RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

الجمهــورية الجزائرية الديمقراطية الشعبية

MINISTERE DE L'ENSEIGENEMENT SUPERIEURE ET DE LA RECHERCHE SCIENTIFIQUE

وزارة التعليم العالي و البحث العلمي

UNIVERSITÉ AMAR TELIDJI LAGHOUAT

جامعة عمار ثليــجي الأغواط

FACULTÉ DES SCIENCES

DÉPARTEMENT DE MATHÉMATIQUE ET INFORMATIQUE

MÉMOIRE DE MASTER

Domaine : Mathématiques et informatiques
Filière: INFORMATIQUE
Option: Réseaux, systèmes et applications répartis (ReSar)

Thème:

SPÉCIFICATION DES SYSTEMES A L'AIDE DE MÉTHODE FORMELLE B

Présenté par:
BELLI ZOUBIDA

Soutenu devant le jury compose de:
Mr. BELABBASI Amel Université de Laghouat (Présidente)
Mr DJOUDI Mohammed Université de Laghouat (Examinateur)
Mr. ABDELHAFIDI Zahra Université de Laghouat (Examinatrice)
Mr. BENDOUMA Taher Université de Laghouat (Rapporteur)

Année universitaire 2011/2012

REMERCIEMENTS

Je voudrais en tout premier lieu adresser mes plus chaleureux remerciements à Mr.Bendouma.T pour avoir accepté de diriger mon mémoire.

Je suis très reconnaissante à Melle. Amel BELABBASI et Mr. DJOUDI Mohammed et Melle. ABDELHAFIDI Zahra d'avoir accepté d'être rapporteurs de ce mémoire.

DEDICACES

A ma famille, mes amis, et

à tous ceux qui me connaissent

ملخص:

في هذا البحث، نسلط الضوء على استعمال الطرق الرياضية القطعية في هندسة البرامج المتعلقة بالأنظمة الموزعة، هته الطرق التي بالرغم من تكاليفها الباهظة إلا أن استعمالها قد يوفر لنا الأمان و الثقة في برامجنا.

و من بين هته الطرق التي رأينا أنها فعالة، الطريقة القطعية B ،وهي عبارة عن طريقة و لغة برمجة في آن واحد التي عمل على إيجادها الأستاذ جين ريموند أبريال، و تقدم الطريقة العديد من الإمكانيات و المزايا التي تسهل عملية التخصيص و إثبات و فحص و مراجعة البرامج التي نصممها خاصة ما تعلق بالأنظمة الموزعة التي وجدنا أن امتداد الطريقة B و هو Event-B ، هو أحسن حل لان هته الأنظمة مبنية على أساس الحوادث Evénements ،و هذا ما يقودنا إلى الجزم أن الطريقة القطعية Event-B هي الأفضل في عائلتها (من بين الطرق القطعية B).

كما نختم البحث بتطبيق الطريقة القطعية B ،على بعض الأمثلة المعروفة في مجال الأنظمة الموزعة، مثل خوارزمية الانتخاب (الاختيار)، و خوارزمية المنتج/المستهلك.

كلمات مفتاح: الطرق القطعية، الطريقة Event-B، الأنظمة الموزعة، خوارزمية الانتخاب، خوارزمية المنتج/المستهلك

RÉSUMÉ

Dans ce document nous avons discuté de l'utilisation des méthodes formelles pour les programmes et les protocoles concernant les systèmes répartis. Ces méthodes malgré qu'elle soient couteuse donnent sureté et fiabilité aux programmes et protocoles.

Parmi ces méthodes, la méthode formelle B inventé par Jean-Raymond Abrial, donne des possibilités de spécifier, vérifier et valider les programmes et les protocoles pour les systèmes répartis.

Dans notre travail nous présentons l'état de l'art des méthodes formelles en particulière la méthode B et l'appliquer sur des algorithmes classiques connus dans les systèmes répartis : Algorithme d'élection et l'algorithme de producteur/consommateur.

Mots-clé : méthodes formelles, méthode Event-B, systèmes répartis, algorithme élection d'un leader, algorithme producteur /consommateur

ABSTRACT

In this paper we discussed the using of formal methods in distributed systems, this method is so expensive but it's give us the safety, efficiency and reliability in our programs and protocols.

And from this different methods those so efficiencies , we find the B formal method invented by Jean-Raymond Abrial, this method give the possibilities to specify, verify, and validate the programs even protocols. And more, Event-B is the extension of B method; this last is the best from its family in B method.

We finished this paper by the application of these methods with some algorithms of distributed system. It's the election leader algorithm and producer/consumer algorithm.

Keywords: formal methods, Event-B method, distributed system, election leader algorithm, producer /consumer algorithm

TABLE DES MATIERES

Dédicace	iii
Remerciements	ii
Résumé	iv
TABLE DES MATIERES …………………………………………………………………………..	v
LISTE DES FIGURES ……………………………………………………………………………	viii
LISTE DES TABLEAUX …………………………………………………………………………	x
INTRODUCTION GÉNÉRALE ………………………………………………………………...	01
1. Chapitre 1 : Les méthodes formelles ………………………………………………………	02
1. Introduction ……………..………………………………………………………………..	03
2. Histoire ……………………………………………………………………………………….	03
1. Erreur logicielle Mortelle ………………………………………………………………	03
2. Pertes financiers colossales ………………………………………………………………..	04
3. Motivation ……………………………………………………………………..................	05
3. Classifications des Méthodes formelles ……………………………………………………	05
1. Méthodes non-formelles (informelles) ………………………………………………………..	05
2. Méthodes semi-formelles …………………………………………………………………….	06
3. Méthodes formelles ………………………………………………………………..................	09
4. Les Langages orientés modèles ……………………………………………………………...	09
1. Langage Z ……………………………………………………………………………………….	09
5. Les langages orientés propriétés ………………………………………………………………	11
1. Langage CCS ……………………………………………………………………...................	11
2. Langage LOTOS ……………………………………………………………….....................	12
6. Notions de bases ………………………………………………………………………..	15
1. La preuve ……………………………………………………………………………………….	15
2. La vérification ………………………………………………………………………………..	16
3. La validation ………………………………………………………………………................	16
4. Raisonnement formel ………………………………………………………………………..	16
5. Vérification de modèles (Model Checking) …………………………………………….	16
7. Les avantages et les limites des méthodes formelles ………………………...………..	16
1. Les avantages des méthodes formelles………………………………………………..	16
2. Les limites des méthodes formelles………………………………………………………	17
8. Les méthodes formelles pour les systèmes répartis et Pour quoi la méthode formelle B ? ……………………………………………………………………...................	17
1. Les méthodes formelles pour les systèmes répartis……………………………………..	17
2. Pour quoi la méthode formelle B ? …………………………………………………....	18
9. Conclusion …………………………………………………………………………………	19
2. Chapitre 2 : La méthode formelle B ……..……………………………………………....	20
1. Introduction ……………………………………………………………………………..	21
1. Le langage B ………………………………………………………………………………	21
2. La méthode B ……………………………………………………………….................	21
2. Etat de l'art de la méthode B……………………………………………………………..	22
1. Historique …………………………………………………………………...................	22
2. Diffusion de la Méthode B dans le monde ………………………………….............	22
3. Les utilisateurs de la Méthode Formelle B ………………………………..............	23
3. Versions de langages et outils de la Méthodes B ……………………………………..	25
1. Classique ………………………………………………………………………………..	25
2. Event-B …………………………………………………………………………………..	26
3. Outils de langage B ………………………………………………………................	26
4. Notions de base de langage B ………………………………………………………....	26

