fonction de contrôle sécuritaire.
- L'autorisation d'ouverture des portes palières est donnée par la fonction de contrôle sécuritaire.
- Les portes du train et les portes palières s'ouvrent presque simultanément. Ces portes restent ouvertes pendant une durée de temps prédéfinie.
- Le portes du train et les portes palières se ferment presque simultanément.

- L'autorisation de démarrage du train est donnée par la fonction de contrôle sécuritaire.

Autorisation d'ouverture des portes d'un train

Avant de déclencher l'ouverture des portes d'un train, le système de contrôle doit obtenir l'autorisation correspondante qui est délivrée par la fonction de contrôle sécuritaire. Cette autorisation n'est donnée que si les conditions suivantes sont vérifiées :

- Le train est immobilisé. Un train immobilisé est un train qui est arrêté et pour lequel il n'y pas de demande de mise en mouvement en cours. Un train immobilisé ne peut pas repartir sans qu'il y ai préalablement une autorisation de mise en mouvement du train.
- Le train est positionné à quai, de manière telle que toutes ses portes soient en face de portes palières. Dans le cas où le quai soit plus long que le train, chaque porte du train doit être en face à face avec une porte palière.
- Les portes à ouvrir sont localisées du coté quai.

Nous les appellerons conditions d'autorisation d'ouverture des portes du train.

Autorisation d'ouverture des portes palières

Avant de déclencher l'ouverture des portes palières, le système de contrôle doit obtenir l'autorisation correspondante qui est délivrée par la fonction de contrôle sécuritaire. Cette autorisation n'est donnée que si les conditions suivantes sont vérifiées :

- Le train est immobilisé.
- Le train est positionné à quai, de manière telle que toutes ses portes soient en face de portes palières. Sur la ligne peuvent circuler deux types de trains de longueur différente : 6 voitures et 8 voitures. Dans le cas des trains à 6 voitures, certaines portes palières ne seront pas en face à face avec les portes du train ; par conséquent, elles ne seront pas ouvertes lors du transfert. Dans le cas des trains à 8 voitures, toutes les portes palières doivent être en face à face avec une des portes du train.
- L'autorisation d'ouverture des portes du train a été donnée.

Nous les appèlerons conditions d'autorisation d'ouverture des portes palières.

Autorisation de mise en mouvement des trains

Avant de déclencher le démarrage d'un train, le système de contrôle doit obtenir l'autorisation correspondante qui est délivrée par la fonction de contrôle

sécuritaire. Cette autorisation n'est donnée que si les conditions suivantes sont vérifiées :

- Toutes les portes du train sont fermées et verrouillées, et il y a absence d'autorisation d'ouverture de portes du train.
- Toutes les portes palières sont fermées et verrouillées, et il y a absence d'autorisation d'ouverture de portes palières.

A.3.3 Les situations dangereuses à bord des trains

Lorsque une situation dangereuse se présente dans un train, les voyageurs peuvent actionner la poignée de déverrouillage d'une des portes du train. Dans ce cas, deux possibilités se présentent :

- Le train peut être conduit jusqu'à la station la plus proche. Dans ce cas le train sera vidé dans cette station.
- Le train ne peut pas être conduit dans une station. Dans ce cas, une évacuation d'urgence en inter station doit être réalisée. Pour cela, le système de contrôle doit mettre en sécurité la zone où circule le train, ce qui entraîne le déclenchement des actions qui suivent :
  - Effectuer un arrêt d'urgence du train concerné.
  - Effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone où circule le train concerné.
  - Couper la haute tension sur la zone ou circule le train concerné.

Dès que la zone est en sécurité, le système de contrôle doit permettre le vidage du train en (inter station). Ceci implique la possibilité d'ouvrir une ou plusieurs portes du train en inter station.

La décision sur le lieu où doit avoir lieu le vidage du train (quai de station ou inter-station) est prise par le système de contrôle.

A.3.4 Panne du système de contrôle

En cas de panne du système de contrôle, ou en absence d'énergie à bord du train, le portes du train sont maintenues verrouillées mécaniquement, ce qui interdit leur ouverture intempestive. Dans cette situation, une porte peut être déverrouillée en actionnant sa poignée de déverrouillage.

A.3.5 Ouverture intempestive d'une porte d'un train en mouvement

Quand le système de contrôle détecte une ouverture d'une porte qui ne correspond pas à une demande effective, il doit considérer que la porte s'est ouverte d'une façon intempestive.

Lorsque le système obtient des informations contradictoires qui indiquent qu'une porte n'est pas fermée et que, au même temps, elle est verrouillée, il doit considérer qu'il y a un défaut dans les dispositifs de détection et que en réalité la porte n'est pas ouverte.

Dans le cas d'une ouverture intempestive d'une porte d'un train en mouvement, le système de contrôle doit mettre en sécurité la zone où circule le train. Pour cela, il doit déclencher les actions qui suivent :

- Effectuer un arrêt d'urgence du train concerné.
- Effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone où circule le train concerné.
- Couper la haute tension sur la zone ou circule le train concerné.

A.3.6 Ouverture intempestive d'une porte palière

Dans le cas d'une ouverture intempestive d'une porte palière, le système de contrôle doit déclencher les actions qui suivent :

- Effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone concernée.

A.3.7 Sécurité intrinsèque

Un système est dit en sécurité intrinsèque si on a l'assurance que toute défaillance d'un ou plusieurs composants ne peut le faire passer que dans une situation moins permissive, qui est reconnue sûre, que la situation dans laquelle il se trouve au moment de la défaillance.

A.4 Propriétés fondamentales

Les propriétés que nous incluons ici concernent seulement les échanges d'individus entre les quais et les trains (voyageurs, personnel d'exploitation). Le système de contrôle de météor doit vérifier d'autres propriétés qui ne sont pas présentées ici.

Le système de contrôle doit vérifier les propriétés qui suivent :

A.4.1 Propriétés concernant le transfert de voyageurs

Propriété A.1. *Lors d'un transfert de voyageurs, les portes du train ne peuvent être ouvertes que si l'autorisation correspondante a été donnée par la fonction de contrôle sécuritaire.*

Propriété A.2. *Lors d'un transfert de voyageurs, les portes palières du quai ne peuvent être ouvertes que si l'autorisation correspondante a été donnée par la fonction de contrôle sécuritaire.*

Propriété A.3. *Lors d'un transfert de voyageurs, le redémarrage d'un train ne peut être effectué que si l'autorisation correspondante a été donnée par la fonction de contrôle sécuritaire.*

A.4.2 Propriétés concernant la fonction de contrôle sécuritaire

Propriété A.4. *Lors d'un transfert de voyageurs, l'autorisation d'ouverture des portes d'un côté d'un train ne peut être donnée que si les conditions qui suivent sont vérifiées :*

Condition A.1. Le train est immobilisé (voir section A.3.2, page 136, Autorisation d'ouverture des portes d'un train).

Condition A.2. Le train est positionné à quai, de manière telle que toutes ses portes soient en face de portes palières ou de portes de secours.

Condition A.3. Les portes sont localisées du côté quai.

Propriété A.5. *Lors d'un transfert de voyageurs, l'autorisation d'ouverture des portes palières qui sont en face des portes du train ne peut être donnée que si les conditions qui suivent sont vérifiées :*

Condition A.4. Le train est immobilisé (voir section A.3.2, page 136).

Condition A.5. Le train est positionné à quai, de manière telle que toutes ses portes soient en face de portes palières.

Condition A.6. L'autorisation d'ouverture des portes du train a été donnée pour les portes du coté quai.

Propriété A.6. *Lors d'un transfert de voyageurs, l'autorisation de redémarrage d'un train ne peut être donnée que si les conditions qui suivent sont vérifiées :*

Condition A.7. Toutes les portes du train sont verrouillées, et il y a absence d'autorisation d'ouverture de portes du train.

Condition A.8. Toutes les portes palières sont verrouillées, et il y a absence d'autorisation d'ouverture de portes palières.

A.4.3 Propriétés concernant l'évacuation d'urgence et l'actionnement d'une poignée de déverrouillage dans un train.

Propriété A.7. *Lorsqu'une poignée de déverrouillage a été actionnée, que le train concerné est en inter-station et que le système de contrôle détermine l'impossibilité du train pour atteindre la prochaine station, le système de contrôle doit déclencher les actions qui suivent :*

Condition A.9. Effectuer un arrêt d'urgence du train concerné.

Condition A.10. Effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone où circule le train concerné.

Condition A.11. Couper la haute tension sur la zone ou circule le train concerné.

Le constat d'arrêt du train en inter-station détermine l'impossibilité du train pour atteindre la prochaine station.

Propriété A.8. *Lorsqu'un train est arrêté, l'actionnement d'une poignée de déverrouillage déverrouille toutes les portes du train localisées du coté cheminement. Seule la porte dont la poignée a été tirée est déverrouillée en sécurité intrinsèque.*

Propriété A.9. *Lorsqu'un train est en mouvement, l'actionnement d'une poignée de déverrouillage ne déverrouille aucune porte du train.*

Propriété A.10. *Lorsqu'un train est en mouvement, l'actionnement d'une poignée de déverrouillage déclenche l'arrêt du train à la prochaine station, si le train est en mesure de l'atteindre, qu'elle soit à desservir ou non. Dans ce cas, un transfert de voyageurs dans cette station n'est pas interdit par le système de contrôle.*

A.4.4 Propriétés concernant l'actionnement de la poignée de déverrouillage de service.

Propriété A.11. *Lorsqu'un train est arrêté, l'actionnement d'une poignée de déverrouillage de service déverrouille la porte concernée quelque soit sa localisation.*

Propriété A.12. *Lorsqu'un train est en mouvement, l'actionnement d'une poignée de déverrouillage de service déclenche l'arrêt d'urgence du train.*

Propriété A.13. *Lorsqu'un train est en mouvement, l'actionnement d'une poignée de déverrouillage de service ne déverrouille aucune porte du train.*

A.4.5 Propriétés concernant une panne du système de contrôle.

Propriété A.14. *En cas de panne du système de contrôle les portes du train verrouillées lors de l'apparition de la panne restent verrouillées. Cependant, l'actionnement d'une poignée de déverrouillage aura pour effet immédiat le déverrouillage et la libération de la porte concernée que le train soit arrêté ou en mouvement.*

Propriété A.15. *En absence d'énergie dans un train les portes du train verrouillées lors de la coupure restent verrouillées. Cependant, l'actionnement d'une poignée de déverrouillage aura pour effet immédiat le déverrouillage et la libération de la porte concernée que le train soit arrêté ou en mouvement.*

A.4.6 Propriétés concernant une ouverture intempestive d'une porte d'un train en mouvement.

Propriété A.16. *Dans le cas d'une ouverture intempestive d'une porte d'un train en mouvement (voir section A.3.5, page 139), le système de contrôle doit déclencher les actions qui suivent :*

Condition A.12. Effectuer un arrêt d'urgence du train concerné.

Condition A.13. Effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone où circule le train concerné.

Condition A.14. Couper la haute tension sur la zone ou circule le train concerné.

A.4.7 Propriétés concernant une ouverture intempestive d'une porte palière.

Propriété A.17. *Dans le cas d'une ouverture intempestive d'une porte palière (voir section A.3.6, page 139), le système de contrôle doit effectuer un arrêt d'urgence de tous les trains se trouvant dans la zone du quai concerné. Par ailleurs, l'accès de nouveau trains dans la zone est aussi interdit.*

A.4.8 Procédure d'intervention sur site du personnel d'exploitation lors de l'évacuation d'un train en inter-station.

Propriété A.18. *Lorsqu'un train est évacué en inter-station, le personnel d'exploitation doit suivre la procédure décrite ci-après afin de rétablir le service.*

Procédure A.18.1. Deux agents d'intervention de ligne (AIL) doivent se rendre aux stations (avant et après) entre lesquelles se trouve le train évacué.

Le premier AIL pénètre dans la voie par la station la plus éloignée du train évacué et contrôle que la zone entre la station et le train est vide. S'il rencontre des passagers, il les conduit vers le train. Arrivé au train, il fait le point avec le deuxième AIL, lui remet ses passagers, et monte dans le train.

Le deuxième AIL pénètre dans la voie par la station la plus proche du train évacué et s'il rencontre des passagers, il les oriente vers le quai de la station. L'agent de station accueille les passagers et contrôle qu'aucun d'entre eux ne s'aventure plus loin dans le tunnel. Arrivé au train, l'AIL 2 fait un point avec l'AIL 1, prend en charge l'ensemble de passagers restant et les rapatrie vers la station la plus proche. Arrivé en station, il indique à l'AIL 1 que la voie est vide.

L'AIL 1, une fois dans le train, a pour mission de refermer les portes et de re verrouiller les poignées de toutes les portes du train (action manuelle). Puis, il indique au PCC la fin de son action.

Le PCC peut alors, sous réserve d'accord explicite des 2 AILs, remettre en service le système.

A.5 Premier niveau de détail

Cette section introduit un premier niveau de détail sur le comportement attendu du système de contrôle de météor.

A.5.1 Autres notions fondamentales

Mode de conduite d'un train

En conditions normales d'exploitation, un train qui circule sur la ligne météor est conduit automatiquement par le système de contrôle. Cependant, afin de pouvoir traiter les anomalies qui peuvent survenir, un mode de conduite manuel est prévu. Dans ce mode de conduite, un opérateur prend le contrôle total du train. Quand un train est conduit par le système de contrôle nous dirons qu'il est en conduite automatique (CA). Quand un train est conduit par un opérateur nous dirons qu'il est en conduite manuelle (CM).

Mode de contrôle d'un train

Les trains qui sont contrôlés par le système SAET peuvent se trouver soit en circulation sur les voies soit arrêtés dans une zone de garage. Quand un train se

trouve sur une voie (qu'il soit arrêté ou en mouvement), ou bien quand il est en mouvement dans une zone de garage nous dirons qu'il est en mode de contrôle ligne (CL). Quand un train est arrêté dans une zone de garage ou de service, nous dirons qu'il est en mode de contrôle garage (CG).

Localisation d'un train

Les trains qui circulent sur la ligne météor doivent, en principe, être en mesure de déterminer le lieu où ils se trouvent sur la ligne, et d'en informer le poste de contrôle centralisé. Si c'est le cas, nous dirons que le train est localisé. Au contraire, si un train n'est pas en mesure d'envoyer cette information vers le PCC, nous dirons qu'il est délocalisé.

Train muet

Lorsqu'un train n'est plus en mesure de communiquer avec le PCC (envoyer ou recevoir des informations), nous dirons qu'il est muet. Dans ce cas, le système doit déclencher une alarme train muet pour ce train.

Train endormi

Lorsqu'un train est arrêté dans une zone de garage et qu'il ne communique plus avec le système de contrôle nous dirons qu'il est endormi. Un train endormi peut être réveillé par le système de contrôle a tout moment, cependant, ce réveil n'est pas instantané.

Demande d'évacuation d'un train

Lorsque des situations dangereuses se présentent, le système de contrôle peut décider d'effectuer l'évacuation d'un train. Dans certains cas une période de temps s'écoule entre la décision d'évacuer un train et la réalisation de l'évacuation. Dans ce cas, le système mémorise une demande d'évacuation pour le train concerné.

Alarme évacuation d'un train

Lorsque le processus d'évacuation d'un train est en cours, le système déclenche une alarme évacuation pour ce train. Dans ce cas, des signaux d'alarme seront émis vers le personnel d'exploitation.

Côté cheminement d'un train

Les portes d'un train sont disposées sur ses deux côtés. A tout moment, quelque soit la position d'un train localisé, le système de contrôle peut identifier le côté qui serait utilisé soit pour un transfert de voyageurs, soit pour une évacuation d'urgence, soit pour un transfert de service. Nous appellerons ce

côté du train " côté cheminement ". Nous appellerons l'autre côté " côté non cheminement ".

Dans certaines situations, il est possible que les deux cotés du train soient de cheminement.

Zone ligne et zone garage

Certaines voies ou parties de voie de météor sont utilisées pour garer des trains. Lorsqu'une partie d'une voie est utilisée pour garer des trains, nous dirons que cette partie est une zone garage. Lorsqu'une partie d'une voie est utilisée pour permettre la circulation des trains, nous dirons qu'elle est une zone ligne.

Le système de contrôle connaît en permanence le type de zone de chaque partie de chaque voie.

Commande et autorisation d'une action sécuritaire

Les actions susceptibles d'être réalisées par le système de contrôle de météor sont de deux types : sécuritaires et non sécuritaires.

Avant de déclencher une action sécuritaire, deux conditions doivent être réunies :

- Une commande de l'action doit être en cours. Ceci correspond à un état du système, puisque la commande d'une action peut être en cours pendant une période de temps sans que celle-ci soit réalisée.
  Le fait de commander une action manifeste la volonté de la réaliser.
- Une autorisation de réalisation de l'action est en cours. Cette autorisation est donnée par la fonction de contrôle sécuritaire. Ceci correspond aussi à un état du système, puisque l'autorisation de réalisation d'une action peut être en cours pendant une période de temps sans que celle-ci soit réalisée, ou même après la réalisation de l'action. Par exemple, l'autorisation d'ouverture des portes d'un train reste en cours après l'ouverture effective des portes.
  L'autorisation de réalisation d'une action manifeste la présence des conditions de réalisation de l'action en toute sécurité.

Actions sécuritaires

La réalisation effective d'une action sécuritaire nécessite au préalable une commande de l'action et l'autorisation correspondante (voir paragraphe précédant).

En conditionnes normales de fonctionnement, les actions sécuritaires sont celles liées à l'ouverture et à la fermeture des portes du train et des portes palières, et celles liées à l'arrêt et à la mise en mouvement des trains.

Longueur des trains

Deux longueurs de trains sont susceptible de circuler sur la ligne météor :

- Trais comportant 6 voitures.
- Trains comportant 8 voitures.

Dérive d'un train

Lorsqu'un train doit être à l'arrêt, que le système de contrôle a généré une interdiction de mise en mouvement pour ce train, et que néanmoins le système de contrôle détecte une mise en mouvement du train, nous dirons que le train dérive.

Mode d'exploitation manuel et mode d'exploitation maintenance d'un quai

En conditions normales de fonctionnement, les portes d'un quai sont commandées par le système de contrôle de météor. Cependant, un agent d'exploitation de ligne peut interrompre le contrôle automatique de ces portes à l'aide des commandes qui sont à leur disposition depuis le PMS. Dans ce cas, le contrôle des portes du quai peut basculer en mode d'exploitation manuel ou en mode d'exploitation maintenance.

A.5.2 Propriétés concernant l'évacuation d'urgence et l'actionnement d'une poignée de déverrouillage dans un train

Actionnement d'une poignée de déverrouillage

Les propriétés qui suivent concernent l'actionnement d'une poignée de déverrouillage dans un train.

Propriété A.19. *Lorsqu'une poignée de déverrouillage est actionnée dans un train se trouvant à vitesse non nulle, le système de contrôle doit :*

Condition A.15. Mémoriser une demande d'évacuation pour le train concerné.

Condition A.16. Déclarer infranchissable par ce train la prochaine station que celle-ci soit ou non à desservir par le train au cours de sa mission.

Propriété A.20. *Lorsqu'une poignée de déverrouillage est actionnée dans un train se trouvant à vitesse nulle et localisé, le système de contrôle doit libérer toutes les portes du côté cheminement. La porte associée à la poignée actionnée est déverrouillée en sécurité intrinsèque si celle-ci est localisée du coté cheminement.*

Propriété A.21. *Lorsqu'une poignée de déverrouillage est actionnée dans un train se trouvant à vitesse nulle et délocalisé, le système de contrôle doit libérer toutes les portes du train.*

Propriété A.22. *Lorsqu'une poignée de déverrouillage est actionnée dans un train qui ne se trouve pas en station, que cette poignée concerne une porte du côté cheminement, et qu'il y a perte du contrôle de verrouillage ou de fermeture des portes, le système de contrôle doit déclencher une alarme évacuation pour ce train.*

Propriété A.23. *Lorsqu'une poignée de déverrouillage est actionnée dans un train, que cette poignée concerne une porte du côté non cheminement, et qu'il y a perte du contrôle de verrouillage ou de fermeture des portes, le système de contrôle doit déclencher une alarme évacuation pour ce train.*

Réarmement des poignées de déverrouillage

Les propriétés qui suivent concernent le réarmement des poignée de déverrouillage. Nous considérons qu'une poignée est réarmée quand elle a été tirée est qu'ensuite elle est réarmée.

Propriété A.24. *Lorsqu'une poignée de déverrouillage est actionnée, le système de contrôle doit commander le réarmement des poignées de déverrouillage du train quand celui-ci n'est ni à vitesse nulle ($V>0$) ni en alarme évacuation.*

Propriété A.25. *Le système de contrôle doit commander le réarmement des poignées de déverrouillage d'un train lorsqu'une commande de réarmement des poignées est envoyée par l'opérateur depuis le PCC.*

Propriété A.26. *Après réarmement des poignées de déverrouillage d'un train arrêté suite à une commande de réarmement des poignées envoyée par l'opérateur depuis le PCC, le système de contrôle doit autoriser la fermeture des portes du train. Cette autorisation doit être interrompue si à nouveau une poignée de déverrouillage est actionnée.*

Propriété A.27. *Après réarmement des poignées de déverrouillage d'un train arrêté, le système de contrôle doit vérifier que toutes les portes du train sont fermées et verrouillées, puis il doit lever l'alarme évacuation pour ce train.*

Propriété A.28. *Lorsque toutes les poignées de déverrouillage d'un train sont armées et que ce train se trouve à l'arrêt sur une place de garage, le système de contrôle doit :*

Condition A.17. Effacer toute demande d'évacuation mémorisée pour le train concerné.

Condition A.18. Déclarer franchissable par ce train la prochaine station.

Propriété A.29. *Lorsque toutes les poignées de déverrouillage d'un train sont armées, que ce train se trouve à l'arrêt sur une station et qu'un acquittement de la demande d'évacuation est présent, le système de contrôle doit :*

Condition A.19. Effacer toute demande d'évacuation mémorisée pour le train concerné.

Condition A.20. Déclarer franchissable par ce train la prochaine station.

Basculement d'un train en conduite manuelle

Les propriétés qui suivent concernent le basculement d'un train en conduite manuelle.

Propriété A.30. *Lorsqu'un train bascule en conduite manuelle, le système de contrôle doit :*

Condition A.21. Effacer toute demande d'évacuation mémorisée pour le train concerné.

Condition A.22. Lever l'alarme évacuation quelque soit l'état des portes et des poignées de déverrouillage.

Condition A.23. Déclarer franchissable par ce train la prochaine station.

Train muet

Les propriétés qui suivent concernent les actions à effectuer lorsqu'un train devient muet.

Propriété A.31. *Lorsqu'un train se trouve en mode de contrôle ligne et qu'il devient muet, le système de contrôle doit déclencher une alarme évacuation et une alarme train muet pour ce train.*

Propriété A.32. *Lorsqu'un train est muet et que l'opérateur a demandé la levée de l'alarme train muet, le système de contrôle doit interrompre tout contrôle sur ce train. Dans ce cas le train est effacé du système de contrôle.*

Etat d'alarme évacuation

Les propriétés qui suivent concernent les actions à effectuer lors de l'évacuation d'un train.

Propriété A.33. *Lorsqu'un train est arrêté en alarme évacuation, et qu'il y a une commande opérateur de déverrouillage des portes côté non cheminement, le système de contrôle doit déverrouiller les portes côté non cheminement.*

A.5.3 Propriétés concernant le côté cheminement d'un train

Propriété A.34. *Le système de contrôle doit en permanence connaître le côté cheminement d'un train localisé et réveillé.*

Propriété A.35. *Lors de la réalisation de tests, le système de contrôle doit en permanence connaître le côté cheminement du train sur lequel sont réalisé ces tests.*

Propriété A.36. *Lorsqu'un train devient délocalisé, le système de contrôle doit maintenir le côté cheminement en cours lors de l'apparition de la délocalisation du train.*

Propriété A.37. *Pendant la période d'initialisation d'un train, le côté cheminement de ce train est indéterminé.*

A.5.4 Propriétés concernant les états où des actions sont commandées

Commande d'ouverture des portes d'un train et des portes palières

Propriété A.38. *Le système de contrôle doit commander l'ouverture des portes d'un train en conduite automatique et des portes palières d'un quai d'une station quand au moins une des conditions suivantes est vérifiée :*

Condition A.24. Le train s'arrête dans la zone d'ouverture des portes d'une station à desservir au cours de sa mission.

Condition A.25. Le train est à l'arrêt bien positionné dans une station qui devient à desservir suite à un changement de mission.

Condition A.26. Le train est à l'arrêt bien positionné dans une station, et l'opérateur commande l'ouverture des portes.

Propriété A.39. *Le système de contrôle doit avertir l'opérateur quand toutes les portes palières commandées à l'ouverture ne se sont pas ouvertes.*

Propriété A.40. *A la réception d'une commande opérateur d'ouverture des portes d'un train, le système de contrôle doit commander l'ouverture des portes du train si les conditions suivantes sont vérifiées :*

Condition A.27. Le train est en conduite automatique.

Condition A.28. Le train est à l'arrêt.

Condition A.29. Le train est localisé dans une zone exploitée en mode garage.

Commande de la séquence de départ d'un train et de la fermeture des portes

Propriété A.41. *A la réception du signal de fin de stationnement (ce qui constitue un événement externe), le système de contrôle doit commander la séquence de départ du train, et la fermeture des portes palières si les conditions suivantes sont vérifiées :*

Condition A.30. Le train est en conduite automatique.

Condition A.31. Aucune demande d'évacuation n'est mémorisée.

Condition A.32. L'opérateur du PCC n'a pas forcé l'ouverture des portes du train, ou il l'a fait mais il a annulée ultérieurement cette commande.

Condition A.33. Il n'y a pas de cause de blocage du train à quai.

Propriété A.42. *A la réception d'une commande opérateur de verrouillage des portes du train, le système de contrôle doit commander la fermeture des portes du train si les conditions suivantes sont vérifiées :*

Condition A.34. Le train est en conduite automatique.

Condition A.35. Les portes du train ne sont pas libres.

Condition A.36. Si le train est en station, il n'y a pas de commande d'ouverture de ses portes en cours.

En plus, si le train est arrêté à quai, le système de contrôle doit commander de manière synchronisé la fermeture des portes palières.

Propriété A.43. *En station, à la réception d'une commande opérateur d'annulation d'ouverture des portes d'un train, le système de contrôle doit commander la fermeture des portes du train.*

A.5.5 Propriétés concernant les autorisations délivrées par la fonction de contrôle sécuritaire

Autorisation d'ouverture et de fermeture des portes d'un train

Propriété A.44. *Le système de contrôle est en mesure de contrôler chaque porte d'un train d'une façon indépendante. Par conséquent, à tout moment, la fonction de contrôle sécuritaire doit délivrer pour chaque porte soit une autorisation d'ouverture soit une autorisation de fermeture.*

Propriété A.45. *En aucun cas, le système ne peut se trouver dans un état où il y ait autorisation d'ouverture et de fermeture d'une porte d'un train simultanément.*

Propriété A.46. *En aucun cas, le système ne peut se trouver dans un état où il y ait autorisation d'ouverture des portes du train et autorisation de mise en mouvement simultanément.*

Propriété A.47. *Lorsqu'un train est en conduite automatique et en mouvement ($V > 0$), aucune autorisation d'ouverture de porte ne peut être délivrée par le système de contrôle.*

Propriété A.48. *Lorsqu'un train est en conduite automatique et arrêté ($V = 0$), le système de contrôle ne peut autoriser l'ouverture de portes que pour celles localisées du coté cheminement.*

Propriété A.49. *En cas de panne du système de contrôle ou en absence d'énergie dans un train, les portes du train doivent être libres. Dans ce cas, Il doit être possible d'ouvrir les portes du train de l'intérieur à l'aide d'une poignée de déverrouillage.*

Propriété A.50. *Lorsqu'un train est en conduite automatique, et qu'une ouverture anormale des portes du train se présente, le système doit informer l'opérateur du PCC. L'ouverture anormale d'une porte d'un train se présente quand la porte s'ouvre et qu'aucune des conditions suivantes n'est vérifiée :*

Condition A.37. Lors d'un transfert voyageur, le train est bien positionné a quai et seul les portes du côté cheminement sont ouvertes.

Condition A.38. Le train est en situation d'évacuation.

Condition A.39. Lors d'un transfert de service, seul la porte du côté cheminement qui correspond à la poignée de service actionnée est ouverte.

Condition A.40. Lors des tests.

Autorisation de mise en mouvement d'un train

Propriété A.51. *Pour chaque train en conduite automatique et arrêté, le système de contrôle doit générer soit une autorisation de mise en mouvement du train, soit une interdiction de mise en mouvement. En aucun cas, le système ne peut se trouver dans un état où il y ait simultanément autorisation de mise en mouvement et interdiction de mise en mouvement d'un train.*

Autorisation d'ouverture des portes palières

Propriété A.52. *Lorsqu'un train est en conduite automatique, à vitesse nulle (V=0), qu'il est bien positionné en face des portes palières, et qu'un transfert de voyageurs doit avoir lieu, le système de contrôle doit générer une autorisation d'ouverture des portes palières du quai qui sont en face des portes du train.*

A.5.6 Propriétés concernant la réalisation effective d'actions sur les portes palières

Ouverture des portes palières

Propriété A.53. *Lorsqu'une commande d'ouverture de portes palières pour un train de 6 voitures est générée par le système de contrôle, celui-ci doit déverrouiller et ouvrir les portes palières du quai concerné qui sont en face des portes du train.*

Propriété A.54. *Lorsqu'une commande d'ouverture de portes palières pour un train de 8 voitures est générée par le système de contrôle, celui-ci doit déverrouiller et ouvrir toutes les portes palières du quai concerné.*

Propriété A.55. *Lorsque les portes palières se trouvent en mode manuel ou en mode maintenance et que un agent d'exploitation commande l'ouverture de celles-ci, le système de contrôle doit déverrouiller et ouvrir toutes les portes palières du quai concerné. Il doit, par ailleurs, informer l'opérateur du PCC de cette ouverture.*

Propriété A.56. *Les portes palières peuvent être déverrouillées dès l'intérieur du train.*

Fermeture des portes palières

Propriété A.57. *Lorsqu'une commande de fermeture de portes palières est générée par le système de contrôle et qu'aucune de portes n'a été déverrouillée manuellement, celui-ci doit fermer puis verrouiller toutes les portes palières du quai concerné*

Propriété A.58. *Lorsqu'une commande de fermeture de portes palières est envoyée par l'opérateur du PCC, le système de contrôle doit fermer puis verrouiller toutes les portes palières du quai concerné*

Propriété A.59. *Lorsqu'une commande de fermeture de portes palières est envoyée par un agent d'exploitation à l'aide du PMS et qu'aucune demande d'ouverture des portes palières n'est en cours, le système de contrôle doit fermer puis verrouiller toutes les portes palières du quai concerné.*