1. Notion de Langage ……………………………………………………………………………..	26
2. Notion machine abstraite ……………………………………………………………………...	27
3. Notion de Raffinage ……………………………………………………………………………	27
5. Syntaxe des données d'une "machine abstraite" ………………………………………....	28
1. Les clauses "MACHINE" et "SETS" …………………………………………………………	28
2. La clause "DEFINITIONS" ………………………………………………………................	29
3. La clause "VARIABLES" ………………………………………………………................	29
4. La clause "INVARIANT" ………………………………………………………………………	29
5. La clause "INITIALISATION" ……………………………………………………………….	29
6. Expressions de la théorie des ensembles de B ………………………………………..	29
7. La clause "PROPERTIES" …………………………………………………………………..	29
6. Syntaxe des opérations d'une "machine abstraite" ……………………………………...	30
1. En-tête d'une opération ……………………………………………………………………….	30
2. Corps d'une opération ………………………………………………………................	31
7. Implantation …………………………………………………………………………………….	34
1. Restrictions ……………………………………………………………………………………	34
2. La clause IMPORTS …………………………………………………………………………	35
3. La clause VALUES …………………………………………………………………………..	35
4. La substitution WHILE ……………………………………………………………………..	36
8. Le langage Event-B …………………………………………………………………………….	37
1. Notions général ………………………………………………………………………………	37
2. Modèle du système avec le langage Event-B …………………………………………..	38
3. Les outils ProB et Rodin …………………………………………………………………….	39
4. Les avantages de Event-B ……….…………………………………………................	40
5. Les limites de Event-B…… ………………………………………………………………….	40
6. Différence entre Event-B et B classique ………………………………………................	40
9. Conclusion ……………………………………………………………………………………….	41
3. Chapitre 3 : Logiciels et outils B utilisés …………………………………………………..	42
1. Introduction ………………………………………………………………………………………	43
1. L'Atelier B ………………………………………………………………………………………	43
2. Le ProB …………………………………………………………………………………….....	43
2. Atelier B ……………………………………………………………………………………..	44
1. Lancement de l'Atelier B et création d'un nouveau projet …………………………....	44
2. Création d'une machine abstraite ………………………………………………..……………	44
3. Machine abstraite et implémentation ……………………………………………………….	46
4. Le prouveur interactif …………………………………………………………….................	48
5. Machine abstraite de réservation ……………………………………………………………	51
3. ProB ……………………………………………………………………………………………...	52
1. L'animateur ProB et contrôleur du modèle pour la méthode B …………………………	53
2. Exemple de system B-événementiel …………………………………………….................	58
4. Rodin ……………………………………………………………………………………………	59
1. L'éditeur de Event-B ………………………………………………………………………	59
2. L'éditeur structurel de Event-B	60
3.Assistants(Wizards) ……………………………………………………………................	63
4.La perspective de preuve …………………………………………………………………	67
5. Conclusion …………………………………………………………………………………….	69
4. Chapitre 1 : Applications et discussion des résultats ………………………………………..	70
1. Introduction …………………………………………………………………………………….	71
2. Un protocole de communication simple ………………………………………………………	71
1. Principe de protocole …………………………………………………………………………..	71
2. Résultat de la spécification …………………………………………………………………	71
3. Discussion de résultat ……………………………………………………................	72
3. Algorithme d'élection d'un leader ………………………………………………………………	73

1. Principe de L'algorithme de l'élection d'un leader ……………………………………..	74
2. Résultat de la spécification ………………………………………………………………	74
3. Discussion de résultat …………………………………………………………................	76
4. Algorithme de consommateur et producteur ………………………………………….	79
1. Principe de L'algorithme de producteur-consommateur …………………................	80
2. Décomposition du système ………………………………………………………………	80
3. Résultat de la spécification ………………………………………………………………	81
5. Conclusion ………………………………………………………………………………	82
CONCLUSION GENERALE …………………………………………………………………..	83
BIBLIOGRAPHIE …………………………………………………………………………………	85
ANNEXES :………………………………………………………………………………………	88
ANNEXE A : Interview Avec Jean-Raymond Abrial: L'inventeur de la méthode formelle B ..	89
ANNEXE B : Eléments du langage de la méthode B ………………………………………..	90
ANNEXE C : Codes sources des algorithmes ………………………………………………..	92

LISTE DES FIGURES

1.1 - Représentation d'un objet UML ………………………………………………………………………	**07**
1.2 - Représentation d'une Association par UML ……………………………………………………..	**08**
1.3 - Représentation d'une Agrégation par UML ……………………………………………………...	**08**
1.4 - Exemple d'application de langage Z.………………………………………………………………	**09**
1.5 - L'initialisation de *Table* dans Z ………………………………………………………………….	**10**
1.6 - structure générale d'un schéma ………………………………………………………………….	**10**
1.7 - Schéma de tenir une fourchette droite ………………………………………………………….…	**10**
1.8 – Exemple spécification par LOTOS ……….………………………………………………………	**15**
2.1 - Diffusion de la méthode B dans le monde et L'utilisation des méthodes formelles dans l'Algérie...	**23**
2.2 - Principe général du raffinement …………………………………………………………	**28**
2.3 - Relation entre un contexte et une machine ……………………………………………..	**38**
3.1 - La machine abstraite *Rech_Element* ………………………………………………….	**44**
3.2 - La machine abstraite *Rech_Maximum* ……………………………………………..	**45**
3.3 - Le statut de la machine abstraite *Rech_Maximum* …………………………………….	**46**
3.4 - La machine abstraite *Operations* …………………………………………………...	**46**
3.5 - La machine abstraite *Operations_i* ………………………………………………...	**47**
3.6 - Le statut de la machine abstraite *Operations_i* ………………………………………..	**47**
3.7 - La machine abstraite *Exemple1* ……………………………………………………..	**48**
3.8 - Le statut de la machine abstraite *Exemple1* …………………………………………..	**49**
3.9 - Un extraire de Exemple1.po de la machine abstraite *Exemple1* ……………………….	**49**
3.10 - Utilisation La commande "*pr*" sur la machine abstraite *Exemple1* ……………………..	**49**
3.11 - La machine abstraite *Exemple2* …………………………………………………….	**50**
3.12 - Le statut de la machine abstraite *Exemple2* ………………………………………….	**50**
3.13 - Utilisation La commande "*pr*" sur la machine abstraite *Exemple2* ……………………..	**51**
3.14 - La machine abstraite *Reservation* ……………………………………………………	**51**
3.15 - Obligations de preuves de *Reserver* ………………………………………………….	**52**
3.16 - Obligations de preuves d'*Annuler* ……………………………………………………...	**52**
3.17 - Option d'animer un spécification avec ProB(Tcl/Tk) ……………………………………..	**52**
3.18 - Self-checker ……………………………………………………………………………..	**53**
3.19 - La machine abstraite *Rech_Maximum* ………………………………………………….	**54**
3.20 - Diagramme de la machine abstraite *Rech_Maximum* ……………………………………..	**54**
3.21 - Après double clic sur *initialise_machine* ……………………………………………...	**55**
3.22 - View Current State de la machine *Rech_Maximum* ……………………………………..	**55**
3.23 - La machine abstraite *Etage.mch* ……………………………………………………...	**56**
3.24 - Erreur de La machine abstraite *Etage.mch* ……………………………………………….	**56**
3.25 - La machine abstraite *Etage_a.mch* ……………………………………………………...	**57**
3.26 - Une violation de l'invariant dans *Etage_a.mch* ……………………………………………..	**57**
3.27 - Dessin informel représentant les comportements attendus ……………………………….	**58**
3.28 - La machine abstraite *CanalComm* ……………………………………………………...	**58**
3.29 - L'éditeur de l'Événement-B ………………………………………………………………...	**59**
3.30 - Ajouter un nouvel élément de la modélisation …………………………………………..	**59**
3.31 - L'éditeur de l'Événement-B Structuré ………………………………………………….	**60**
3.32 - Les Dépendances tabulent de l'éditeur de l'Événement-B d'un contexte ……………..	**60**
3.33 - Fenêtre de La clause EXTENDS(Include) ………………………………………………..	**61**
3.34 - Les Dépendances tabulent de l'éditeur de l'Événement-B d'une machine ……………	**61**
3.35 - La tabulation de la Synthèse de l'éditeur de l'Événement-B d'un contexte …………...	**62**
3.36 - La tabulation de la Synthèse de l'éditeur de l'Événement-B d'une machine …………	**62**
3.37 - La tabulation Pretty Print de l'Impression de l'éditeur de l'Événement-B ……………	**63**
3.38 - Assistants pour les machines et les contextes ……………………………………….	**63**