Propriété A.60. *Si lors du processus de fermeture d'une porte palière celle-ci se bloque, le système doit maintenir l'effort de fermeture pendant une durée X de temps. Passée cette période, si la porte est toujours bloquée, le système doit libérer la porte pendant une durée Y de temps, puis rappliquer l'effort de fermeture. Ce cycle doit être répété tant que la porte reste bloquée et qu'aucune action de l'opérateur du PCC n'est réalisée. Après l'échec de la deuxième tentative de fermeture, le système doit informer du phénomène l'opérateur du PCC.*

Propriété A.61. *Lorsque le système ferme une porte palière, il doit tout de suite après la verrouiller.*

A.5.7 Propriétés concernant le contrôle de l'état des portes localisées sur un quai

Propriété A.62. *Lorsqu'une porte palière s'ouvre, que cette ouverture n'est pas la conséquence d'une commande provenant du PMS, et qu'il n'y a pas de train arrêté en face du quai, le système de contrôle doit générer une alarme porte palière.*

Propriété A.63. *Lorsqu'une porte de secours s'ouvre, le système de contrôle doit générer une alarme porte de secours.*

Propriété A.64. *Lorsqu'un portillon d'extrémité de quai s'ouvre, et qu'il n'y a pas d'inhibition du contrôle des portillons en cours, le système de contrôle doit générer une alarme portillon.*

Propriété A.65. *Lorsqu'un commande d'inhibition du contrôle des portillons est envoyée par l'opérateur du PCC, le système de contrôle doit inhiber le contrôle des portillons pour le quai concerné. Cette inhibition doit être indiquée au moyen d'un voyant lumineux localisé à côté du portillon. L'inhibition peut être annulée.*

Propriété A.66. *Lorsque toutes les portes palières, toutes les portes de secours et les deux portillons d'extrémité de quai sont fermés, le système de contrôle doit allumer un voyant localisé en sortie de station. Ce signal lumineux est destiné à un éventuel conducteur.*

Propriété A.67. *Lorsqu'un quai est en mode d'exploitation manuel ou en mode d'exploitation maintenance, le système de contrôle doit inhiber le contrôle de fermeture des portes palières, des portes de secours, et des portillons d'extrémité de quai. Il doit aussi informer l'opérateur du PCC de cette inhibition.*

Propriété A.68. *Lorsqu'un quai est en mode d'exploitation manuel ou en mode d'exploitation maintenance, et qu'un agent d'exploitation autorise un train en conduite manuel à partir, le système de contrôle doit informer le conducteur au moyen d'un voyant en sortie de station qu'il est autorisé à partir.*

A.6 Découpage du système de contrôle

Le système de contrôle de météor est découpé en quatre parties, suivant les équipements utilisés dans son implantation :

- Pilote Automatique Embarqué (PAE).
- Pilote Automatique Section (PAS).
- Poste de Contrôle Centralisé (PCC).
- Façade de Quai (FQ).

A.6.1 Pilote Automatique Embarqué

Chaque train qui circule sur la ligne Météor est équipé d'un ensemble d'équipements qui font partie du système de contrôle. Ces équipements constituent le support du logiciel embarqué dans le train. Nous appellerons cet ensemble d'équipements plus le logiciel correspondant "pilote automatique embarqué". Le PAE communique avec les autres sous-systèmes qui forment le SAET.

Par ailleurs, le PAE communique avec un autre système indépendant du SAET. Nous appellerons ce système Matériel Roulant (MR).

A.6.2 Pilote Automatique Section

Les voies de la ligne Météor sont équipés d'ensemble d'équipements qui font partie du système de contrôle. Ces équipements constituent le support de logiciel qui permet de contrôler les segments de voie. Nous appellerons cet ensemble d'équipements plus le logiciel correspondant "pilote automatique section". Le PAS communique avec les autres sous-systèmes qui forment le SAET.

A.6.3 Poste de Contrôle Centralisé

Le PCC est équipé d'ensemble d'équipements qui font partie du système de contrôle. Ces équipements constituent le support de logiciel de contrôle centralisé. Le PCC communique avec les autres sous-systèmes qui forment le SAET.

A.6.4 Façade de Quai

Chaque façade de quai est équipé d'un ensemble d'équipements qui font partie du système de contrôle. Ces équipements constituent le support du logiciel

qui contrôle les éléments du quai. Nous appellerons cet ensemble d'équipements plus le logiciel correspondant "façade de quai". Le FQ communique avec le PAS.

A.6.5 Définition des interfaces entre les différentes sous-systèmes

Les différents sous-systèmes du SAET communiquent au moyen des interfaces qui suivent. Ces informations sont classées en deux groupes : sécuritaires et non sécuritaires. Les informations sécuritaires sont celles qui ne peuvent pas être affectées par un dysfonctionnement du système.

Le MR communique au PAE :

- KSA (sécuritaire) : Les poignées de déverrouillage de portes actionnées dans le train.
- KSE (sécuritaire) : Les poignées de déverrouillage de service actionnées dans le train.
- KVF (sécuritaire) : Contrôle verrouillage fermeture.
- Le mode de conduite du train (sécuritaire) : Trois états sont possibles : CA, CM, Indéterminé.

Le PAE communique au MR :

- AuTr (sécuritaire) : L'autorisation de mise en mouvement du train.
- FU (sécuritaire) : La demande de freinage d'urgence.
- AuO A(sécuritaire) : L'autorisation d'ouverture des portes côté A.
- AuO B(sécuritaire) : L'autorisation d'ouverture des portes côté B.
- CDEPT (sécuritaire) : Commande ouverture de portes.
- Réarmement KSA (sécuritaire) : La demande de réarmement des poignées de déverrouillage.
- Auf A(sécuritaire) : L'autorisation de fermeture des portes côté A.
- Auf B(sécuritaire) : L'autorisation de fermeture de portes côté B.
- Le commande de fermeture de portes.
- Le côté service du train.

Le PAS communique au PAE :

- AuDem (sécuritaire) : L'autorisation de démarrage du train. Une par quai.

- Le Mode d'exploitation de chaque zone (sécuritaire).
- AuM-Quai (sécuritaire) : Autorisation de mouvement du train.
- AuM (sécuritaire) : Autorisation de mouvement du train sur les zones.
- Le Top départ pour le train.

Le PAE communique au PAS :

- AuO-PP (sécuritaire) : L'autorisation d'ouverture des portes palières.
- La localisation du train (sécuritaire).
- Vnulle (sécuritaire) : Train a vitesse nulle. C'est un booléen.
- ModeConduite (sécuritaire) : Le Mode de conduite du train.
- TE à Quai (sécuritaire).
- Blocage à Quai.
- DdeO-PP. La demande d'ouverture des portes palières.

Le PAS communique au FQ :

- CdeOuvPP (sécuritaire) : La commande d'ouverture de portes palières.
- CdeFermPP : La commande de fermeture de portes palières.
- La commande d'allumage du voyant d'autorisation du départ.

Le PCC communique au PAE et au PAS :

- La Télécommande de réarmement des poignées de déverrouillage (sécuritaire).
- L'acquittement de la demande d'évacuation (sécuritaire).
- La Télécommande de libération des portes côté opposé au cheminement.

Le FQ communique au PAS :

- KV-PP (sécuritaire) : Contrôle de verrouillage des portes palières.
- KV-PEQ (sécuritaire) : Contrôle de verrouillage des portes d'extrémité de quai.
- KV-PS (sécuritaire) : Contrôle de verrouillage des portes de secours.
- Mode-PMS (sécuritaire) : Mode pupitre manuel de station.
- AuDemarrage Manuel (sécuritaire) : Autorisation de départ du train.
- KO-PP (sécuritaire) : Contrôle d'ouverture des portes palières.

Annexe B

Cas d'Etude : Spécification Formelle

B.1 Analyse du système

Dans cette section, nous allons d'abord identifier les notions fondamentales contenues dans la spécification informelle et nous allons les classer par niveau d'abstraction. Ensuite, nous allons classer les propriétés du système par niveau d'abstraction.

B.1.1 Identification des notions fondamentales

La section A.2, page 130, décrit les éléments qui forment la ligne. Cette description permet de relever les notions qui suivent que nous plaçons dans un premier niveau car elles correspondent aux objets les plus abstrait du système :

- *Voie*
- *Train*
- *Quai de station*

D'autres notions moins générales sont présentées dans cette section. Nous les plaçons dans un deuxième niveau car nous pensons qu'elles seront introduites dans les premiers raffinements :

- *Zone d'évacuation.* Une voie est découpée en zones. Il existe plusieurs critères de découpage en zones. Les zones dont nous parlons ici résultent du découpage effectué afin de gérer les évacuations d'urgence.
- *Poignée de déverrouillage.* Un train comporte plusieurs poignées de déverrouillage. Cette notion est nécessaire pour exprimer les propriétés associées à l'évacuation d'urgence.

D'autres notions sont présentées dans la section A.3, page 135. Nous les plaçons dans le troisième niveau car elles permettent d'exprimer des détails du système :

- *Voiture.* Un train est formé de plusieurs voitures.
- *Porte de train.* Un train comporte plusieurs portes.
- *Poignée de service.* Un train comporte plusieurs poignées de déverrouillage de service.
- *Bouton d'alarme de train.* Un train comporte plusieurs boutons d'alarme.
- *Mission d'un train.* Chaque train a une mission associée.
- *Porte palière.* Un quai comporte plusieurs portes palières.
- *Portillon d'extrémité de quai.*
- *Porte de secours.* Un quai comporte plusieurs portes de secours.
- *Bouton d'alarme à quai.*
- *Pupitre manuel de station.*

B.1.2 Classement des propriétés fondamentales

La section A.1, page 129, présente les principes fondamentaux de sécurité. Notre premier groupe contient ces quatre principes.

Les propriétés 1, 2 et 3 de la section A.4, page 139, introduisent les notions d'autorisation d'ouverture des portes et de démarrage des trains. Nous considérons qu'elles correspondent à une décision d'implantation. Par conséquent, nous retarderons leur introduction jusqu'à l'introduction des détails du système.

Les propriétés 4,5,6 précisent le **comportement nominal du système**. Elles décrivent les conditions d'ouverture et de fermeture des portes, et les conditions de démarrage des trains. Nous les plaçons dans le deuxième groupe.

Les propriétés 7 à 12 et 17 précisent le **comportement du système lorsque des situations dangereuses apparaissent**. Elles décrivent le processus d'évacuation d'urgence et la problématique associée aux poignées de déverrouillage. Nous les plaçons dans le troisième groupe.

Les propriétés 13 à 16 précisent le **comportement dégradé du système**. Elles décrivent les situations anormales et les pannes du système. Nous les plaçons dans le quatrième groupe.

B.2 Modélisation des principes fondamentaux

Dans la section A.1, page 129, de la spécification informelle ont été décrits les principes sur lesquels a été conçue la protection des voyageurs. Le but de ce premier modèle est de les exprimer formellement.

B.2.1 Analyse de propriétés

Premier Principe (A.1)

Les voyageurs se trouvant à bord des trains et à quai ne sont protégés que dans la mesure où ces deux univers sont hermétiquement clos vis-à-vis du site dans lequel les trains sont en mouvement.

Ce principe donne la notion fondamentale de sécurité des voyageurs : un voyageur est en sécurité s'il se trouve dans un univers hermétiquement clos ; soit dans un train soit sur un quai. Il est possible d'identifier ici trois autres notions importantes :

- *Train hermétiquement clos.* Un train ouvert est un train qui n'est pas hermétiquement clos.
- *Quai hermétiquement clos.* Un quai ouvert est un quai qui n'est pas hermétiquement clos.
- *Voyageur protégé.*

Le système doit cependant permettre aux personnes d'aborder et de quitter les trains. Le deuxième principe introduit la notion de sécurité dans ce cas.

Deuxième Principe (A.2)

Il est nécessaire d'assurer la protection des voyageurs pendant leur transfert entre le quai et le train lors des arrêts en station, en assurant l'immobilité du train.

Ce principe donne la notion fondamentale de sécurité lors d'un transfert de voyageurs : un voyageur est en sécurité pendant un transfert si le train est immobilisé. Il est possible d'identifier ici deux notions importantes :

- *Transfert de voyageurs.*
- *Train en situation d'immobilité.*

Les deux premiers principes définissent la sécurité des voyageurs en fonctionnement nominal. Néanmoins, des dangers peuvent apparaître à bord des trains ou à quai. Le troisième principe définit la sécurité des voyageurs lorsque des situations dangereuses se présentent à bord d'un train. La sécurité lorsque des situations dangereuses se présentent dans un quai n'est pas definie.

Troisième Principe (A.3)

Les voyageurs se trouvant à bord d'un train ne sont en sécurité que si aucune situation dangereuse n'apparaît à bord ou si, en cas de situation dangereuse (incendie par exemple), ils ont, en dernier recours, la possibilité de quitter le train en urgence, où qu'il soit (évacuation), et de regagner la sortie de secours la plus proche (station), dans un environnement rendu sûr.

Ce principe donne la notion fondamentale de sécurité lorsqu'un danger se présente dans un train : un voyageur est en sécurité si aucune situation dangereuse n'apparaît à bord ou si, en cas de situation dangereuse, le train peut toujours être évacué, et les voyageurs peuvent être conduits en toute sécurité jusqu'à une sortie de secours. Il est possible d'identifier ici quatre autres notions importantes :

- *Train en danger.*
- *Evacuation d'urgence d'un train.*
- *Environnement rendu sûr lors d'une évacuation.* Cette notion est fortement liée à la notion de zone d'évacuation que nous avons présentée plus haut. Il est possible de se déplacer en sécurité sur une zone d'évacuation qui est en sécurité.
- *Possibilité de gagner une sortie de secours.*

Il existe d'autres cas où la sécurité des personnes est en jeux. Ils correspondent à des opérations d'exploitation et de maintenance. Le quatrième principe définit la sécurité dans le cas où le personnel d'exploitation pénètre dans les zones où circulent les trains.

Quatrième Principe (A.4)

D'autre part, le système doit autoriser, avec un minimum de perturbation du trafic, certaines interventions du personnel d'exploitation ou de maintenance. En particulier, la pénétration de personnel habilité dans les zones où les trains circulent doit être possible, avec un maximum de garanties de sécurité.

Ce principe traite la notion fondamentale de sécurité lorsqu'une personne pénètre dans la zone où circulent les trains. Il est possible d'identifier ici deux notions importantes :

- *Possibilité de pénétrer dans la zone où circulent les trains.*
- *Donner des garanties de sécurité lorsqu'une personne pénètre dans cette zone.* Cette notion est fortement liée à la mise en sécurité d'un quai. Il est possible de pénétrer dans la voie par un quai mis en sécurité avec des bonnes garanties de sécurité.

Dans tous ces principes, nous pouvons observer la notion de voyageur ou de personne en sécurité. Ces notions n'apparaîtront pas dans le modèle parce que notre intérêt est de construire un système qui à tout moment garantie la sécurité des personnes qui l'utilisent. Cette sécurité est un but, une conséquence du système.

Lors d'un transfert de voyageurs ou d'une évacuation d'urgence les portes du train et du quai doivent s'ouvrir. Ceci conduit à une situation où les univers dont parle le premier principe ne sont pas hermétiquement clos. L'ouverture des portes pendant un transfert de voyageur correspond à une situation normale du

système. Nous ajoutons les notions suivantes :

- *Train ouvert.* Un train est ouvert si au moins une de ses portes est ouverte.
- *Quai ouvert.* Un quai est ouvert si au moins une de ses portes est ouverte.

B.2.2 Objets utilisés

Pour modéliser les principes fondamentaux, nous introduisons d'abord deux notions de niveau 1 identifiées dans la section (B.1.1) : les trains et les quais.

- tt : Ensemble de trains de la ligne. Ils peuvent être soit en circulation, soit dans une zone de garage.
- qq : Ensemble de quais qui existent sur la ligne.

Le principe 3 utilise la notion de zone d'évacuation en sécurité. Nous introduisons dès le début cette notion.

- ze : Ensemble de zones d'évacuation définies pour la ligne.

Afin de décrire la position des trains et des quais sur la ligne, nous utilisons les trois relations qui suivent :

- t_q : Détermine les localisations des trains vis-à-vis des quais. Lorsqu'un train arrive dans un quai ou lorsqu'un train quitte un quai, cette relation doit être modifiée. Nous supposons qu'un train ne peut se trouver que dans un seul quai à un instant précis. Par ailleurs, un train en circulation, en zone de garage ou arrêté entre deux stations ne se trouve sur aucun quai. Cette relation est donc une injection partielle des trains vers quais :

$$t\_q \in tt \rightarrowtail qq \tag{B.1}$$

- t_z : Détermine les localisations des trains vis-à-vis des zones d'évacuation. Lorsqu'un train arrive dans une zone d'évacuation ou lorsqu'un train quitte une zone d'évacuation, cette relation doit être modifiée. Nous supposons qu'un train se trouve toujours au minimum dans une zone d'évacuation, et qu'il peut se trouver dans plusieurs zones d'évacuation à un instant précis. Par ailleurs, plusieurs trains peuvent se trouver dans la même zone d'évacuation :

$$t\_z \in tt \leftrightarrow ze \wedge dom(t\_z) = tt \tag{B.2}$$

- q_z : Détermine les localisations des quais vis-à-vis des zones d'évacuation. Cette localisation ne change pas. Par conséquent, cette relation est une constante du système. Nous supposons qu'un quai peut appartenir a plusieurs zones d'évacuation. Par ailleurs, une zone d'évacuation peut couvrir plusieurs quais.

$$q\_z \in qq \leftrightarrow ze \tag{B.3}$$

Les trois relations précédentes sont liées par la propriété qui suit : un train à quai se trouve forcément dans une des zones d'évacuation qui couvrent ce quai. La formule correspondante est :

$$t\_q^{-1}; t\_z \subseteq q\_z \tag{B.4}$$

Le premier principe introduit les notions de train hermétiquement clos et de quai hermétiquement clos. Ce principe indique que les passagers à bord d'un train sont protégés si le train est hermétiquement clos. D'un autre côté, il est nécessaire de permettre aux personnes d'aborder et de quitter les trains. Quatre cas sont prévus à cette effet (voir section A.1).

Les objets qui suivent permettent d'introduire ces notions :

- to : Trains ouverts : Un train ouvert n'est pas hermétiquement clos, un train fermé est hermétiquement clos :

$$to \subseteq tt \tag{B.5}$$

- ttr : Trains qui réalisent un des quatre transferts prévus :

$$ttr \subseteq tt \tag{B.6}$$

- ttv : Trains qui réalisent un transfert de voyageurs :

$$ttv \subseteq ttr \tag{B.7}$$

  Il existe une autre propriété importante qui doit être vérifiée par les trains qui réalisent un transfert de voyageurs : ils doivent se trouver à quai. Cette propriété n'est pas issue des principes fondamentaux mais plutôt de la connaissance global du système :

$$ttv \in dom(t\_q) \tag{B.8}$$

- *teu* : Trains qui réalisent une évacuation d'urgence :

$$teu \subseteq ttr \tag{B.9}$$

Les ensembles *ttv*, *teu* sont disjoints puisqu'un train ne peut pas effectuer les deux types de transferts simultanément. Nous ajoutons, en conséquence, la formule qui suit :

$$ttv \cap teu = \emptyset \tag{B.10}$$

Le troisième principe définie la notion de sécurité à bord d'un train qui est dans une situation dangereuse. Le deuxième principe indique qu'un train qui effectue un transfert de voyageurs doit être immobilisé.

Pour introduire ces notions, nous utilisons les objets qui suivent :

- *td* : Trains en danger. Un train en évacuation d'urgence est un train en danger :

$$td \subseteq tt \tag{B.11}$$

$$teu \subseteq td \tag{B.12}$$

- *ti* : Trains immobilisés. Tout train qui effectue un transfert doit être immobilisé :

$$ti \subseteq tt \tag{B.13}$$

$$ttr \subseteq ti \tag{B.14}$$

Nous procédons d'une façon semblable pour introduire les objets qui servent à décrire les états des quais :

- *qo* : Quais ouverts. Un quai ouvert est non hermétiquement clos, un quai fermé est hermétiquement clos :

$$qo \subseteq qq \tag{B.15}$$

- *qtv* : Quais qui effectuent un transfert de voyageurs. Nous considérons que ce sont les quais où il y a un train qui effectue un transfert de voyageurs :

$$qtv = t\_q[ttv] \tag{B.16}$$

- qd : Quais en danger. Par exemple, lorsqu'une porte palière s'ouvre d'une façon intempestive :

$$qd \subseteq qq \tag{B.17}$$

- qs : Quais mis en sécurité :

$$qs \subseteq qq \tag{B.18}$$

Pour terminer, nous introduisons l'objet qui sert à décrire les états des zones d'évacuation :

- zs : Zones d'évacuation qui ont été mises en sécurité. Cette notion est introduite afin de pouvoir écrire les contraintes contenues dans le troisième principe :

$$zs \subseteq ze \tag{B.19}$$

B.2.3 Invariants du système

Les formules de cette section expriment les propriétés statiques du système.

Premier principe

$$to \subseteq ttr \qquad \text{(P 1.1)} \tag{B.20}$$

$$qo \subseteq qtv \cup qs \qquad \text{(P 1.2 et P 4)} \tag{B.21}$$

(B.20) tient compte de la sécurité à bord des trains. Cette formule peut être lue comme suit : un train ouvert effectue un des quatre transferts prévus. Cette formule peut aussi s'écrire :

$$\overline{ttr} \subseteq \overline{to} \tag{B.20a}$$

Ce qui peut être lu comme suit : un train qui ne effectue pas de transfert est toujours hermétiquement clos. Les deux formules sont équivalentes.

(B.21) tient compte de la sécurité sur les quais. La formule peut être lue comme suit : Un quai ouvert qui n'effectue pas de transfert de voyageur est un quai en sécurité. Ceci peut aussi s'écrire :

$$\overline{qtv} \cap \overline{qs} \subseteq \overline{qo} \tag{B.21a}$$

Ce qui peut être lu comme suit : un quai qui ne effectue pas de transfert de voyageurs et qui n'est pas en sécurité est toujours hermétiquement clos. Les deux formules sont équivalentes.

Deuxième principe

Les contraintes contenues dans le deuxième principe sont exprimées dans la formule qui suit :

$$ttv \subseteq ti \qquad \text{(P 2)} \tag{B.22}$$

Cette formule tient compte de l'immobilisation d'un train qui effectue un transfert de voyageurs. La formule dit que tout train qui effectue un transfert de voyageurs est immobilisé. Elle correspond à un cas particulier de (B.14).

Troisième principe

Les contraintes contenues dans le troisième principe sont exprimées dans la formule qui suit :

$$teu \subseteq t\_z^{-1}[zs] \qquad \text{(P 3.1)} \tag{B.23}$$

(B.23) tient compte de la sécurité des passagers lors d'une évacuation d'urgence. Cette formule peut être lue comme suit : un train en évacuation d'urgence se trouve toujours dans une zone d'évacuation mise en sécurité.

Ce principe indique en plus qu'il doit toujours avoir une sortie possible pour les personnes qui se trouvent dans le train évacué. La formule correspondant à cette contrainte est :

$$ran(q\_z) = ze \qquad \text{(P 3.2)} \tag{B.24}$$

Ceci signifie que toute zone d'évacuation possède au moins un quai qui permet l'évacuation du train en toute sécurité.

Quatrième principe

Ce principe est déjà inclus dans la formule (B.21). Elle introduit la notion de personnel habilité qui pénètre dans les zones où les trains circulent avec un maximum de garanties de sécurité. Nous considérons qu'une personne habilitée peut pénétrer dans les zones où circulent les trains si le quai est en sécurité.

B.2.4 Dynamique du système

Dans les sections qui suivent nous présentons les évènements que le système permet d'observer. Nous utilisons le raffinement pour les introduire graduellement.

Nous procédons par niveaux d'abstraction. Pour chaque niveau, nous décrivons d'abord la problématique traitée. Ensuite nous donnons les évènements en les justifiant.

Les objets et formules présentés dans les sections (B.2.2) et (B.2.3) apparaîtront peu à peu suivant le niveau d'abstraction modélisé. Le modèle de plus bas niveaux devra contenir tous ces objets et formules.

A la fin de chaque niveau, nous incluons une présentation des preuves que nous voulons réaliser. Elles permettent de valider les modèles et de montrer des propriétés sur la dynamique du système.

B.2.5 Premier modèle

Le modèle que nous allons construire permet d'observer la ligne toute entière. Il comporte des trains en mouvement, des quais, des transferts de personnes entre les quais et les trains, etc. Nous observons le système dans toute sa globalité.

En principe, le système ne doit permettre l'ouverture de portes que dans les quatre cas prévus d'échange de personnes, voir section A.1, page 129. Cependant, des dysfonctionnements peuvent entraîner la violation de cette propriété ; la ligne devient alors non sûre. L'ouverture d'une porte d'un train en mouvement met en danger les personnes à bord. D'une façon semblable, l'ouverture d'un porte palière d'un quai qui n'effectue pas de transfert met en danger les personnes qui se trouvent sur le quai. Dans ces cas, le système doit réagir rapidement afin de rétablir un environnement sûr.

A ce niveau d'abstraction, nous souhaitons observer seulement l'événement qui permet d'effectuer un transfert de personnes. Ces personnes peuvent soit aborder le train soit le quitter. Tous les cas de transfert sont observés avec cet événement. Son action doit garantir que le train et le quai sont sûrs juste après son observation ; c'est à dire, hermétiquement clos.

A chaque fois que l'événement est observé, il concerne un transfert de personnes pour un train déterminé. Il n'a pas de condition de déclenchement, ce qui signifie qu'il peut être observé à tout moment. Sa garde est :

$$TRUE \tag{B.25}$$

Son action ne modifie pas l'état du système : Avant son observation, les trains et les quais étaient hermétiquement clos ; après, ils restent dans le même état. Son action est :

$$SKIP \tag{B.26}$$

Nous l'appellerons *Transfert*. Avec lui, nous pouvons observer tous les cas de transfert : Transfert de voyageurs, évacuation d'urgence etc.

Il est cependant nécessaire de tenir compte, dès le premier niveau, des situations qui rendent la ligne non sûre. Nous considérons que la ligne est dans un état sûr si tous ses trains et tous ses quais sont sûrs. Dans ce but, nous ajoutons au modèle quatre autres évènements :

- *TrainDanger* : Permet d'observer toutes les situations qui rendent un train non sûr. Il peut être observé à tout moment pour les trains en sûreté. Si $t1$ dénote le train concerné, sa garde est :

$$\exists_{t1} \cdot (t1 \in tt \wedge t1 \notin td) \tag{B.27}$$

Son action rend le train non sûr. Elle peut être écrite comme suit :

$$td := td \cup \{t1\} \tag{B.28}$$

- *QuaiDanger* : Permet d'observer toutes les situations qui rendent un quai non sûr. Il peut être observé à tout moment pour les quais en sûreté. Si $q1$ dénote le train concerné, sa garde est :

$$\exists_{q1} \cdot (q1 \in qq \wedge q1 \notin qd) \tag{B.29}$$

Son action rend le quai non sûr. Elle peut être écrite comme suit :

$$qd := qd \cup \{q1\} \tag{B.30}$$

- *TrainSur* : Permet d'observer la remise dans un état sûr d'un train qui ne l'était pas. Il peut être observé à tout moment mais seulement pour les trains qui se trouvent en danger. Si $t1$ dénote le train concerné, sa garde est :

$$\exists_{t1} \cdot (t1 \in td) \tag{B.31}$$

Son action rend le train sûr. Elle peut être écrite comme suit :

$$td := td - \{t1\} \tag{B.32}$$

- *QuaiSur* : Permet d'observer la remise dans un état sûr d'un quai qui ne l'était pas. Il peut être observé à tout moment mais seulement pour les quais qui se trouvent en danger. Si $q1$ dénote le quai concerné, sa garde est :

$$\exists_{q1} \cdot (q1 \in qd) \tag{B.33}$$

Son action rend le quai sûr. Elle peut être écrite comme suit :

$$qd := qd - \{q1\} \tag{B.34}$$

Les deux derniers évènements sont importants car ils permettent au système d'évoluer correctement. Si nous ne les incluons pas, un train ou un quai qui devient en danger ne pourra plus devenir sûr.

Idéalement, nous voulons construire un système toujours sûr, ce qui nous conduit à vouloir écrire l'invariant suivant :

$$td = \emptyset \quad \wedge \quad qd = \emptyset$$

Ce qui indiquerait que le système ne permet jamais l'ouverture anormale d'un train ou d'un quai, ni aucune situation dangereuse dans un train ou sur un quai. Cependant, nous devons bien admettre que ces cas peuvent se présenter

et que le système doit se défendre lorsqu'ils arrivent. Nous n'incluons donc pas l'invariant précédent dans notre modèle.

Nous considérons par contre q'un train est soit sûr soit en danger, mais pas les deux simultanément. La même propriété est valable pour les quais.

Initialement la ligne est sûre. Par conséquent, nous initialisons le système avec la substitution qui suit :

$$td, qd := \emptyset, \emptyset \tag{B.35}$$

Le système suivant correspond à la spécification complète en B du premier niveau d'abstraction.

SYSTEM	
$Meteor1.1$	
SETS	
tt ;	Trains de la ligne.
qq	Quais de la ligne.
VARIABLES	
td,	Trains en danger.
qd	Quais en danger.
INVARIANT	
$td \subseteq tt \wedge$	(B.11).
$qd \subseteq qq$	(B.17).
INITIALISATION	
$td, qd := \emptyset, \emptyset$	(B.35) ; La ligne est sûre.
EVENTS	
Transfert $\hat{=}$	
SELECT $TRUE$	(B.25) ; Un transfert peut toujours être observé.
THEN $SKIP$	(B.26) ; La sûreté n'est pas affectée.
END ;	
TrainDanger $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin td$	(B.27) ; $t1$ n'est pas en danger.
THEN $td := td \cup \{t1\}$	(B.28) ; $t1$ devient en danger.
END ;	
QuaiDanger $\hat{=}$	
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qd$	(B.29) ; $q1$ n'est pas en danger.
THEN $qd := qd \cup \{q1\}$	(B.30) ; $q1$ devient en danger.
END ;	
TrainSur $\hat{=}$	
ANY $t1$ WHERE $t1 \in td$	(B.31) ; $t1$ est en danger.
THEN $td := td - \{t1\}$	(B.32) ; $t1$ n'est plus en danger.
END ;	
QuaiSur $\hat{=}$	
ANY $q1$ WHERE $q1 \in qd$	(B.33) ; $q1$ est en danger.
THEN $qd := qd - \{q1\}$	(B.34) ; $q1$ n'est plus en danger.
END	

B.2.6 Preuves effectuées sur le premier modèle

Afin de valider le modèle que nous avons construit, nous devons effectuer les preuves indiquées dans cette section. Une bonne partie de ces preuves seront

générées et réalisées à l'aide de l'atelier B.

Les preuves ont été classées en quatre groupes suivant le but recherché :

- Initialisation.
- Préservation de l'invariant.
- Absence de blocage. Le système doit toujours être en mesure d'évoluer.
- Traces d'événements.

Hypothèse B.1. *Nous supposons que les ensembles tt et qq sont disjoints et non vides. Ceci signifie que la ligne possède au moins un train et au moins un quai de station* [1] *:*

$$tt \in \mathbb{P}_1(INT) \wedge qq \in \mathbb{P}_1(INT) \wedge tt \cap qq = \emptyset \qquad \text{(A)}$$

Afin de simplifier l'écriture des obligations de preuve, nous appellerons cette formule A.