3.39 - Nouvel opérateur assistant des ensembles ……………………………………………….	**63**
3.40 - Nouvel opérateur assistant des ensembles énumérés ………………………………...	**64**
3.41 - Nouvel assistant des constantes ……………………………………………………...	**64**
3.42 - Nouvel Assistant des axiomes ……………………………………………………..	**64**
3.43 - Nouvel Assistant des variables …………………………………………………….	**65**
3.44 - Nouvel Assistant des invariants …………………………………………………..	**65**
3.45 - Nouvel Assistant d'un événement……………………………………………………...	**66**
3.46 - Présentation de la Prove Interactively …………………………………………………	**67**
3.47 - L'arbre de démonstrations ………………………………………………………………	**67**
3.48 - Contrôleur de démonstrations …………………………………………………………..	**68**
4.1 - La Statut de la machine abstraite *SysCommunication* ………………………………...	**71**
4.2 – Un extraire de *SysCommunication.po* de la machine abstraite *SysCommunication*	**72**
4.3 - La Statut de la machine abstraite *SysCommunication* après la démonstration par «pr »	**73**
4.4 - La Statut de la machine abstraite *SysCommunication* après la démonstration par Model Check	**73**
4.5 - Résultats de spécification après création *Cntxt1* et *MachineElectVainqueur* …………	**74**
4.6 - Résultats de spécification après création *Cntxt2* ………………………………………	**75**
4.7 - Résultats de spécification après la création de *MachineElectVainqueurRaf* …………..	**76**
4.8 - Obligation POs (axm6/WD) non prouvée ………………………………………………	**77**
4.9 - Après l'utilisation du prouveur p0 ………………………………………………………	**77**
4.10 - Obligation POs (axm5/WD) non prouvée ………………………………………………	**78**
4.11 - Après l'utilisation du prouveur p0 ………………………………………………………	**78**
4.12 - Obligation POs (axm4/WD) non prouvée ……………………………………………...	**78**
4.13 - Après l'utilisation du prouveur p0 ……………………………………………………..	**78**
4.14 - Obligation POs (inv2/INV) non prouvée ………………………………………………	**78**
4.15 - Après l'utilisation du prouveur p0 ……………………………………………………..	**79**
4.16 - Obligation POs (inv3/WD) non prouvée ………………………………………………	**79**
4.17 - Après l'utilisation du prouveur p0 ……………………………………………………..	**79**
4.18 – Variables et invariants partagés …………………………………………………………...	**80**
4.19 - Résultats de spécification après la création de *ProdCon* ……………………………..	**81**
4.20 - Résultats de spécification après la création de *Producteur et Consommateur* ………	**82**

LISTE DES TBLEAUX

1.1- **La liste des expressions de comportement de base LOTOS ……………………………………………….**	**15**
2.1 - **Les différentes actions de raffinage sur les données du système et sur les traitements …………….**	**27**
2.2 - **La syntaxe des données d'une machine abstraite ………………………………………………………..**	**28**
2.3 - **Notations relationnelles dans Event-B ……………………………………………………………………**	**37**

INTRODUCTION GÉNÉRALE

Les méthodes formelles survenu type particulier des techniques mathématiques qui sont utilisé pour la spécification, la vérification et la validation de logiciels. Les méthodes formelles sont basées sur l'informatique théorique, tels que les langages formels, la théorie des automates, la sémantique du programme, le calcul de logique, types de données algébriques… etc.

Comme pour les langages de programmation, les méthodes formelles peuvent également être classées selon leurs "degré de formalité": méthodes informelles, méthodes semi-formelles, méthodes formelles.

Dans ce mémoire, nous discutons de l'utilisation des méthodes formelles dans les systèmes répartis en utilisant la méthode B avec son extension Event-B.

Le présent mémoire est structuré comme suit:

Le premier chapitre présente des généralités concernant les méthodes formelles. Le deuxième chapitre introduit la méthode formelle B et ses extensions et nous détaillons la méthode Event-B. Le chapitre 3 discute des outils qui associes pour améliorer et faciliter l'utilisation de la méthode B notamment ceux de la méthode Event-B. Chapitre 4 contient quelques applications concernant des problèmes connus dans les systèmes répartis : un système de communication simple, le problème de l'élection, et le problème de consommateur /producteur.

Le mémoire se termine par une conclusion de ce mémoire, on va résumant les travaux réalisés durant toutes nos études ainsi que des possibles études générées sous le nom des méthodes formelles dans les systèmes répartis.

Chapitre 1

LES MÉTHODES FORMELLES

1.Introduction

Aujourd'hui de nombreux codes écrits rapidement au fil des besoins et sans autre validation que quelques tests expérimentaux sont en service dans tous les domaines de l'activité humaine.

Toute personne ayant déjà manipulé des logiciels informatiques a déjà rencontré des défaillances, ou "bugs". Ces défaillances sont en général causées par une erreur du concepteur du logiciel, lequel n'a pas pu identifier malgré une ou plusieurs relectures attentives. Et pour cause, tout développeur sait que plus son programme devient gros, plus il est difficile de s'assurer que celui-ci possède le comportement souhaité dans toutes les situations et de le concevoir sans erreur [1]. Pour ces raisons se sont imposées on utilise les méthodes formelles.

Dans ce chapitre nous montrons quelques histoires sur les incidents concernant des défaillances (Guerre de golf, Ariane 5..), et puis nous citons quelques motivations que possèdent les méthodes formelles, et après nous avons besoin de connaitre bien quel sont les classifications de ces méthodes. Le chapitre se termine par les avantages et les limites d'utilisation des méthodes formelle.

2.Histoire

Nous allons dans cette section citer quelques événements historiques parmi plusieurs qui ont poussé les industries à mettre on ouvre des moyens pour tester, valider et même vérifier le programme utilisé. Ces exemples sont des erreurs logicielles, engendrant des pertes financières graves ainsi que des pertes humaines, ceci implique l'augmentation des niveaux de sûreté des programmes.

2.1. Erreur logicielle Mortelle

Pendant la guerre de Golfe en février 1991, Un missile anti-missile ne parvient pas à détecter et intercepter un missile Scud Irakien, tuant 28 soldats Américains. L'enquête faite par une commission conclut à une erreur logicielle dans le calcul du temps de parcours. L'horloge interne du système qui calculait ce temps en dixième de seconde, où : 1/10 s'écrit 0.1 en décimal et en binaire 0.0001100110011001100110011... Aussi, on voit que le nombre 1/10 n'a pas d'écriture finie dans le système binaire, et le stockage dans le système introduisait une erreur de 0.000000000000000000000011001100... en binaire sur un registre de 24 bits soit environ 0.000000095 en décimal. Le système avait 100 heures de fonctionnement produisant une erreur de 0.34 secondes (0.000000095*100*60*60*10) de retard ce qui laisse le temps au missile Scud de parcourir de 500 mètres sans détection par l'anti-missile.

2.2. Pertes financières colossales

2.2.1. Explosion lanceur Ariane 5

Après 40 secondes de vol, le lanceur Ariane 5 explose le 4 Juin 1996. C'était une erreur logicielle à cause d'un composant développé pour Ariane 4 et transféré à Ariane 5 sans faire de spécifications ou tests complémentaires. Le code contenait une conversion d'un nombre flottant codé sur 24 bits (correspondant à la vitesse horizontale de la fusée) en un nombre entier codé sur 16 bits. Cette conversion était correcte dans le cas d'Ariane 4(où le nombre flottant ne dépassait jamais 32768 sur 16 bits) compte tenu des trajectoires de vol possibles pour Ariane 4.

[**The internal SRI (Inertial Reference System) software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected ... The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values...**]. [2]

Mais les trajectoires de vol possibles pour Ariane 4 et les trajectoires de vol avec Ariane 5, sensiblement différentes, notamment en phase de décollage, ont provoqué ce dépassement, rendant la valeur du nombre entier incohérent. Cette valeur a été transmise au calculateur de pilotage, un comportement anormal de la fusée et obligeant son auto-destruction.la fusée avait coûté un demi-milliard d'euros, sur un projet de sept milliards d'euros.

2.2.2. Écrasement de la sonde Mars Climate Orbiter

Le 23 septembre 1999, la sonde Mars Climate Orbiter s'écrase sur le sol martien .Ses moteurs de freinage ne sont mis en route qu'à une altitude de 57 kilomètres, au lieu de 140 kilomètres prévus.

Une partie du programme avait été écrit par deux équipes, une équipe utilisant une unité de mesure anglo-saxonne pour la force du moteur de freinage (pound-force), la deuxième équipe travaillant dans le système métrique international(Newton).la perte financière s'élève à 160 millions d'euros.

2.2.3. Bug de l'an 2038

Le bug de l'an 2038 devrait affecter les systèmes Unix en 2038; ce bug est un problème similaire au bug de l'an 2000 qui pourrait endommager le fonctionnement d'ordinateurs 32 bits au environ du 19 janvier 2038, et plus précisément à 3 h 14 min 7 s, temps universel.[3]

Le problème retient des logiciels qui utilisent la représentation POSIX du temps, dans lequel le temps est représenté comme un nombre de secondes depuis le 1er janvier 1970 à minuit (0 heure). Sur les ordinateurs 32 bits, la plupart des systèmes d'exploitation concernés représentent ce nombre comme un nombre entier signé de 32 bits, ce qui limite le nombre de secondes à 231 - 1, soit 2 147 483 647 secondes (01111111 1111111111111111 11111111 en binaire). Ce nombre maximum sera atteint le 19 janvier

2038 à 3 h 14 min 7 s (temps universel). Dans la seconde suivante, la représentation du temps _ bouclera _ (10000000 00000000 00000000 00000000 en binaire) et représentera -2 147483 648 en complément à deux, et ainsi, l'ordinateur affichera la date du 13 décembre 1901.