Initialisation

Nous voulons prouver ici que le système est initialisé correctement. Pour cela nous devons prouver la formule qui suit :

Obligation de Preuve B.1. Soit I l'invariant du système et soit U la substitution d'initialisation. Nous devons prouver :

$$A \quad \Rightarrow \quad [U]I \qquad \text{(OP B.1)}$$

Les obligations de preuve de ce groupe sont générées automatiquement par l'atelier B.

Préservation de l'invariant

Nous voulons prouver ici qu'à chaque observation d'un événement E le système reste dans un état qui vérifie l'invariant. Pour cela nous devons prouver pour chaque événement la formule qui suit :

Obligation de Preuve B.2. Soit I l'invariant du système, soit E la substitution associée à l'événement. Nous devons prouver :

$$A \wedge I \quad \Rightarrow \quad [E]I \qquad \text{(OP B.2)}$$

Nous avons par conséquent une obligation de preuve dans ce groupe pour chaque évènement du système. Ces obligations de preuve sont générées automatiquement parl'atelier B.

[1] Nous soupposons aussi qu'ils sont des sous-ensembles finis d'entiers, [Abr 96-1], section 5.2.5.

Absence de blocage

Nous voulons prouver ici que le système peut toujours évoluer. Pour cela, il suffit de prouver qu'il y a toujours un événement qui peut être observé. Ce qui signifie qu'à tout moment il y a au moins une garde qui est satisfaite.

Nous devons donc prouver la formule qui suit :

Obligation de Preuve B.3. Soit $G1, G2, \ldots, Gn$ les gardes associées aux événements du système :

$$A \wedge I \quad \Rightarrow \quad (G1 \vee G2 \vee \ldots \vee Gn) \qquad \text{(OP B.3)}$$

Traces d'événements

Nous voulons prouver ici que le système admet ou ne admet pas certaines traces d'événements.

D'abord, nous allons classer les événements spécifiés en trois groupes afin de définir leurs synchronisations.

Le *premier groupe* est formé uniquement par l'événement $Transfert$ puisqu'il peut être observé à tout moment indépendamment des traces déjà observées. Ceci signifie qu'il n'y a aucune synchronisation entre cet événement et les autres, ce qui correspond bien à notre intention : un transfert peut être observé pour tout train à tout moment, qu'il y ait ou non des trains ou des quais en danger.

Obligation de Preuve B.4. Afin de montrer que le système permet à l'événement $Transfert$ d'être observé à tout moment, nous devons prouver la formule qui suit :

$$A \wedge I \quad \Rightarrow \quad TRUE \qquad \text{(OP B.4)}$$

Le *deuxième groupe* est formé par les événements $TrainDanger$ et $TrainSur$ puisqu'ils peuvent être observés à tout moment indépendamment des traces déjà observées pour les événements des autres groupes. Ceci signifie qu'il n'y a aucune synchronisation entre ces deux événements et les autres, ce qui correspond bien à notre intention : un train peut devenir en danger (ou sûr, s'il est en danger) à tout moment, qu'il y ait ou non des quais en danger.

Obligation de Preuve B.5. Afin de montrer que le système permet aux événements $TrainDanger$ et $TrainSur$ d'être observés à tout moment, nous devons prouver la formule qui suit :

$$A \wedge I \quad \Rightarrow \quad \exists_{t1} \cdot (t1 \in tt \wedge t1 \notin td) \quad \vee \quad \exists_{t1} \cdot (t1 \in td) \qquad \text{(OP B.5)}$$

D'un autre côté, l'événement $TrainDanger$ ne peut pas être observé pour un train qui est déjà en danger. D'une façon semblable, l'événement $TrainSur$ ne peut pas être observé pour un train qui est déjà sûr.

Obligation de Preuve B.6. Afin de montrer que le système ne permet pas l'observation de ces traces, nous devons prouver les deux formules qui suivent, qui font intervenir les gardes des événements concernés :

$$t1 \in td \quad \Rightarrow \quad \neg(t1 \in tt \wedge t1 \notin td) \qquad \text{(OP B.6a)}$$
$$t1 \in tt \wedge t1 \notin td \quad \Rightarrow \quad \neg(t1 \in td) \qquad \text{(OP B.6b)}$$

Le *troisième groupe* est formé par les événements $QuaiDanger$ et $QuaiSur$ puisqu'ils peuvent être observés à tout moment indépendamment des traces déjà observées pour les événements des autres groupes. Ceci signifie qu'il n'y a aucune synchronisation entre ces deux événements et les autres, ce qui correspond bien à notre intention : un quai peut devenir en danger (ou sûr, s'il est en danger) à tout moment, qu'il y ait ou non des trains en danger.

Obligation de Preuve B.7. Afin de montrer que le système permet aux événements $QuaiDanger$ et $QuaiSur$ d'être observés à tout moment, nous devons prouver la formule qui suit :

$$A \wedge I \quad \Rightarrow \quad \exists_{q1} \cdot (q1 \in qq \wedge q1 \notin td) \quad \vee \quad \exists_{q1} \cdot (q1 \in td) \qquad \text{(OP B.7)}$$

D'un autre côté, l'événement $QuaiDanger$ ne peut pas être observé pour un quai qui est déjà en danger. D'une façon semblable, l'événement $QuaiSur$ ne peut pas être observé pour un quai qui est déjà sûr.

Obligation de Preuve B.8. Afin de montrer que le système ne permet pas l'observation de ces traces, nous devons prouver les deux formules qui suivent, qui font intervenir les gardes des événements concernés :

$$q1 \in qd \quad \Rightarrow \quad \neg(q1 \in qq \wedge q1 \notin qd) \qquad \text{(OP B.8a)}$$
$$q1 \in qq \wedge q1 \notin qd \quad \Rightarrow \quad \neg(q1 \in qd) \qquad \text{(OP B.8b)}$$

B.2.7 Deuxième niveau

Notre objectif ici est d'introduire les évènements qui permettent d'observer les détails d'un transfert normal de voyageurs.

Le deuxième principe donne la condition qu'il faut assurer pendant toute la durée d'un transfert : *Garantir l'immobilité du train.*

Un transfert de personnes n'est pas instantané, il est réalisé pendant une durée de temps. Lorsqu'un train réalise un transfert, il est dans un état très particulier qui l'habilite à ouvrir ses portes. Nous avons donc besoin de définir deux évènements pour observer le début et la fin d'un transfert.

Debut d'un transfert de voyageurs : TransfertVoyDeb. Dans la spécification $Meteor1.1$, il n'est pas possible d'observer le début d'un transfert. Nous ajoutons cet évènement ici. Il peut être observé à tout moment pour les trains qui sont immobilisés, qui ne réalisent pas de transfert et qui sont localisés sur un quai de station qui ne réalise pas de transfert. Si $t1$ dénote le train concerné, la garde

est :

$$\exists_{t1} \cdot (t1 \in ti \wedge t1 \notin ttr \wedge t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv) \tag{B.36}$$

Dans cette garde, nous avons supposé qu'un transfert de voyageurs ne peut commencer que lorsque le train est immobilisé.

L'action rend le train dans un état qui indique qu'il réalise un transfert. Ceci ne signifie pas que les portes sont ouvertes ; elles le seront un peu plus tard. L'événement permet plutôt d'observer le début de la séquence d'actions qui ont lieu lors d'un transfert. Il rend aussi le quai en cours de transfert. L'action peut s'écrire comme suit :

$$ttv, qtv, ttr := ttv \cup \{t1\}, qtv \cup t\_q[t1], ttr \cup \{t1\} \tag{B.37}$$

La variable ttr est modifié aussi par cette action. Ceci est indispensable pour être en mesure de prouver l'invariant (B.7).

Fin d'un transfert de voyageurs : Dans la spécification $Meteor1.1$, il est possible d'observer la fin de tout transfert avec l'évènement $Transfert$. Notre intention ici est d'observer séparément la fin d'un transfert de voyageurs. Pour cela nous décomposons l'évènement $Transfert$ en deux de la façon indiquée dans la section 5.6.2, page 103 :

- $TransfertVoy$: Permet d'observer la fin d'un transfert de voyageurs. Il peut être observé seulement pour les trains qui effectuent déjà un transfert de voyageurs. Sa garde est :

  $$\exists_{t1} \cdot (t1 \in ttv) \tag{B.38}$$

  Son action rend le train dans un état qui indique qu'il ne effectue pas de transfert. Il a le même effet sur le quai associé. Si $t1$ dénote le train concerné, l'action peut s'écrire comme suit :

  $$ttv, qtv, ttr := ttv - \{t1\}, qtv - t\_q[t1], ttr - \{t1\} \tag{B.39}$$

- $Transfert1$: Permet d'observer la fin de tous les autres types de transfert. Evacuation d'urgence, transfert de service et tests de portes. Il peut être observé pour tous les trains qui n'effctuent pas de transfert de voyageurs mais qui sont immobilisés. Sa garde est :

  $$\exists_{t1} \cdot (t1 \in ti \wedge t1 \notin ttv) \tag{B.40}$$

  Son action est la même que celle de l'évènement $Transfert$ de la spécification $Meteor1.1$: il ne modifie pas les variables du système. L'action peut s'écrire comme suit :

  $$SKIP \tag{B.41}$$

Puisque dans la formule B.36, page 174 nous utilisons la variable ti, nous devons prévoir ici les évènements qui lui permettent d'évolouer de la façon souhaitée. Pour cela, nous ajoutons les deux évènements qui suivent :

- $TrainImmobil$: Permet d'observer l'immobilisation d'un train. Il peut être observé seulement pour les trains non immobilisés. Sa garde est :

$$\exists_{t1} \cdot (t1 \in tt \wedge t1 \notin ti) \tag{B.42}$$

  Son action rend le train immobil :

$$ti := ti \cup \{t1\} \tag{B.43}$$

- $TrainMobil$: Permet d'observer l'effet contraire : le train devient mobil. Il peut être observé pour tous les trains immobilisés qui n'effectuent pas de transfert. Sa garde est :

$$\exists_{t1} \cdot (t1 \in ti \wedge t1 \notin ttr) \tag{B.44}$$

  Son action rend le train mobil :

$$ti := ti - \{t1\} \tag{B.45}$$

Puisque dans la formule B.36, page 174, nous utilisons la variable t_q qui introduit la notion de train localisé dans un quai, nous devons prévoir ici les évènements qui lui permettent d'évolouer de la façon souhaitée. Pour cela, nous ajoutons les deux évènements qui suivent :

- $ArriveeQuai$: Permet d'observer l'arrivée d'un train dans un quai. Il peut être observé seulement pour les trains mobils qui ne sont pas à quai, et pour les quais vides. Sa garde est :

$$\exists_{t1,q1} \cdot (t1 \in tt \wedge q1 \in qq \wedge t1 \notin dom(t\_q) \wedge q1 \notin ran(t\_q) \wedge t1 \notin ti) \tag{B.46}$$

  Son action rend le train localisé précisemment dans le quai indiqué :

$$t\_q := t\_q \cup \{t1 \mapsto q1\} \tag{B.47}$$

- $DepartQuai$: Permet d'observer l'effet contraire : le train quitte le quai. Il peut être observé pour tous les trains mobils à quai. Sa garde est :

$$\exists_{t1} \cdot (t1 \in dom(t\_q) \wedge t1 \notin ti) \tag{B.48}$$

  Son action fait partir le train et rend le quai vide : [2] :

$$t\_q := \{t1\} \lhd\!\!\!-\ t\_q \tag{B.49}$$

[2] Voir [Abr 96-1], section 2.4.1.

Pour initialiser le système, nous supposons qu'au départ :

- la ligne est sûre,
- il n'y a aucun transfert en cours,
- il n'y a aucun train immobilisé,
- il n'y a aucun train à quai.

En conséquence, le système est initialisé comme suit :

$$t\_q, ttr, ttv, td, ti, qtv, qd := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \quad \text{(B.50)}$$

La spécification complète du deuxième niveau d'abstraction est :

SYSTEM	
$Meteor1.2$	
VARIABLES	
$\mathbf{t\_q}$	Trains localisés à quai.
$\mathbf{ttr}$,	Trains qui effectuent un transfert.
$\mathbf{ttv}$,	Trains qui effectuent un transfert voyageurs.
td,	Trains en danger.
$\mathbf{ti}$,	Trains immobilisés.
$\mathbf{qtv}$,	Quais qui effectuent un transfert voyageurs.
qd,	Quais en danger.
INVARIANT	
$\mathbf{t\_q} \in \mathbf{tt} \nrightarrow \mathbf{qq} \wedge$	(B.1).
$\mathbf{ttr} \subseteq \mathbf{tt} \wedge$	(B.6).
$\mathbf{ttv} \subseteq \mathbf{ttr} \wedge$	(B.7).
$\mathbf{ttv} \in \mathbf{dom}(\mathbf{t\_q}) \wedge$	(B.8).
$td \subseteq tt \wedge$	(B.11).
$\mathbf{ti} \subseteq \mathbf{tt} \wedge$	(B.13).
$\mathbf{ttr} \subseteq \mathbf{ti} \wedge$	(B.14).
$\mathbf{qtv} = \mathbf{t\_q}[\mathbf{ttv}] \wedge$	(B.16).
$qd \subseteq qq \wedge$	(B.17).
$\mathbf{ttv} \subseteq \mathbf{ti}$	(B.22) **(P 2).**
INITIALISATION	
$t\_q, ttr, ttv, td, ti, qtv, qd :=$ $\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$	(B.50) ; Etat initial de la ligne.
EVENTS	
TrainDanger $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin td$	(B.27) ; $t1$ n'est pas en danger.
THEN $td := td \cup \{t1\}$	(B.28) ; $t1$ devient en danger.
END ;	
QuaiDanger $\hat{=}$	
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qd$	(B.29) ; $q1$ n'est pas en danger.
THEN $qd := qd \cup \{q1\}$	(B.30) ; $q1$ devient en danger.
END ;	
TrainSur $\hat{=}$	
ANY $t1$ WHERE $t1 \in td$	(B.31) ; $t1$ est en danger.
THEN $td := td - \{t1\}$	(B.32) ; $t1$ n'est plus en danger.
END ;	

QuaiSur $\hat{=}$	
ANY $q1$ WHERE $q1 \in qd$	(B.33) ; $q1$ est en danger.
THEN $qd := qd - \{q1\}$	(B.34) ; $q1$ n'est plus en danger.
END ;	
TransfertVoyDeb $\hat{=}$	
ANY $t1$ WHERE	
$t1 \in ti \wedge t1 \notin ttr \wedge$	(B.36) ;
$t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv$	$t1$ n'est pas en transfert.
THEN $ttv, qtv, ttr := ttv \cup \{t1\}$,	(B.37) ;
$qtv \cup t\_q[t1], ttr \cup \{t1\}$	$t1$ commence un Transfert.
END ;	
TransfertVoy $\hat{=}$	
ANY $t1$ WHERE $t1 \in ttv$	(B.38) ; $t1$ réalise un transfert.
THEN $ttv, qtv, ttr := ttv - \{t1\}$,	
$qtv - t\_q[t1], ttr - \{t1\}$	(B.39) ; Le transfert est fini.
END ;	
Transfert1 $\hat{=}$	
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttv$	(B.40) ;$t1$ n'est pas en transfert.
THEN $SKIP$	(B.41) ; le transfert est fini.
END ;	
TrainImmobil $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin ti$	(B.42) ; $t1$ est mobil.
THEN $ti := ti \cup \{t1\}$	(B.43) ; $t1$ est immobil.
END ;	
TrainMobil $\hat{=}$	
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttr$	(B.44) ; $t1$ est immobil.
THEN $ti := ti - \{t1\}$	(B.45) ; $t1$ est mobil.
END ;	
ArriveeQuai $\hat{=}$	
ANY $t1, q1$ WHERE $t1 \in tt \wedge$	
$q1 \in qq \wedge t1 \notin dom(t\_q) \wedge$	(B.46) ; $t1$ n'est pas à quai,
$q1 \notin ran(t\_q) \wedge t1 \notin ti$	$q1$ est vide.
THEN $t\_q := t\_q \cup \{t1 \mapsto q1\}$	(B.47) ; $t1$ est au quai $q1$.
END ;	
DepartQuai $\hat{=}$	
ANY $t1$ WHERE $t1 \in dom(t\_q) \wedge$	
$t1 \notin ti$	(B.48) ; $t1$ est à quai.
THEN $t\_q := \{t1\} \lhd\!\!\!- \; t\_q$	(B.49) ; $t1$ n'est plus à quai.
END ;	

B.2.8 Preuves effectuées sur le deuxième modèle

Afin de valider la spécification $Meteor1.2$, nous devons effectuer plusieurs classes de preuve :

- Préservation de l'initialisation.
- Préservation de l'invariant.
- Absence de blocage.
- Correction du raffinement.
- Traces d'évènements.

La démarche à suivre est semblable à celle présentée pour la spécification $Meteor1.1$. Nous ne le ferons pas ni pour cette spécification ni pour les suivantes.

B.2.9 Troisième niveau

Notre objectif ici est d'introduire les évènements qui permettent d'observer les détails d'ouverture des portes.

En conditions normales de fonctionnement, les portes ne doivent être ouvertes que pendant les transferts de personnes. Nous allons donc introduire quatre nouveaux évènements qui permettent d'observer les différents cas d'ouverture normale de portes des trains et des quais. Pour cela nous avons besoin des variables to, qo.

- $TrainOuvert$: Permet d'observer l'ouverture d'un train. Nous considérons qu'un train est ouvert lorsqu'une de ses portes est ouverte. L'événement peut être observé seulement pour les trains fermés qui effectuent un transfert. Sa garde est :

$$\exists_{t1} \cdot (t1 \in ttr \wedge t1 \notin to) \tag{B.51}$$

  Son action rend le train ouvert :

$$to := to \cup \{t1\} \tag{B.52}$$

- $TrainFerme$: Permet d'observer l'effet contraire : le train devient fermé. L'événement peut être observé pour tous les trains ouverts. Sa garde est :

$$\exists_{t1} \cdot (t1 \in to) \tag{B.53}$$

  Son action rend le train fermé :

$$to := to - \{t1\} \tag{B.54}$$

- $QuaiOuvert$: Permet d'observer l'ouverture d'un quai. Nous considérons qu'un quai est ouvert lorsqu'une de ses portes est ouverte. L'événement peut être observé seulement pour les quais fermés qui soit effectuent un transfert soit sont en sécurité. Sa garde est :

$$\exists_{q1} \cdot (q1 \in qtv \cup qs \wedge q1 \notin qo) \tag{B.55}$$

  Son action rend le quai ouvert :

$$qo := qo \cup \{q1\} \tag{B.56}$$

- $QuaiFerme$: Permet d'observer l'effet contraire : le quai devient fermé. L'événement peut être observé pour tous les quai ouverts. Sa garde est :

$$\exists_{q1} \cdot (q1 \in qo) \tag{B.57}$$

  Son action rend le quai fermé :

$$qo := qo - \{q1\} \tag{B.58}$$

Puisque la formule B.55, page 178 utilise la variable qs, nous devons prévoir ici les évènements qui lui permettent d'évoluer. Pour cela, nous allons introduire deux nouveaux évènements qui permettent d'observer la mise en sécurité de la zone où est localisé un quai :

- $ZoneQuaiSec$: Permet d'observer la mise en sécurité de la zone où est localisé un quai. L'événement peut être observé seulement pour les quais dont les zones ne sont pas en sécurité. Sa garde est :

$$\exists_{q1} \cdot (q1 \in qq \wedge q1 \notin qs) \tag{B.59}$$

  Son action rend la zone associée au quai en sécurité :

$$qs := qs \cup \{q1\} \tag{B.60}$$

- $ZoneQuaiNor$: Permet d'observer l'effet contraire. La zone où se trouve le quai n'est plus en sécurité. L'événement peut être observé pour tous les quais dont leurs zones sont en sécurité. Sa garde est :

$$\exists_{q1} \cdot (q1 \in qs) \tag{B.61}$$

  Son action rend normal la zone associée au quai, c'est à dire que la zone n'est plus en sécurité :

$$qs := qs - \{q1\} \tag{B.62}$$

La fin d'un transfert de voyageurs ne peut être observé que si les portes du train et celles du quai sont déjà fermées. Nous devons donc modifier le garde de l'évènement $TransfertVoy$ comme suit :

$$\exists_{t1} \cdot (t1 \in ttv \wedge t1 \notin to \wedge t\_q(t1) \notin qo) \tag{B.63}$$

Nous supposons qu'au départ :

- aucun train n'est ouvert,

- aucun quai n'est ouvert,
- aucune zone de quai n'est en sécurité.

En conséquence, le système est initialisé comme suit :

$$t\_q, to, ttr, ttv, td, ti, qo, qtv, qd, qs := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \qquad \text{(B.64)}$$

La spécification complète que nous obtenons est :

SYSTEM	
$Meteor1.3$	
VARIABLES	
t_q	Trains localisés à quai.
$\mathbf{to}$,	Trains ouverts. Non hermétiquement clos.
ttr,	Trains qui effectuent un transfert.
ttv,	Trains qui effectuent un transfert voyageurs.
td,	Trains en danger.
ti,	Trains immobilisés.
$\mathbf{qo}$,	Quais ouverts. Non hermétiquement clos.
qtv,	Quais qui effectuent un transfert voyageurs.
qd,	Quais en danger.
$\mathbf{qs}$,	Quais mis en sécurité.
INVARIANT	
$t\_q \in tt \mapsto qq \wedge$	(B.1).
$\mathbf{to} \subseteq \mathbf{tt} \wedge$	(B.5).
$ttr \subseteq tt \wedge$	(B.6).
$ttv \subseteq ttr \wedge$	(B.7).
$ttv \in dom(t\_q) \wedge$	(B.8).
$td \subseteq tt \wedge$	(B.11).
$ti \subseteq tt \wedge$	(B.13).
$ttr \subseteq ti \wedge$	(B.14).
$\mathbf{qo} \subseteq \mathbf{qq} \wedge$	(B.15).
$qtv = t\_q[ttv] \wedge$	(B.16).
$qd \subseteq qq \wedge$	(B.17).
$\mathbf{qs} \subseteq \mathbf{qq} \wedge$	(B.18).
$\mathbf{to} \subseteq \mathbf{ttr}$	(B.20) **(P 1.1).**
$\mathbf{qo} \subseteq \mathbf{qtv} \cup \mathbf{qs}$	(B.21) **(P 1.2 et P 4).**
$ttv \subseteq ti$	(B.22) **(P 2).**
INITIALISATION	
$t\_q, to, ttr, ttv, td,$ $ti, qo, qtv, qd, qs :=$ $\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$	(B.64) ; Etat initial de la ligne.
EVENTS	
TrainDanger $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin td$	(B.27) ; $t1$ n'est pas en danger.
THEN $td := td \cup \{t1\}$	(B.28) ; $t1$ devient en danger.
END ;	
QuaiDanger $\hat{=}$	
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qd$	(B.29) ; $q1$ n'est pas en danger.
THEN $qd := qd \cup \{q1\}$	(B.30) ; $q1$ devient en danger.
END ;	
TrainSur $\hat{=}$	
ANY $t1$ WHERE $t1 \in td$	(B.31) ; $t1$ est en danger.
THEN $td := td - \{t1\}$	(B.32) ; $t1$ n'est plus en danger.
END ;	

QuaiSur $\hat{=}$	
ANY $q1$ WHERE $q1 \in qd$	(B.33) ; $q1$ est en danger.
THEN $qd := qd - \{q1\}$	(B.34) ; $q1$ n'est plus en danger.
END ;	
TransfertVoyDeb $\hat{=}$	
ANY $t1$ WHERE	
$t1 \in ti \wedge t1 \notin ttr \wedge$	(B.36) ;
$t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv$	$t1$ n'est pas en transfert.
THEN $ttv, qtv, ttr := ttv \cup \{t1\},$	(B.37) ;
$qtv \cup t\_q[t1], ttr \cup \{t1\}$	$t1$ commence un Transfert.
END ;	
TransfertVoy $\hat{=}$	
ANY $t1$ WHERE $t1 \in ttv \wedge$	(B.38) ;
$t1 \notin to \wedge t\_q(t1) \notin qo$	(B.63) ;$t1$ réalise un transfert.
THEN $ttv, qtv, ttr := ttv - \{t1\},$	
$qtv - t\_q[t1], ttr - \{t1\}$	(B.39) ; Le transfert est fini.
END ;	
Transfert1 $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge$	(B.40) ;
$t1 \notin ttv \wedge t1 \in ti$	$t1$ n'est pas en transfert.
THEN $SKIP$	(B.41) ; le transfert est fini.
END ;	
TrainImmobil $\hat{=}$	
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin ti$	(B.42) ; $t1$ est mobil.
THEN $ti := ti \cup \{t1\}$	(B.43) ; $t1$ est immobil.
END ;	
TrainMobil $\hat{=}$	
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttr$	(B.44) ; $t1$ est immobil.
THEN $ti := ti - \{t1\}$	(B.45) ; $t1$ est mobil.
END ;	
ArriveeQuai $\hat{=}$	
ANY $t1, q1$ WHERE $t1 \in tt \wedge$	
$q1 \in qq \wedge t1 \notin dom(t\_q) \wedge$	(B.46) ; $t1$ n'est pas à quai,
$q1 \notin ran(t\_q) \wedge t1 \notin ti$	$q1$ est vide.
THEN $t\_q := t\_q \cup \{t1 \mapsto q1\}$	(B.47) ; $t1$ est au quai $q1$.
END ;	
DepartQuai $\hat{=}$	
ANY $t1$ WHERE $t1 \in dom(t\_q) \wedge$	
$t1 \notin ti$	(B.48) ; $t1$ est à quai.
THEN $t\_q := \{t1\} \lhd t\_q$	(B.49) ; $t1$ n'est plus à quai.
END ;	
TrainOuvert $\hat{=}$	
ANY $t1$ WHERE $t1 \in ttr \wedge t1 \notin to$	(B.51) ;$t1$ n'est pas ouvert.
THEN $to := to \cup \{t1\}$	(B.52) ; $t1$ devient ouvert.
END ;	
TrainFerme $\hat{=}$	
ANY $t1$ WHERE $t1 \in to$	(B.53) ; $t1$ est ouvert.
THEN $to := to - \{t1\}$	(B.54) ; $t1$ est fermé.
END ;	
QuaiOuvert $\hat{=}$	
ANY $q1$ WHERE $q1 \in qtv \cup qs \wedge$	
$q1 \notin qo$	(B.55) ;$q1$ n'est pas ouvert.
THEN $qo := qo \cup \{q1\}$	(B.56) ; $q1$ devient ouvert.
END ;	

QuaiFerme $\hat{=}$	
ANY $q1$ WHERE $q1 \in qo$	(B.57) ; $q1$ est ouvert.
THEN $qo := qo - \{q1\}$	(B.58) ; $q1$ est fermé.
END ;	
ZoneQuaiSec $\hat{=}$	
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qs$	(B.59) ; $q1$ n'est pas en sécurité.
THEN $qs := qs \cup \{q1\}$	(B.60) ; $q1$ est en sécurité.
END ;	
ZoneQuaiNor $\hat{=}$	
ANY $q1$ WHERE $q1 \in qs$	(B.61) ; $q1$ est en sécurité.
THEN $qs := qs - \{q1\}$	(B.62) ; $q1$ n'est plus en sécurité.
END ;	

B.2.10 Quatrième niveau

Notre objectif ici est d'introduire les évènements qui permettent d'observer les détails d'une évacuation d'urgence.

Le troisième principe donne les conditions qu'il faut assurer pendant toute la durée d'une évacuation d'urgence : *garantir l'évacuation dans un environnement sûr.*

Une évacuation d'urgence n'est pas instantanée ; elle est réalisée pendant une durée de temps. Lorsqu'un train est évacué, il est dans un état qui l'habilite à ouvrir ses portes et à permettre l'accès de ses voyageurs sur la zone de circulation de trains. Nous avons donc besoin de spécifier deux évènements pour observer le début et la fin d'une évacuation d'urgence.

Debut d'une évacuation d'urgence (EvacuationDeb) : Dans la spécification $Meteor1.3$, il n'est pas possible d'observer le début d'une évacuation. Par conséquent, nous ajoutons cet évènement ici. Il peut être observé à tout moment pour les trains qui sont immobilisés, qui sont en danger, qui ne réalisent pas de transfert et qui se trouvent dans une zone d'évacuation sûre. Si $t1$ dénote le train concerné, la garde est :

$$\exists_{t1} \cdot (t1 \in ti \wedge t1 \in td \wedge t1 \notin ttr \wedge t\_z[t1] \subseteq zs) \tag{B.65}$$

L'action rend le train dans un état d'évacuation. L'événement permet d'observer le début de la séquence d'actions qui ont lieu lors d'une évacuation d'urgence. L'action peut s'écrire comme suit :

$$teu, ttr := teu \cup \{t1\}, ttr \cup \{t1\} \tag{B.66}$$

Fin d'une évacuation d'urgence : Dans la spécification $Meteor1.3$, il est possible d'observer la fin de plusieurs types de transfert avec l'évènement $Transfert1$. Notre intention ici est d'observer séparément la fin d'une évacuation d'urgence. La fin des autres types de transfert sera observée avec un autre évènement. Pour cela nous décomposons l'évènement $Transfert1$ en deux, comme suit :

- *Evacuation* : Permet d'observer la fin d'une évacuation d'urgence. L'événement peut être observé seulement pour les trains qui réalisent déjà une

évacuation d'urgence, qui ne sont pas ouverts et qui ne sont plus en danger. Sa garde est :

$$\exists_{t1} \cdot (t1 \in teu \wedge t1 \notin to \wedge t1 \notin td) \tag{B.67}$$

Son action termine l'évacuation :

$$teu, ttr := teu - \{t1\}, ttr - \{t1\} \tag{B.68}$$

- $Transfert2$: Permet d'observer la fin de tous les autres types de transfert : *Transfert de service et tests des portes.* L'événement peut être observé pour tous les trains qui ne réalisent ni de transfert de voyageurs ni d'évacuation d'urgence mais qui sont immobilisés. Sa garde est :

$$\exists_{t1} \cdot (t1 \notin ttv \cup teu \wedge t1 \in ti) \tag{B.69}$$

Son action ne modifie pas les variables du système :

$$SKIP \tag{B.70}$$

Puisque dans la garde B.65 nous utilisons les variables t_z et zs nous devons inclure ici les évènements qui leur permettent d'évoluer. Pour cela, nous introduisons les quatre évènements qui suivent :

- $EntreeZoneEva$: Permet d'observer l'arrivée d'un train dans une zone d'évacuation. L'événement peut être observé seulement pour les trains mobiles qui entrent dans une nouvelle zone. D'un autre côté, nous imposons une autre contrainte tirée plutôt de la connaissance globale du système qui interdit à tout train d'entrer dans une zone d'évacuation mise en sécurité. Sa garde est donc :

$$\exists_{t1,z1} \cdot (t1 \in tt \wedge t1 \notin ti \wedge z1 \notin t\_z[t1] \wedge z1 \in ze \wedge z1 \notin zs) \tag{B.71}$$

Son action fait entrer le train dans la zone concernée :

$$t\_z := t\_z \cup \{t1 \mapsto z1\} \tag{B.72}$$

- $SortieZoneEva$: Permet d'observer le départ d'un train d'une zone d'évacuation. L'événement peut être observé seulement pour les trains mobiles qui quittent une zone. Nous imposons ici aussi que la zone ne soit pas en sécurité, ce qui signifie que dans une zone en sécurité les trains sont arrétées. Sa garde est donc :

$$\exists_{t1,z1} \cdot (t1 \in tt \wedge t1 \notin ti \wedge z1 \in t\_z[t1] \wedge z1 \notin zs) \tag{B.73}$$

Son action fait sortir le train de la zone concernée :

$$t\_z := t\_z - \{t1 \mapsto z1\} \tag{B.74}$$

- *ZoneEvaSec* : Permet d'observer la mise en sécurité d'une zone d'évacuation. L'événement peut être observé pour toutes les zones d'évacuation qui ne sont pas en sécurité. Sa garde est :

$$\exists_{z1} \cdot (z1 \in ze \wedge z1 \notin zs) \tag{B.75}$$

Son action met la zone en sécurité :

$$zs := zs \cup \{z1\} \tag{B.76}$$

- *ZoneEvaNor* : Permet d'observer la remise en état normal d'une zone d'évacuation. L'événement peut être observé pour toutes les zones d'évacuation en sécurité qui ne contiennent pas de train en danger. Sa garde est :

$$\exists_{z1} \cdot (z1 \in zs \wedge z1 \notin t\_z[td]) \tag{B.77}$$

Son action remet la zone dans son état normal de fonctionnement :

$$zs := zs - \{z1\} \tag{B.78}$$

Nous supposons qu'au départ :

- aucune train n'est en évacuation d'urgence,
- aucune zone n'est en sécurité,
- chaque train se trouvent au moins dans une zone.