2.3. Motivation

Les méthodes formelles sont utilisées dans des domaines où des erreurs peuvent causer des risques en vies humaines, des dégâts financiers importants. Aussi, il est souhaitable qu'une spécification soit claire, non ambiguë, compréhensible, complète, et cohérente (sans contradictions).

Les descriptions en langue naturelle manquent souvent de précision. C'est la raison pour laquelle on préfère utiliser des méthodes formelles, même si cela est difficile à faire.

Pour ces raisons et d'autres, les méthodes formelles ont été utilisées dans les différents domaines de l'informatique telle que les réseaux, les systèmes distribues ou hybride, la sécurité des logiciels et autres.

Nous nous concentrons dans ce mémoire sur l'usage des méthodes formelles pour valider le bon fonctionnement de systèmes repartis.

3. Classifications des Méthodes formelles

Les méthodes de spécification formelles peuvent être catégorisées selon leur degré de formalité: Méthodes non-formelles, méthodes semi-formelles, méthodes formelles. [4]

3.1. Méthodes non-formelles

Ce sont les notations simples définies essentiellement dans le langage naturel, diagrammes simple, table simple... etc.

Une modélisation informelle est construite en langue naturelle avec ou sans règles de structuration. Son usage introduit des risques d'ambiguïtés car ni sa syntaxe, ni sa sémantique ne sont parfaitement définies. Pour réduire ces risques, deux approches parmi plusieurs ont été proposées.

- La première consiste à apporter un soin particulier à la rédaction peut par exemple se réduire à remplir des formulaires.
- La seconde est de restreindre la langue naturelle utilisée (généralement nommé une langue naturelle "contrôlée"). Ces langues contrôlées, limitent le lexique et la syntaxe de la langue naturelle pour diminuer le nombre de formulations possibles d'une spécification.

3.1.1. Points forts d'une modélisation non-formelle

A. Facilite la compréhension qui consiste à communiquer entre les différents acteurs de développements d'un logiciel ou d'un système (par exemple. utilisée dans l'écriture d'un cahier des charges).

B. L'utilisation de la langue naturelle ne requiert aucune formation particulière, d'où un le faible coût de formation pour savoir rédiger une modélisation informelle (notamment il ne diminue pas le travail de modélisation).

3.1.2. Points faibles d'une modélisation non-formelle

A. Le logiciel (ou le système) obtenu ne correspond pas aux attentes du client dans les deux cas suivants :
   - L'étendue de la langue naturelle et surtout les ambigüités (mauvaise compréhension du système) ;
   - Il peut exister un décalage entre les besoins du client et ceux compris par un malentendu entre concepteurs et développeurs.

B. Il existe de fortes possibilités d'erreurs dans la modélisation du système (car les ambigüités remettent tout raisonnement pour analyser ou vérifier la spécification).

3.2. Méthodes semi-formelles

Les méthodes semi-formelles sont les méthodes basées principalement sur des notations graphiques (diagramme de classes, modèle conceptuel de données MCD, diagramme de flots de données DFD, ...). Elle sont fondées pour modéliser les systèmes d'information généralement basé sur les graphes. Nous pouvons citer les méthodes suivantes : OMT[1], UML[2], MERISE[3] ...etc.

Deux modèles sont toujours proposés par ces méthodes. L'un décrit l'aspect statique, c'est-à-dire la structure des données et leurs relations. L'autre décrit les traitements qui ont lieu sur les données (l'aspect dynamique).

Certaines méthodes comportent d'autres modèles. Par exemple, on trouve dans la méthode OMT le modèle fonctionnel qui décrit la transformation des valeurs des données dans le système.

Il est important de noter que, malgré une terminologie propre (parfois) à chaque méthode, la sémantique des concepts des différentes méthodes reste très proche. Nous présentons maintenant, les principaux concepts de ces méthodes.

3.2.1. Entité

Ce concept a été proposé par CHEN P.P [5] dans le modèle E/A, « An entity is a thing which can be distinctly identified ». Et sa traduction en français[6]: « Une "entité" est une chose qui peut être distinctement identifiée ». Une personne donnée, une compagnie sont des exemples d'entité. Un ensemble d'entités regroupe les entités ayant les mêmes structures et comportements.

[1] Object Modeling Technique
[2] Unified Method Language
[3] Méthode d'Etude et de Réalisation Informatique de Systèmes d'Entreprise

Exemple. *p1* est un élément d'un ensemble de processus (entités) avec les attributs: Etat, voisinDroit, voisinGauche...etc. En utilisant les méthodes qui manipulent ces entités des processus: *executerCode()*, en*Attente()*, *stop()*, voir la figure 1.1 .

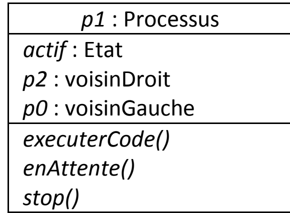


Figure 1.1 **-** Représentation d'un objet UML

3.2.2. Association

Pour faire le lien entre des entités, il est proposé également le concept d'association, « A relationship is an association among entities »[5], ce qui est traduit par « Une "association" est un lien entre entités ». L'association "père-fils" est définie entre deux entités personnes. CHEN a défini un ensemble d'associations comme étant une relation mathématique entre plusieurs entités. Des contraintes de cardinalité peuvent accompagner la description d'un ensemble d'associations. Ces contraintes expriment le nombre minimum /maximum de liens pour chaque entité.

Dans la méthode NIAM[1], un ensemble d'associations est appelé relation. Dans OMT et UML une association désigne l'ensemble d'associations[5]. Une association peut avoir des attributs.

Exemple. Dans la figure 1.2, l'association entre CYCLE et PROCESSUS dit que : un cycle contient au minimum un processus.

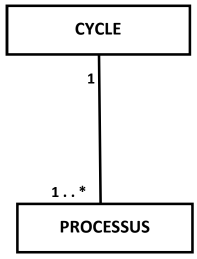


Figure 1.2 **-** Représentation d'une Association par UML

3.2.3. Agrégation

Une agrégation, dans OMT et UML est une association particulière dans laquelle les objets d'une classe sont les composants d'objets d'une autre classe. Ce concept exprime la relation "composé-composant" ou "partie-de".

[1] Nijssen Information Analysis Method (appelé aussi Modèle relationnel binaire)

Exemple. PROCESSUS est un composant de CYCLE (c'est-à-dire une partie de PROCESSUS) et un CYCLE est composé de un ou plusieurs PROCESSUS.

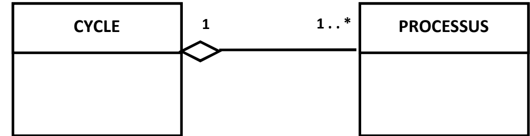


Figure 1.3 **-** Représentation d'une Agrégation par UML

3.2.4. Généralisation/Spécialisation

Le concept de généralisation/spécialisation a été également introduit par SMITH J.M., SMITH D.C.P [7]. Le concept de généralisation consiste à définir une entité (appelée super-entité) comme l'union de plusieurs autres entités (appelées sous-entités). La relation de généralisation signifie "est un" ou "est une sorte de».

Exemple1. CYCLE est une généralisation de "unidirectionnel" ou "bidirectionnel".

Exemple 2. *p1* est une spécialisation de l'entité PROCESSUS.

3.2.5. Points forts d'une modélisation semi-formelle

Les notations graphiques dans les méthodes semi-formelles permettent d'avoir une vision claire du système. Elles représentent le système à modéliser d'une manière à la fois intuitive et synthétique [8]. De ce fait, elles sont bien adaptées à la plupart des utilisateurs. Ces avantages ont contribué à répandre largement leur utilisation dans l'industrie.

3.2.6. Points faibles d'une modélisation semi-formelle

Ces méthodes souffrent du manque d'une sémantique à précise de différentes notations utilisées, ce qui entraîne dans la plupart de temps des **ambiguïtés**.

En outre, il est impossible de prouver la cohérence du système, ce qui limite la fiabilité des logiciels produits, sauf à augmenter notablement la phase de test. [8]

3.3. Méthodes formelles

Ce sont des méthodes basées sur des notations mathématiques (fonctions, relations,...), permettant l'obtention d'une spécification précise et concise et surtout analysable par des outils, donc ces méthodes sont des langages aussi.

Les méthodes formelles sont catégorisées [9], en deux grandes catégories. La première catégorie est basée sur les langages orientés modèles, et la deuxième basée sur les langages orientés propriétés.

Dans les deux prochaines sections, nous allons voir quelques langages de ces deux catégories de méthodes ainsi que quelques exemples concernant ces langages.

4. Les langages orientés modèles

Les langages orientés modèles basées sur la théorie des modèles, adaptées aux spécifications de systèmes d'information orientée vers une structure (Langage B, Z, Vienna Development Method VDM). Nous avons choisi de présenter Z au lieu VDM, car les schémas de Z sont plus lisibles que les spécifications textuelles de VDM [11].

4.1. Langage Z

Le langage Z, se basé sur deux notions principales. A savoir la notion d'ensemble, et la notion de schéma. On prend un problème très connu est le problème de philosophes qui sera la base de notre spécification.[11]

4.1.1. La notion d'ensemble

Les spécifications Z sont basées sur la notion d'ensemble. Un ensemble est une collection d'entités distinctes qui ont toutes le même type. Il fournit juste l'ensemble des entiers et des naturels. Pour disposer de plus de types, il faut introduire des ensembles de base. Introduire un ensemble de base assure juste qu'il existe un tel type.[8]

Exemple 1. La déclaration suivante introduit les ensembles de base etatPhilosophe et etatFourchette :

[etatPhilosophe, etatFourchette]

Exemple 2. On définit le schéma *Table*, basé sur le type [etatPhilosophe, etatFourchette].

```
┌─ Table ─────────────────────────────
│
│ phils : nbre ⟶ etatPhilosophe
│ fourch : nbre ⟶ etatFourchette
│
└─────────────────────────────────────
```

Figure 1.4 **-** Exemple d'application de langage Z

O ù :

- *nbre* == 1 .. N (N est le nombre de philosophes et les fourchettes)
- *etatPhilosophe ::= PENSE |aFourchDroite|MANGE*
- *etatFourchette ::= LIBRE | OCCUPÉ*

Il est à noter que quelque soit le schéma, on doit définir un état initial abstrait lors d'une première utilisation de système.

```
┌─ TableInit ─────────────────────────
│
│ Table
├──────────────
│ ran phils = {PENSE}
│ ran fourch ={LIBRE}
│
└─────────────────────────────────────
```

Figure 1.5 **-** L'initialisation de *Table* avec Z

4.1.2. La notion de schéma

Un schéma est une structure de spécification de données ou de traitements. Il est caractérisé par un nom. Un schéma est composé de deux parties : les déclarations (signatures) et les prédicats. Les déclarations constituent le lexique des objets locaux

aux schémas. Les prédicats précisent les contraintes portant sur les variables ou la relation entre les états initiaux et finaux d'une opération.

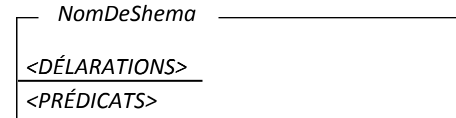


Figure 1.6 **-** structure générale d'un schéma

Dans Z, on a deux types de schéma, schéma d'état et schéma d'opération. Un schéma d'état consiste en la déclaration de variables et de prédicats qui contraignent ces variables.

Les schémas peuvent inclure d'autres schémas (Cela signifie que toutes les variables et les prédicats du schéma inclus deviennent des parties du schéma incluant) c'est-à-dire que les déclarations sont fusionnées et que les prédicats sont joints.

tenirFourchDroite

△ Table
n? : nbre

phils(n?) = PENSE [n? est la valeur entrée n]
fourch(n? **mod** *N + 1) =LIBRE*
phils' = phils $\oplus \{n? \mapsto aFourchDroite\}$
fourch' = fourch $\oplus \{n? \, \mathbf{mod} \, N + 1 \mapsto OCCUPÉ\}$

Figure 1.7 **-** Schéma de tenir une fourchette droite

- ✓ **Note1.** △, signifie qu'on a un raccourci d'un nouveau schéma contenant les déclarations du schéma *Table*
- ✓ **Note2.** Dans Z, les parties entre crochets ([...]) correspondent à des commentaires.

5. Les langages orientés propriétés

Les langages orientés propriétés basées sur des algèbres de processus, adaptées aux spécifications de systèmes concurrents orientés vers un événement on trouve parmi ces langages le langage CCS[1] et LOTOS[2],… .

5.1. Langage CCS

CCS a été défini par **Milner[12]**. CCS est fondé sur l'observation d'agents qui représentent une partie unitaire d'un système concurrent à modéliser.

5.1.1. Agent

Un agent peut donc être composé de plusieurs sous-agents. Les deux notions fondamentales de CCS sont "concurrence" et "communication".

La concurrence en CCS est caractérisée par l'indépendance des actions d'un agent par rapport aux autres agents du système. La communication des agents en CCS est de deux types : les actions peuvent agir à l'intérieur d'un agent ou bien interagir avec ses

[1] Calculus of Communicating Systems
[2] Langage Of Temporal Ordering Specification

agents voisins. Le comportement d'un système est défini en CCS par l'observation de ses actions.

Un agent composé de deux ports qui permettent de communiquer avec l'extérieur (port d'entrée et un port de sortie). Une action de label $\boldsymbol{a}$ entrant dans un agent est désignée par $\boldsymbol{a}$, cependant, une action sortante de même label est désignée par $\overline{a}$. Les deux actions a et $\overline{a}$ sont dites complémentaires.

5.1.2. Comportement d'un agent

Un agent est défini par une expression spécifiant les actions possibles et le comportement final de l'agent. La syntaxe du langage CCS est la suivante.

Exemple 1. La préfixation (.) permet d'associer une action à un agent résultant de l'exécution de cette action, l'action suivante représente l'agent $\boldsymbol{A}$ qui accepte une action $\boldsymbol{\alpha}$ et se comporte ensuite comme l'agent $\boldsymbol{A'}$.

$$\boldsymbol{A = \alpha \,.\, A'}$$

Les actions en CCS sont indéterministes et récursives.

Exemple 2. La sommation (+) permet de représenter deux comportements possibles, l'action suivante définie $\boldsymbol{C}$ par :

$$\boldsymbol{C = A + B}$$

Se comporte soit comme l'agent $\boldsymbol{A}$, soit comme l'agent $\boldsymbol{B}$.

Exemple 3. La composition de deux agents $\boldsymbol{A}$ et $\boldsymbol{B}$ est désigner par $\boldsymbol{A \mid B}$. Dans ce cas, les actions complémentaires sont synchronisées et deviennent des actions internes du résultat de la composition. Si les agents $\boldsymbol{A}$ et $\boldsymbol{B}$ sont définis par :

$$\boldsymbol{A = a \,.\, A' + \overline{c} \,.\, A''}$$

$$\boldsymbol{B = b \,.\, B' + c \,.\, B''}$$

Exemple 4. Les actions $\boldsymbol{a}$ et $\boldsymbol{b}$ précédentes restent indépendantes dans la composition $\boldsymbol{A \mid B}$. La transition d'état suivante est possible :

$$\boldsymbol{A \mid B \overset{a}{\rightarrow} A \mid B'}$$

Exemple 5. La composition des actions complémentaires $\boldsymbol{c}$ et $\boldsymbol{\overline{c}}$ est une action interne de l'agent $\boldsymbol{A \mid B}$, elle est désignée en CCS par $\boldsymbol{\tau}$:

$$\boldsymbol{A \mid B} \overset{\tau}{\rightarrow} \mathrm{A''} \mid \boldsymbol{B''}$$

5.1.3. Restriction

L'exemple de composition précédent n'impose aucune restriction sur les actions $\boldsymbol{c}$ et $\boldsymbol{\overline{c}}$: elles peuvent donc être exécutées dans la composition.

Exemple 1. Dans ce cas, il est possible d'agir sur la transition :

$$\boldsymbol{A \mid B} \overset{c}{\rightarrow} \mathrm{A''} \mid \mathrm{B}$$

Exemple 2. Le symbole de restriction (\) permet d'éviter ce type d'action. Dans ce cas, il n'est possible d'exécuter sur l'agent que les actions autres que $\boldsymbol{c}$ et $\boldsymbol{\overline{c}}$:

$$(\boldsymbol{A \mid B}) \setminus \{\boldsymbol{c}\}$$

5.2. Langage LOTOS

LOTOS, est un langage permettant de spécifier l'architecture et le fonctionnement de systèmes distribués. La définition formelle de LOTOS fait l'objet de la norme ISO[1] 8807 [ISO88].