En conséquence, nous initialisons le système comme suit :

$$\begin{array}{l} t\_q, t\_z, to, ttr, ttv, teu, td, ti, qo, qtv, qd, qs, zs : \\ \quad t\_q = \emptyset \wedge t\_z \in tt \leftrightarrow ze \wedge dom(t\_z) = tt \wedge \\ \quad to = \emptyset \wedge ttr = \emptyset \wedge ttv = \emptyset \wedge teu = \emptyset \wedge td = \emptyset \wedge \\ \quad ti = \emptyset \wedge qo = \emptyset \wedge qtv = \emptyset \wedge qd = \emptyset \wedge qs = \emptyset \wedge zs = \emptyset \end{array} \tag{B.79}$$

La spécification complète obtenue est :

SYSTEM
$Meteor1.4$

SETS
ze — Zones d'évacuation ; la voie est découpée en zones.

CONSTANTS
q_z — Localisation des quais par rapport aux zones.

PROPERTIES
$\mathbf{q\_z} \in \mathbf{qq} \leftrightarrow \mathbf{ze}$ — (B.3).

VARIABLES
t_q — Trains localisés à quai.
t_z — Localisation des trains par rapport aux zones.
to, — Trains ouverts. Non hermétiquement clos.
ttr, — Trains qui effectuent un transfert.
ttv, — Trains qui effectuent un transfert voyageurs.
teu, — Trains qui effectuent une évacuation d'urgence.
td, — Trains en danger.
ti, — Trains immobilisés.
qo, — Quais ouverts. Non hermétiquement clos.
qtv, — Quais qui effectuent un transfert voyageurs.
qd, — Quais en danger.
qs, — Quais en sécurité.
zs, — Zones d'évacuation en sécurité.

INVARIANT

$t\_q \in tt \rightarrowtail qq \wedge$ — (B.1).
$\mathbf{t\_z} \in \mathbf{tt} \leftrightarrow \mathbf{ze} \wedge \mathbf{dom(t\_z)} = \mathbf{tt} \wedge$ — (B.2).
$\mathbf{t\_q^{-1}}; \mathbf{t\_z} \subseteq \mathbf{q\_z}$ — (B.4).
$to \subseteq tt \wedge$ — (B.5).
$ttr \subseteq tt \wedge$ — (B.6).
$ttv \subseteq ttr \wedge$ — (B.7).
$ttv \in dom(t\_q) \wedge$ — (B.8).
$\mathbf{teu} \subseteq \mathbf{ttr} \wedge$ — (B.9).
$\mathbf{ttv} \cap \mathbf{teu} = \emptyset \wedge$ — (B.10).
$td \subseteq tt \wedge$ — (B.11).
$\mathbf{teu} \subseteq \mathbf{td} \wedge$ — (B.12).
$ti \subseteq tt \wedge$ — (B.13).
$ttr \subseteq ti \wedge$ — (B.14).
$qo \subseteq qq \wedge$ — (B.15).
$qtv = t\_q[ttv] \wedge$ — (B.16).
$qd \subseteq qq \wedge$ — (B.17).
$qs \subseteq qq \wedge$ — (B.18).
$\mathbf{zs} \subseteq \mathbf{ze} \wedge$ — (B.19).
$to \subseteq ttr$ — (B.20) **(P 1.1).**
$qo \subseteq qtv \cup qs$ — (B.21) **(P 1.2 et P 4).**
$ttv \subseteq ti$ — (B.22) **(P 2).**
$\mathbf{teu} \subseteq \mathbf{t\_z^{-1}}[\mathbf{zs}]$ — (B.23) **(P 3.1).**
$\mathbf{ran(q\_z)} = \mathbf{ze}$ — (B.24) **(P 3.2).**

INITIALISATION

$t\_q, t\_z, to, ttr, ttv, teu,$
$td, ti, qo, qtv, qd, qs, zs :$
$t\_q = \emptyset \wedge t\_z \in tt \leftrightarrow ze \wedge$
$dom(t\_z) = tt \wedge to = \emptyset \wedge$
$ttr = \emptyset \wedge ttv = \emptyset \wedge teu = \emptyset \wedge$
$td = \emptyset \wedge ti = \emptyset \wedge qo = \emptyset \wedge$
$qtv = \emptyset \wedge qd = \emptyset \wedge qs = \emptyset \wedge$
$zs = \emptyset$ — (B.79) ; Etat initial de la ligne.

EVENTS

TrainDanger $\hat{=}$
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin td$ (B.27) ; $t1$ n'est pas en danger.
THEN $td := td \cup \{t1\}$ (B.28) ; $t1$ devient en danger.
END ;

QuaiDanger $\hat{=}$
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qd$ (B.29) ; $q1$ n'est pas en danger.
THEN $qd := qd \cup \{q1\}$ (B.30) ; $q1$ devient en danger.
END ;

TrainSur $\hat{=}$
ANY $t1$ WHERE $t1 \in td$ (B.31) ; $t1$ est en danger.
THEN $td := td - \{t1\}$ (B.32) ; $t1$ n'est plus en danger.
END ;

QuaiSur $\hat{=}$
ANY $q1$ WHERE $q1 \in qd$ (B.33) ; $q1$ est en danger.
THEN $qd := qd - \{q1\}$ (B.34) ; $q1$ n'est plus en danger.
END ;

TransfertVoyDeb $\hat{=}$
ANY $t1$ WHERE
$t1 \in ti \wedge t1 \notin ttr \wedge$ (B.36) ;
$t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv$ $t1$ n'est pas en transfert.
THEN $ttv, qtv, ttr := ttv \cup \{t1\},$ (B.37) ;
$qtv \cup t\_q[t1], ttr \cup \{t1\}$ $t1$ commence un Transfert.
END ;

TransfertVoy $\hat{=}$
ANY $t1$ WHERE $t1 \in ttv \wedge$ (B.38) ;$t1$ réalise un transfert.
$t1 \notin to \wedge t\_q(t1) \notin qo$ (B.63) ;$t1$ est fermé.
THEN $ttv, qtv, ttr := ttv - \{t1\},$
$qtv - t\_q[t1], ttr - \{t1\}$ (B.39) ; Le transfert est fini.
END ;

TrainImmobil $\hat{=}$
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin ti$ (B.42) ; $t1$ est mobil.
THEN $ti := ti \cup \{t1\}$ (B.43) ; $t1$ est immobil.
END ;

TrainMobil $\hat{=}$
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttr$ (B.44) ; $t1$ est immobil.
THEN $ti := ti - \{t1\}$ (B.45) ; $t1$ est mobil.
END ;

ArriveeQuai $\hat{=}$
ANY $t1, q1$ WHERE $t1 \in tt \wedge$
$q1 \in qq \wedge t1 \notin dom(t\_q) \wedge$ (B.46) ; $t1$ n'est pas à quai,
$q1 \notin ran(t\_q) \wedge t1 \notin ti$ $q1$ est vide.
THEN $t\_q := t\_q \cup \{t1 \mapsto q1\}$ (B.47) ; $t1$ est au quai $q1$.
END ;

DepartQuai $\hat{=}$
ANY $t1$ WHERE $t1 \in dom(t\_q) \wedge$
$t1 \notin ti$ (B.48) ; $t1$ est à quai.
THEN $t\_q := \{t1\} \lhd t\_q$ (B.49) ; $t1$ n'est plus à quai.
END ;

TrainOuvert $\hat{=}$
ANY $t1$ WHERE $t1 \in ttr \wedge t1 \notin to$ (B.51) ;$t1$ n'est pas ouvert.
THEN $to := to \cup \{t1\}$ (B.52) ; $t1$ devient ouvert.
END ;

TrainFerme $\hat{=}$	
ANY $t1$ WHERE $t1 \in to$	(B.53) ; $t1$ est ouvert.
THEN $to := to - \{t1\}$	(B.54) ; $t1$ est fermé.
END ;	
QuaiOuvert $\hat{=}$	
ANY $q1$ WHERE $q1 \in qtv \cup qs \wedge$	
$q1 \notin qo$	(B.55) ;$q1$ n'est pas ouvert.
THEN $qo := qo \cup \{q1\}$	(B.56) ; $q1$ devient ouvert.
END ;	
QuaiFerme $\hat{=}$	
ANY $q1$ WHERE $q1 \in qo$	(B.57) ; $q1$ est ouvert.
THEN $qo := qo - \{q1\}$	(B.58) ; $q1$ est fermé.
END ;	
ZoneQuaiSec $\hat{=}$	
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qs$	(B.59) ; $q1$ n'est pas en sécurité.
THEN $qs := qs \cup \{q1\}$	(B.60) ; $q1$ est en sécurité.
END ;	
ZoneQuaiNor $\hat{=}$	
ANY $q1$ WHERE $q1 \in qs$	(B.61) ; $q1$ est en sécurité.
THEN $qs := qs - \{q1\}$	(B.62) ; $q1$ n'est plus en sécurité.
END ;	
EvacuationDeb $\hat{=}$	
ANY $t1$ WHERE	
$t1 \in ti \wedge t1 \in td \wedge$	(B.65) ;
$t1 \notin ttr \wedge t\_z[t1] \subseteq zst1$	$t1$ n'est pas en évacuation.
THEN $teu, ttr :=$	(B.66) ;
$teu \cup \{t1\}, ttr \cup \{t1\}$	$t1$ commence une évacuation.
END ;	
Evacuation $\hat{=}$	
ANY $t1$ WHERE $t1 \in teu \wedge$	
$t1 \notin to \wedge t1 \notin td$	(B.67) ;$t1$ est en évacuation.
THEN $teu, ttr :=$	(B.68) ;
$teu - \{t1\}, ttr - \{t1\}$	L'évacuation est terminée.
END ;	
Transfert2 $\hat{=}$	
ANY $t1$ WHERE $t1 \notin ttv \cup teu \wedge$	(B.69) ;
$t1 \in ti$	$t1$ ne réalise pas de transfert.
THEN $SKIP$	(B.70) ; un transfert est fini.
END ;	
EntreeZoneEva $\hat{=}$	
ANY $t1, z1$ WHERE $t1 \in tt \wedge$	
$t1 \notin ti \wedge z1 \notin t\_z[t1] \wedge$	(B.71) ;
$z1 \in ze \wedge z1 \notin zs$	$t1$ n'est pas dans la zone $z1$.
THEN $t\_z := t\_z \cup \{t1 \mapsto z1\}$	(B.72) ; $t1$ entre dans la zone $z1$.
END ;	
SortieZoneEva $\hat{=}$	
ANY $t1, z1$ WHERE $t1 \in tt \wedge$	(B.73) ;
$t1 \notin ti \wedge z1 \in t\_z[t1] \wedge z1 \notin zs$	$t1$ est dans la zone z1.
THEN $t\_z := t\_z - \{t1 \mapsto z1\}$	(B.74) ; $t1$ n'est plus à quai.
END ;	
ZoneEvaSec $\hat{=}$	
ANY $z1$ WHERE $z1 \in ze \wedge$	(B.75) ;
$z1 \notin zs$	$z1$ n'est pas en sécurité.
THEN $zs := zs \cup \{z1\}$	(B.76) ; $z1$ est en sécurité.
END ;	
ZoneEvaNor $\hat{=}$	
ANY $z1$ WHERE $z1 \in zs \wedge$	(B.77) ;
$z1 \notin t\_z[td]$	$z1$ est en sécurité.
THEN $zs := zs - \{z1\}$	(B.78) ; $z1$ n'est plus en sécurité.
END ;	

Bibliographie

[Aba,Lam 88] M. Abadi and L. Lamport, *The existence of refinement mappings*, Technical Report. Digital Systems Research Center, Palo Alto, California, 1988.

[Aba,Lam 93] M. Abadi and L. Lamport, *Composing Specifications*, ACM Transactions on Programming Languages and Systems 15, 1, 73-132, 1993.

[Aba,Lam 95] M. Abadi and L. Lamport, *Conjoining Specifications*, ACM Transactions on Programming Languages and Systems 17, 3, 507-534, 1995.

[Abr 96-1] Abrial J.R., *The B Book. Assigning programs to Meanings*, Cambridge University Press, 1996

[Abr 96-2] Abrial J.R., *Extending B without changing it, for Developing Distributed Systems*, in First B conference, H.Habrias editor, 169-190, Nantes, 1996.

[Abr,Mus 97] Abrial J.R., Mussat L., *Specification and Design of a Transmission Protocol by successive Refinements using B*, in Mathematical Methods in Program Development, Edited by M. Broy and B. Schieder, Springer-Verlag, 1997.

[Abr,Mus 98] Abrial J.R., Mussat L., *Introducing Dynamic Constraints in B*, Edited by Didier Bert, Lecture Notes in Computer Science 1393, 83-128, Springer, 1998.

[Bac 80] Back R.J., *Correctness preserving program refinements : Proof theory and applications*, Mathematical Center Tracts, Volume 131, Amsterdam, 1980.

[Bac 88] Back R.J., *A calculus of refinements for program derivations*, Acta Informatica 25, 593-624, 1988.

[Bac 89] Back R.J., *A Method for Refining Atomicity in Parallel Algorithms*, in PARLE Conference on Parallel Architectures and Languages Europe, 366, 199-216, 1989.

[Bac,Wri 90] Back R.J., J. Von Wright, *Refinement Calculus, Part 1 : Sequential Nondeterministic Programs*, in J.W. deBakker, W.P. deRoever, and G. Rozenberg editors, Lecture Notes in Computer Science 430, 42-66, Springer, 1990.

[Bac 90] Back R.J., *Refinement Calculus, Part 2 : Parallel and Reactive Programs*, in J.W. deBakker, W.P. deRoever, and G. Rozenberg editors, Lecture Notes in Computer Science 430, 67-93, Springer, 1990.

[Bac,Kur 83] Back R.J.R., Kurki-Suonio R., *Decentralization of Process Nets with Centralized Control*, in 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing, 131-142, ACM, 1983.