LOTOS a été défini pour permettre la description des services et des protocoles de télécommunication en particulier pour les systèmes OSI[2]. En effet les organismes de normalisation, confrontés aux problèmes posés par l'emploi du langage naturel (même complété par des tables d'états) dans les normes [LS88], ont recherché des moyens d'expression mieux adaptés à leurs besoins. C'est ainsi que l'ISO et le CCITT[3] ont normalisé trois langages de description formelle : LOTOS, ESTELLE[4] et SDL[5].

5.2.1. Portes et signaux

Un signal ou action, la proposition effectuée par un comportement qui désire participer à un rendez-vous[13].

Un signal se compose d'une porte et une liste d'offres pour l'émission ou la réception de valeurs typées (dans la terminologie ISO pour LOTOS, le type d'une valeur ou d'une variable porte le nom de sorte).

Exemple 1. Le signal suivant signifie que l'on cherche à émettre simultanément, sur la porte ***OUTPUT***, les trois valeurs ***5, X1*** et ***X2*** :

OUTPUT !5 !X1 !X2

Exemple 2. De même, le signal suivant indique que l'on s'attend à recevoir simultanément, sur la porte ***INPUT***, trois valeurs réelles qui seront rangées dans les variables ***A, B*** et ***C*** :

INPUT ? A : REAL ?B : REAL ?C : REAL

Exemple 3. On peut combiner émissions et réceptions au cours d'un même rendez-vous et spécifier des conditions sur les valeurs émises ou reçues. Le signal suivant exprime que, sur la porte ***EXCHANGE***, on désire émettre la valeur ***5*** tout en recevant deux valeurs réelles ***X1*** et ***X2*** :

EXCHANGE !5 ? X1 : REAL ? X2 : REAL [X1 < X2]

5.2.2. Opérateurs sur les valeurs

Il existe plusieurs opérateurs LOTOS permettant de manipuler les valeurs et de créer des variables. Par exemple l'opérateur "*[...] —> . . .*" permet de conditionner l'exécution d'un comportement par une garde booléenne, il s'apparente à une instruction "***if . . . then . . .***".

L'opérateur "***let . . . in . . .*** " permet de donner un nom à une expression en définissant une variable.

L'opérateur "***choice ... [] ...***" permet de choisir, de manière non-déterministe, une valeur dans un domaine et de la nommer en définissant une variable. On peut l'utiliser,

[1] Organisation Internationale de Normalisation
[2] Open Systems Interconnection
[3] Comité Consultatif International pour la Téléphonie et la Télégraphie
[4] Extended Finite State Machine Language
[5] Specification and Description Language

entre autres, pour spécifier que la valeur exacte d'un résultat renvoyé n'est pas significative.

Exemple. L'exemple suivant, consacré au calcul des racines d'un polynôme du **2*nd*** degré, illustre l'emploi de ces opérateurs :

```
INPUT ?A:REAL ?B:REAL ?C:REAL;
    (
    let DELTA:REAL = (B ^ 2) - (4 * A * C) in
       (
       [DELTA > 0] ->
       (
        let X1:REAL = (-B - sqrt (DELTA)) / (2 * A) in
           let X2:REAL = (-B + sqrt (DELTA)) / (2 * A) in
             OUTPUT !2 !X1 !X2;
                   stop
        )
       [ ]
       [DELTA = 0] ->
         (
         let X0:REAL = -B / (2 * A) in
           OUTPUT !1 !X0 !X0;
                   stop
         )
       [ ]
       [DELTA < 0] ->
           (
           choice X1, X2:REAL []
              OUTPUT !0 !X1 !X2;
                    stop
              )
         )
    )
```

5.2.3. Processus

Il est possible de donner un nom à un fragment de programme LOTOS en définissant un processus. L'appel d'un processus se fait en donnant le nom du processus et la liste de portes passées en paramètres effectifs. Les appels récursifs sont permis et, grâce à eux, on peut exprimer les comportements cycliques.

Exemple. Le processus suivant, acquiert deux valeurs réelles sur la porte ***INPUT*** et renvoie leur minimum sur la porte ***OUTPUT***, après quoi il recommence indéfiniment , il suffit de reprendre le comportement défini précédemment en remplaçant "***stop***" par un appel récursif pour indiquer qu'après l'émission du résultat sur la porte "***OUTPUT***" le comportement ne s'arrête pas, mais repart dans l'état où il était initialement .

Note. LOTOS possède un opérateur "***exit***" qui permet à un comportement de se terminer en renvoyant comme résultat une liste de valeurs. Et Le mot-clé "***noexit***" signifiant que le processus ne retourne aucun résultat.

```
process MIN [INPUT,OUTPUT] : noexit :=
INPUT ? X1, X2 :REAL ;
     (
     [X1<= X2] ->
          OUTPUT !X1 ;
                 MIN [INPUT,OUTPUT]
     []
     [X2 <= X1] ->
          OUTPUT !X2 ;
                 MIN [INPUT,OUTPUT]
     )
endproc
```

5.2.4. Opérateurs parallèles

En LOTOS l'opérateur équivalent à l'opérateur "&" du shell se note " |||", il permet l'exécution simultanée de deux comportements sans aucune synchronisation entre eux, sauf en ce qui concerne la terminaison par "***exit***" : le processus qui se termine le premier attend l'autre.

Comme il n'existe aucune variable partagée entre les comportements qui s'exécutent en parallèle, on évite les problèmes liés à l'emploi de "&" (par exemple lorsque deux processus UNIX concurrents écrivent dans le même fichier).

LOTOS possède un opérateur notée " **|[G_1, ... G_n] |**", qui généralise l'opérateur " **|** " du shell puisqu'il autorise n canaux de communication au lieu d'un seul. L'opérateur " **|[G_1, ... G_n] |**" indique que les deux comportements auxquels il s'applique doivent fonctionner en parallèle, tout en se synchronisant par rendez-vous sur les portes ***G_1, ... G_n*** et sur ces portes-ci exclusivement (en outre la terminaison par "***exit***" est toujours synchrone).

Exemple. La commande shell suivante, qui lance l'exécution simultanée des processus ***P_1*** et ***P_2*** et, lorsqu'ils ont terminé, exécute les processus ***P_3*** et ***P_4*** en parallèle, la sortie de ***P_3*** servant d'entrée à ***P_4*** :

$$(P_1 \,\&\, P_2);\ (P_3 \mid P_4)$$

A comme équivalent en LOTOS :

$$(P_1 ||| P_2) >> (P_3[G] \mid [G] \mid P_4[G])$$

La structure typique de base LOTOS donné dans la figure suivant :(Exemple spécification par LOTOS).

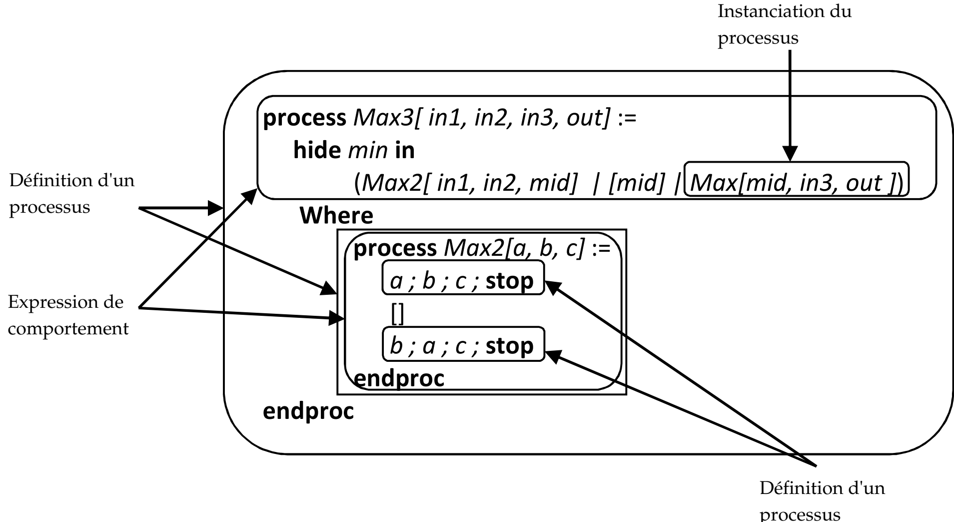


Figure 1.8 **-** Exemple spécification par LOTOS

La liste complète des expressions de comportement de base LOTOS est donnée dans le tableau suivant, il comprend tous les opérateurs de base LOTOS.

Nom	Syntaxe
Inaction	***Stop***
préfixe d'action	
- inobservable (interne)	***i ; B***
- observable	***g ; B***
Choix	B_1 [] B_2
composition parallèle	
- cas général	B_1 | [g_1, …, g_2] | B_2
- pure interleaving"Permission interne pure"	B_1 ||| B_2
- synchronisation complète	B_1 || B_2
Caché	***hide*** g_1, …, g_n ***in B***
instanciation d'un processus	$p[g_1, \ldots; g_n]$
terminaison réussite	***Exit***
composition séquentiel (permission)	B_1 >> B_2
Désactivation	B_1 [> B_2

Tableau 1.1 **–** Liste des expressions de comportement de base LOTOS

Les symboles ***B***, B_1, B_2 données pour toute expression d'un comportement et toute expression doit correspondre à un des formats donnés dans la colonne de la syntaxe.

6. Notions de bases

On distingue plusieurs approches pour vérifier la cohérence d'un système par rapport à sa spécification; ces approches différent selon le domaine.

6.1. La preuve

Elle consiste à démontrer de manièrer systématique, en utilisant les règles d'inférences d'une logique, un théorème de correction.

6.2. La vérification

Consiste à vérifier, de manière exhaustive en parcourant tous les états possibles du système, que le théorème est satisfait. Elle indique généralement ce que le programme ne doit pas faire.

Exemple. Montrer que le système (***S***) est correct par rapport à des propriétés(***P***) ***:S = P***

6.3. La validation

La validation d'une théorie ou d'outils est un problème abstrait et difficile, mais nécessaire[14], dans les méthodes formelles la validation consiste à vérifier la correction du programme vis-à-vis de sa spécification, c'est à dire prouver ou tester que le système fait ce qu'il est censé faire.

Exemple. Montrer que le système est correct par rapport aux spécifications informelles :

$$S \sim S_{informelle}$$

6.4. Raisonnement formel

Consiste à appliquer un système formel à une spécification (Raffinement de spécification, vérification des propriétés d'un système, validation par vérification,...etc.).

6.5. Vérification de modèles (Model Checking)

La vérification de modèles est une technique qui consiste à construire un modèle fini d'un système et à vérifier qu'une propriété cherchée est vraie dans ce modèle. Il y a deux façons générales de vérification dans le model-checking : vérifier qu'une propriété exprimée dans une logique temporelle est vraie dans le système, ou comparer (en utilisant une relation d'équivalence ou de pré ordre) le système avec une spécification pour vérifier si le système correspond à la spécification ou non.

Au contraire du theorem proving, le model-checking est complètement automatique et rapide. Il produit aussi des contre-exemples qui représentent des erreurs subtiles dans la conception et ainsi il peut être utilisé pour aider le débogage.

7. Les avantages et les limites des méthodes formelles

7.1. Les avantages des méthodes formelles

- Les spécifications formelles peuvent jouer un rôle fondamental dans la réutilisation d'une part en donnant précisément les fonctionnalités d'un module réutilisable et d'autre part en facilitant l'intégration d'autres modules lors du raffinement [15].
- Utilisables dans les Gros projets industriels (contrôle de processus, systèmes embarques...) et dans les développements linguiciels (comme les logiciel du traduction)[16], contrôle de trafic aérien, pilotage d'avion, systèmes médicaux...etc. Alors les méthodes formelles utilisable dans des variétés de domaines.
- Les méthodes formelles peuvent être employées pour donner une description du système à développer, à quelques niveaux de détail désirés. Cette description formelle peut être employée pour guider d'autres activités de développement.
- Développer des systèmes sûrs, fiables, robustes.

- Dans les systèmes embarqués (des appareils divers, d'usage courant) par exemple dans les cartes à puce.

7.2. Les limites des méthodes formelles

Il faut bien comprendre que dans l'informatique rien n'est parfait, quelque soit un logiciel ou bien matériel ou bien méthode de résolution .Les méthodes formelles ont des limites.

- Les méthodes formelles sont coûteuse (entre ***30 %*** et ***50 %*** du développement)[17] et elles n'ont n'a pas encore fait leur preuves.
- Les méthodes formelles sont difficiles à utiliser (pour certains logiciels), de plus elles deviennent très complexes lorsque la taille des problèmes devient importante.
- Des outils efficaces et utilisables limités sur de vrais systèmes existent actuellement. La question de la correction ces outils supportant les méthodes formelles se pose aussi.[18]

8. Méthodes formelles pour les systèmes repartis et pour quoi la formelle B

8.1. Méthodes formelles pour les systèmes répartis

Le model-checking est une technique puissante, largement utilisée pour vérifier le matériel, les systèmes embarqués et les logiciels séquentiels ou à mémoire partagée. En général les spécifications sont exprimées dans une logique temporelle propositionnelle et le système est représenté comme un graphe de transitions d'états (connu sur le nom de structure de Kripke).

Cependant, quand on travaille avec des systèmes distribues (des processus concurrents et communicants), les modèles basés sur les états ne sont pas bien appropriés (adéquats).

En l'absence d'une mémoire partagée, où les états peuvent être facilement identifiés par les états des variables du système, il est difficile (sinon impossible)[19] d'identifier l'état actuel d'un système distribué et partant de son modèle.

Par une autre voie, la concurrence ajoute au modèle de l'entrelacement et de l'indéterminisme, ce qui augmente exponentiellement la taille des modèles basés sur les états.

D'un autre côté, dans les systèmes concurrents et communicants, il est plus facile de distinguer les actions que chaque processus peut exécuter à un moment donné, en particulier des actions qui peuvent représenter des communications entre processus. Les actions communicantes devront être exécutées au même temps (synchronisées) dans tout le processus qui participe à la communication.

La forme pour modéliser un processus où nous observons les actions possibles à exécuter mais pas les états, est connu comme les systèmes de transitions étiquetés (Labelled ***Transition Systems = LTSs***).

8.2. Pour quoi la méthode B ?

Dans ce mémoire, nous utilisons la méthode B, qui appartient à la première catégorie cité dans la classification des méthodes formelles, ce choix est faite par plusieurs causes le plus important sont les recherches de **Régine Laleau**[20], on va comprendre comment.

Régine Laleau analysé les différentes catégories de langages de spécifications formelles suivant.

8.2.1. Les langages de l'approche objets

On peut citer OBJECT-Z et VDM ++, sont des méthodes orientées objets. Ces langages sont jeunes et bien souvent en cours de développement.

8.2.2. Les langages de l'approche logiques d'ordre

Coq, PVS et HOL, sont des langages basés sur les logiques d'ordre supérieur, ils sont utilisés essentiellement comme outils de preuve pour les programmes et permettent de réaliser des preuves complexes. Mais ils ne nous ont pas paru bien adaptés, ils relèvent encore du domaine de la recherche et sont essentiellement utilisés pour réaliser des preuves complexes.

8.2.3. Les langages de l'approche événementielle (dynamique)

Cette approche a été utilisée à l'origine pour spécifier des systèmes communicants et en particulier les protocoles, le principe est de décrire les événements qui sont échangés entre le système et son environnement ou entre les différents processus qui composent le système. Et parmi ces langages on peut citer les réseaux de Pétri, les algèbres de processus comme CSP ou les logiques temporelles. Cette approche n'a été pas considérer dans les spécifications des systèmes répartis.[20]

8.2.4. Les langages de l'approche ensembliste

Où l'état du système est modélisé par des ensembles ou des constructeurs ensemblistes et par des contraintes exprimées en logique du premier ordre. Chaque opération est spécifiée par pré-condition ou par commandes gardées qui définissent son effet sur l'état du système, parmi ces langages on citer : Les langages Z, VDM, et B.

Les méthodes de l'approche ensembliste sont celles qui se prêtent le mieux à nos besoins. Elles sont caractérisées par la notation d'état. Un état d'un processus est défini par des ensembles et des relations entre les ensembles. Il lui est associé un prédicat qui exprime des contraintes sur ces ensembles et relations. Ces méthodes reposent sur des bases théoriques simples : les notions d'ensemble, relations et fonctions sont des concepts mathématiques connus dont l'apprentissage est relativement aisé. Ces notions ont déjà été utilisées pour formaliser les modèles sémantiques de données : des spécifications des algorithmes concernant les systèmes répartis sont étaient proposée par **J.R.Abrial** dans le livre **B-Event**.

Nous avons éliminé Z car, comme le dit **J.R .Abrial** « Z est plus une notation pour écrire des spécifications qu'une méthode de développement complète ».

En outre, la méthode B est supportée par des différents outils industriels : L'**Atelier B**, **ProB**, **Rodin**, **AnimB, … .**Ces outils ont une licence libre (open source).

9. Conclusion

Dans ce chapitre, nous avons présenté les méthodes formelles et son importance dans la modélisation, spécification et verification des programmes et protocoles des systèmes répartis. Nous devons donc disposer d'outils et de méthodes nous permettant d'atteindre ce but.