[Bac,Kur 88] Back R.J.R., Kurki-Suonio R., *Distributed cooperation with action systems*, in ACM Transactions on Programming Languages and Systems, vol. 10, No. 4, pages 513-554, ACM, 1983.

[Bac,Ser 89] Back R.J.R., Sere K., *Stepwise Refinement of Actions Systems*, in Mathematics of Program Construction , Lectures Notes in Computer Science 375, 115-138, Springer, 1989.

[Beh,Ben,Fai,Mey 99] Behm P., Benoit P., Faivre A., Meynadier J-M., *Meteor : A Successful Application of B in a Large Project*, in Jeannette M. Wing, Jim Woodcock, Jim Davies editors, Lecture Notes in Computer Science 1708, 369-387, Springer, 1999.

[Beh,Bur,Mey 98] Behm P., Burdy Lilian, Meynadier J-M., *Well defined B*, in Didier Bert editor, Lecture Notes in Computer Science 1393, 29-45, Springer, 1998.

[Bel,Dar,Jul,Kou 2000] Bellegarde F., Darlot C., Julliand J., Kouchnarenko O., *Reformulate dynamic properties during B refinement and forget variants and loop invariants*, In the proceedings of the 1st International Conference of B and Z users, edited by Jonathan P. Bowen, Steve Dunne, Andy Galloway, Steve King, Lecture Notes in Computer Science 1878, 209-229, Springer, 2000.

[Bur 99] Burdy L., *Traitement des expressions dépourvues de sens de la théorie des ensembles : Application à la méthode B*, Thèse de doctorat, CNAM Paris, 1999.

[But 96] Butler M.J., *Stepwise refinement of communicating systems*, Science of Computer Programming 27, 139-173, 1996.

[But,Wal 96] Butler M., Walden M., *Distributed System Development in B*, in First B conference, H.Habrias editor, 155-168, Nantes, 1996.

[Can,Mér 2000] Cansell D., Méry D., *Playing with Abstraction and Refinement for Managing Features Interactions*, in the proceedings of the 1st International Conference of B and Z users, edited by Jonathan P. Bowen, Steve Dunne, Andy Galloway, Steve King, Lecture Notes in Computer Science 1878, 148-167, Springer, 2000.

[Cha,Mis 88] Chandy K.M., Misra J., *Parallel program design : A foundation*, Adisson-Wesley, 1988.

[Dij 65] Dijkstra E.W., *Solution of a problem in concurrent programming control*, Communications of ACM Vol 8, Num 9, 569, 1965.

[Dij 71] Dijkstra E.W., *Notes on structured programming*, Structured Programming, E.D. Dahl, C. Hoare editors, Academic Press, 1971.

[Dij 76] Dijkstra E.W., *A Discipline of programming*, Prentice-Hall International, 1976.

[Fra 86] Francez N., *Fairness*, Springer-Verlag, Berlin, 1986.

[Flo 67] Floyd R. W., *Assigning meanings to programs*, in Proceedings of the Symposium on Applied Mathematics, American Mathematics Society, Symposia in applied Mathematics, Vol 19, 19-31, 1967.

[Ger 75] Gerhart S.L., *Correctness preserving program transformations*, in Proceedings of the second ACM conference of Principles of Programming Languages, 54-66, 1975.

[Gri 81] Gries D., *The science of programming*, Springer-Verlag, New York, 1981.

[Hai 82] Hailpern B.T., *Verifying Concurrent Processes Using Temporal Logic*, Lecture Notes in Computer Science 129, Springer-Verlag, 1982.

[Hai,Owi 83] Hailpern B.T., Owicki S.S., *Modular Verification of Computer Communications Protocols*, in IEEE Transactions on Communications, COM 31, 1, 56-68, 1983.

[Hen 84] Hehner E.C.R., *Predicative programming. Part I*, Communications of ACM 27, 2, 134-143, 1984.

[Hen 84] Hehner E.C.R., *Predicative programming. Part II*, Communications of ACM 27, 2, 144-151, 1984.

[Hoa 69] Hoare C.A.R., *An Axiomatic Bases For computer Programming*, Communications of ACM 12,10, 576-580, 1969.

[Hoa 85] Hoare C.A.R., *Communicating Sequential Processes*, Prentice-Hall. 1985.

[Jon 86] Jones C.B., *Systematic software development using VDM*, Prentice-Hall. 1986.

[Kro 87] Kröger Fred, *Temporal Logic of Programs*, EATCS Monographs on Theoretical Computer Science, Volume 8, Springer-Verlag, New York, 1987.

[Lamp 80] Lamport L., *Sometimes is Sometimes "not Never"- On the Temporal Logic of Programs*, in Proceedings of the 7th ACM Symposium on the Principles of Programming Languages, ACM, 174-185, 1980.

[Lamp 83] Lamport L., *What good is temporal logic*, in Proceedings of information processing 83, Elsevier Science Publishers, IFIP, 1983.

[Lamp 89] Lamport L., *Reasoning About Nonatomic Operations*, in Proceedings of the 10th Annual Symposium on Principles of Programming Languages, ACM, New York, 28-37, 1989.

[Lamp 89] Lamport L., *A simple approach to specifying concurrent systems*, in Communications of the ACM, Vol. 32, No 1, 32-45, 1989.

[Lamp 91] Lamport L., *The Temporal Logic of Actions*, DEC SRC Rep. 57 Palo Alto, California, 1991.

[Lam,Sha 90] Lam S.S., Shankar U., *A Relational Notation for State Transition Systems*, in IEEE Transactions on Software Engineering, Vol 16, No 7, 755-775, 1990.

[Led 93] Ledru Y., *Developing Reactive Systems in a VDM Framework*, Science of Computer Programming 20, 51-71, 1993.

[Lop 96-1] Lopez N., *Construction de la specification formelle d'un systeme complexe*, in First B conference, H.Habrias editor, 63-119, Nantes, 1996.

[Lop 96-2] Lopez N., *Construction de la specification formelle d'un systeme complexe*, Memoire d'ingenieur CNAM, 1996.

[Lop 99] Lopez N., *An 'event based B' industrial experience*, in the proceedings of the B user Group Meeting, edited by Ken Robinson, Applying B in an industrial context, World Congress on Formal Methods 1999.

[Lop,Sim,Don 2000] Lopez N., Simonot M., Donzeau-Gouge V., *Deriving software specifications from event-based models*, in the proceedings of the 1st International Conference of B and Z users, edited by Jonathan P. Bowen, Steve Dunne, Andy Galloway, Steve King, Lecture Notes in Computer Science 1878, 209-229, Springer, 2000.

[Lyn,Tut 87] Lynch N.A., Tuttle M.R., *Hierarchical correctness proofs for distributed algorithms*, in 6th ACM Symposium on Principles of Distributed Computing, 137-151, 1987.

[Man,Pnu 84] Manna Z., Pnueli A., *Adequate proof principles for invariance and liveness properties of concurrent programs*, in Science of computer programming, Vol 4, 257-289.

[Man,Pnu 92] Manna Z., Pnueli A., *The Temporal Logic of Concurrent and Reactive Systems*, Springer-Verlag, New York, 1992.

[McCa 63] McCarthy J., *Towards a mathematical theory of computation*, in Proceedings of IFIP congress, North Holland Pub. Co., Amsterdam, 1963.

[Mil 80] Milner R., *A Calculus of Communicating Systems*, in Lectures Notes in Computer Science, Vol 92, Springer Verlag, 1980.

[Morg 88] Morgan C., *The Specification Statement*, in ACM Transactions on Programming Languages and Systems, Vol 10, No 3, 403-419, 1988.

[Morg 90] Morgan C., *Programming from Specifications*, Prentice-Hall International. 1990.

[Morr 87] Morris J.M., *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, Science of Computer Programming 9, 287-306, North-Holland, 1987.

[Morr 89] Morris J.M., *Laws of data refinement*, Acta Informatica 26, 287-308, Springer-Verlag, 1989.

[Owi,Gri 76] Owicki S., Gries D., *Verifying properties of parallel programs : An axiomatic approach*, in Communications of ACM, Vol 19, No 5, 1976.

[Owi,Lam 82] Owicki S., Lamport L., *Proving liveness properties of concurrent programs*, in ACM Transactions on Programming Languages and Systems, Vol 4, No 3, 455-495, 1982.

[Pnu 77] Pnueli A., *The temporal logic of programs*, in Proceedings of the 18th ACM Symposium on the Foundations of Computer Science, ACM, New York, 46-57, 1977.

[Pnu 79] Pnueli A., *The temporal semantics of concurrent computation*, Lecture Notes in Computer Science 70, 1-20, Springer-Verlag, 1979.

[Sch 99] Ouvrage Collectif, Coordination Schnoebelen P., *Vérification de Logiciels. Techniques et outils du model-checking*, Vuibert, Paris, 1999.

[Sch,Mel 82] Schwartz R.L., Melliar-Smith *From state machine to temporal logic : Specification methods for protocol standards*, in IEEE Transactions on communications, Vol 30, 12, 2486-2496, 1982.

[Sha 93] Shankar A. U., *An Introduction to Assertional Reasoning for Concurrent Systems*, in ACM Computing Surveys, Vol. 25, N. 3, 225-262, 1993.

[Sim 2001] Simonot Marianne, *Un modèle relationel du raffinement*, à apparaître.

[Spivey 92] Spivey J.M., *The Z notation : A reference manual*, Prentice Hall, second edition, 1992.

[Wir 71] Wirth N., *Program development of stepwise refinement*, in Communications of the ACM, 14, 221-227, 1971.

Index

abort, 14
false, 11
skip, 14
    Spécification, 24
true, 11

$[]$, Choix, 15
$||$, 26
    Spécification, 26
$\sqsubseteq^p$ Raffinement
    de Programmes, 17
$\sqsubseteq$ Raffinement
    de Spécifications, 20
SL, 22
    Post-Condition, 23
    Pré-Condition, 23
    Spécification Associée, 22
SP, 26
    Spécification Associée, 26
do od, Choix Itératif, 15
if fi, Choix Borné, 15
if G ; then S_1 else S_2 end
    Spécification, 24
if then else end, Conditionnelle, 15
var in end, 26
    Spécification, 26
while do end, Boucle, 15
WP, 12
:=, Affectation, 14
;, Séquence, 14

Affectation, 14
    Spécification, 24
Alarme, 108
Atteignabilité
    Propriété, 65, 84

Boucle, 15
    Condition de, 25
    Invariant, 25
    Spécification, 25
    Variant, 25

Choix, 15
    Itératif, 15
    Spécification, 23
Choix non Borné
    Spécification, 26
Commande de Contrôle, 109
Composant, 106
    Spécification de, 106
Composition Parallèle
    Spécification, 26
Conditionnelle, 15
    Spécification, 24
Constantes, 46
Correction
    Partielle, 10, 11
    Préservation, 17
    Totale, 10, 16

Dérivation
    de Programmes, 16

Elémentaire
    Spécification, 19
Equité, 85
Evénement, 99
    Raffinement, 100
    Spécification, 99
    Transformation, 105

Faisabilité
    d'un Programme, 13
    d'une Spécification, 20
    d'une Spécification

Définition, 20
FWS, 106
Comportement, 109
Description, 108
Propriétés, 109
Spécification
Formelle, 110
Spécification de, 106

Garde, 15
Spécification, 23

Hoare
Logique, 10
Triplé, 10

Instruction
Spécification Associée, 22
Invariant
de Boucle, 12, 25
de Collage, 31
de Module, 41

Logique
de Hoare, 10

Machine
Abstraite, 93
Implantation, 95
Miracle Exclu, 13
Module, 37
Composition, 79
Définition, 79
Définition, 38
Dérivation, 103
Implantation, 43
Invariant, 41
Partagé, 70
Implantation, 84
Raffinement, 39
Réflexivité, 40
Transitivité, 40
Spécification, 38
Utilisation, 47
Monitor, 84
Méthode
Formelle, 89
Méthode B, 89
Météor, 119, 129, 159

Non déterminisme
Externe, 38
Interne, 20, 21, 38
Module, 38
Réduction, 21, 100

Opération, 41
Atomique, 98
Auxiliaire, 66
Implantation de, 42
Nom, 41

Phase de Vol, 108
Post-Condition, 10
Renforcement, 21, 32
Spécification Elémentaire, 19
Procédure, 30
Spécification de, 30
Programme
Correction, 27
Définition, 28
Elémentaire
Spécification de, 30
Faisabilité de, 13
Spécification Associée, 26
Spécification de, 37
Séquentiel, 9
Propriété, 65, 84
Atteignabilité, 65, 84
Vivacité, 65, 84
Pré-Condition, 10
Affaiblissement, 21, 32
Plus Faible, 12
Spécification Elémentaire, 19
Weakest Precondition, 12

Raffinement
Algorithmique, 16
de Données, 16, 31
Définition, 32
de Module
Définition, 40
de Modules, 39

de Programmes, 17, 18
de Spécifications, 18
Définition, 20, 22
Monotonie, 18, 27
Relation, 18, 20
Réflexivité, 17, 22
Transitivité, 17, 22, 33

Signal d'alarme, 108
Spécification, 9
WP, 18
d'un Programme, 30
de Programmes Complexes, 37
Elémentaire, 19
Evénement, 99
Transformation, 103
Substitution
Généralisée, 89
Forme Normale, 90
Sureté
Propriété de, 41
Système, 101
Composant, 106
Concurrent, 71, 97, 98
Distribué, 97
Observation de, 99
Raffinement, 101
Décomposition, 102
Nouveaux Evénements, 102
Spécification, 101
Séquence, 14
Spécification, 24

Terminaison, 23, 85
Triplé
de Hoare, 10

Variable
Abstraite, 43
Auxiliaire, 54
Théorème, 58, 59
de Contrôle, 54
Historique, 54
Locale, 43
Spécification, 26
Prophécie, 54
Stable, 83
Variant
de Boucle, 25
Vivacité
Propriété, 65, 84
Vérification
de Programmes, 10

RESUME : Nous proposons une méthodologie de spécification et de construction de systèmes qui transforme la spécification *événementielle* d'un système en une spécification *pré-post*, ce qui permet d'effectuer le processus de développement à l'aide de techniques connues de dérivation de programmes par raffinements et sans rompre le processus de raisonnement formel. Le rôle de la spécification événementielle est de décrire les interactions du système avec son environnement. Le résultat de la transformation est un module qui fournit les moyens de communication du système.

Pour prendre en compte les aspects concurrents et distribués d'un système, nous proposons la notion de module partagé. Nous utilisons des variables auxiliaires, non implantées, pour exprimer des propriétés abstraites et pour faire la preuve de correction des implantations effectuées.

Pour répondre au besoin de réutilisation de modules existants, nous proposons une technique pour compléter une spécification à l'aide de variables auxiliaires afin de l'adapter à la solution de problèmes particuliers, tout en préservant l'implantation fournie initialement.

MOTS CLES : Spécification Formelle, Spécification du Système, Spécification des Composants, Evénement, Opération Atomique, Système Concurrent, Système Distribué, Interactions avec l'Environement, Module Partagé, Raffinement, Implantation.

ABSTRACT : We propose a specification and development method which allows the transformation of an event based system specification into a pre-post specification so that the implementation process can be achieved using the well known refinement techniques for building sequential programs. The role of the event based specification is to describe interactions between the system and its environment. The transformation yields a module specification which provides the communication mechanisms of the system.

Concurrent and distributed aspects are taken in account by means of shared modules. Auxiliary variables are used to express abstract properties and to prove the correctness of component implementations. They are also used to complete specifications of existing components so that these components can be reused in the implementation of new systems.

MOTS CLES : Formal Specification, System Specification, Component Specification, Event, Atomic Operation, Concurrent System, Distributed System, Environment Interaction, Shared Module, Refinement, Implementation.