Nous avons découvert qu'il existe de nombreuses méthodes appartient aux méthodes formelles qui nous permettant de spécifie des protocoles d'un système. Et parmi ces méthodes nous avons choisi la méthode B et son extension Event-B, car les outils de cette méthode sont disponibles en licence libre « open source ».

Chapitre 2

LA MÉTHODE FORMELLE B

1. Introduction

Dans ce chapitre, on va aborder la méthode formelle B en présentant en détails le langage B et son évaluation, les machines abstraites et les raffinements.

1.1. Le langage B

Le langage B faisant référence à la théorie des ensembles et à la logique des prédicats, comprenant également une syntaxe pour décrire des "substitutions", des opérations, et les liens entre les machines, raffinements et implémentations. La description du langage et celle du raffinement et de la preuve associée au langage sont décrits dans le B-Book[21].

1.2. La méthode B

La "méthode B" consiste à définit l'ensemble comprenant : le langage B, le raffinement, la preuve, et les outils associés. Et au contraire des langages informatiques habituels, B est associé à une démarche constructive qui permet, à partir des spécifications, de les raffiner jusqu'à un niveau similaire à un code.

Cette méthode a l'avantage de couvrir toutes les phases du cycle de conception [22], depuis l'analyse des besoins jusqu'à l'implémentation finale.

Elle est de plus très bien outillée et aussi très utilisée. En outre La méthode B est utilisée dans l'industrie, ce qui rend son attractivité encore plus grande.

Donc, la méthodes B est une méthode formelle de spécification qui permet de modéliser de façon abstraite dans le langage de B le comportement d'un programme, puis par raffinements successifs, d'aboutir à un modèle concret(langage de programmation), sous-ensemble du langage transcodable en Ada ou en C[23] et concerne les étapes suivantes : spécification, conception et codage.

Toutes les raisons citées plus haut nous ont conduits à considérer B plutôt que d'autres langages du même type comme VDM ou Z (ancêtre de B), qui ne couvrent pas tous ces aspects.

Dans ce chapitre nous allons définir les méthodes B depuis sa création, passant par les différents outils et versions et ses propriétés, puis on met l'accent sur les notions de base de langage B. Et puis on explique la notion la machine abstraite par les notions attachés à elle surtout les clauses qui sont utilisée dans le langage B en détail.

2. Etat de l'art de la méthode B

2.1. Historique

La méthode B a été créé par Jean-Raymond.Abrial[1], basé sur les travaux scientifiques de E.W. Dijkstra, C.A.R. Hoare, C.B. Jones, C. Morgan, He Jifeng (Programming Research Group Université d'Oxford) et autre.

La méthode formelle B est une nouvelle approche permettant de spécifier et concevoir des logiciels et des protocoles en s'assurant de leur sûreté ainsi que de leur fiabilité. Aussi, l'ensemble des processus de spécification, de conception et de codage sont entièrement basés sur la réalisation d'un certain nombre de preuves mathématiques [24]. Ce n'est qu'après avoir prouvé mathématiquement un modèle qu'il est alors jugé cohérent et sans défaut.

2.2. Diffusion de la Méthode B dans le monde

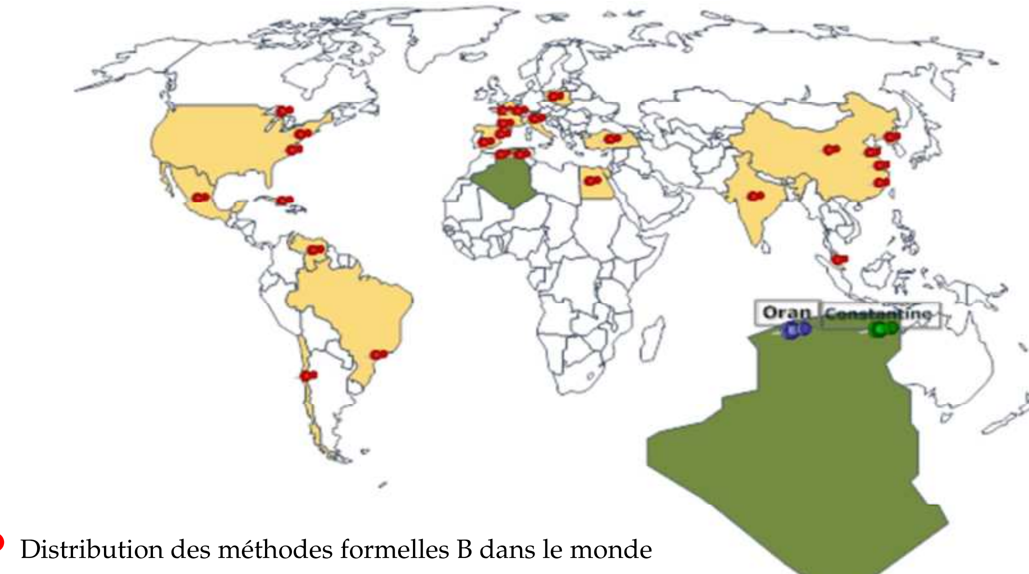
Grace à son efficacité, la méthode formelle B est largement diffusée et reconnue dans le monde surtout industriel et universitaire.

Elle est adopté dans des secteurs variés, tels que la Défense, l'aéronautique, le transport ferroviaire, ou encore la Recherche et le Développement (R&D) et des autres domaines.

La grande diffusion d'outils des méthodes B, tels que l'Atelier B, B-Event, est la cause principale de succès de la méthode formelle B.

L'**Atelier B** est un outil industriel permettant une utilisation opérationnelle de la méthode B pour des développements logiciels prouvés. ClearSy [2] en assure le développement avec des outils comme l'Atelier B, en collaboration avec des industriels, des centres de recherche et Jean Raymond ABRIAL.

[1] Un des principaux concepteurs de Z dans les années 80 et l'inventeur de la méthode B
[2] Une société de développement de logiciel dans le domaine méthode B, créer l'Atelier B



- Distribution des méthodes formelles B dans le monde
- **1. LIRE :** est un laboratoire de recherche créer en 1987 ; son objectif est de contribuer dans la recherche en informatique et plus précisément dans le domaine des logiciels et des systèmes, couvre un large spectre d'activités regroupées au sein de 4 équipes. Parmi eux, il y a l'équipe de « Génie logiciel et Système Distribués », intéresser par la vérification formelle des systèmes concurrents**[http:// www.ume.edu.dz/vf/Labo/facSclng/Labo-LIRE]**
  2. MISC : Laboratoire de Modélisation et d'implémentation des systèmes complexes, est un laboratoire offre des séminaires pour les chercheurs et les développeurs qui sont travaillé dans le domaine des systèmes complexes**[Seconde séminaire MISC . 20/21 Mai 2012. Université Mentouri de Constantine]**
- Formation professionnelle, Adem Info : Formation et Développement de logiciels 04/1998 – 07/ 1999 ; Formation, Installation Soft et Développement de Logiciels 10/1997 - 04/1998**[http: // www.vitaminedz.com/fr/oran/Formation-professionelle]**

Figure 2.1 - Diffusion de la méthode B dans le monde
et L'utilisation des méthodes formelles dans l'**Algérie**

2.3. Les utilisateurs de la Méthode Formelle B

Les utilisateurs de la Méthode B sont souvent issus de milieux divers et variés. Parmi eux, nous pouvons compter les industriels les plus souvent utilisateurs de ces méthodes. Ils cherchent des systèmes sécuritaires faisant appel aux méthodes formelles, ainsi que de nouvelles technologies pouvant répondre à leur besoin.[25]

GEC Alsthom Transport : est l'un des premiers à utiliser ces méthodes pour lesquels la sécurité est assurée par logiciel sous le couvre de projet SACEM[1].

Matra Transport International(MTI) [2] : en 1998 qui réalise le projet de métro sans conducteur parisien Météor[27]. Depuis l'année 1998 , date de mise en service Météor, les logiciels sécuritaires du métro automatique parisien , développés à l'aide des méthodes formelles , n'ont connu aucune défaillance[27].

[1] Système d'aide à la Conduite, à l'exploitation et à la Maintenance
[2] Aujourd'hui renommé SIEMENS Transportation Systèmes

Pour ces industries la méthode **B** a pris sa force et a vu fabrique l'outil qui est aujourd'hui l'**Atelier B**, ce système utilise particulièrement dans les métros sans chauffeurs, dans par exemple Hong Kong(en 2002), et à New-York ligne Canarsie en 2003), la ligne 9 du métro de Barcelone (2008), la navette Roissy VAL de l'aéroport Roissy Charles de Gaulle (2006) et la ligne 2 du métro de Budapest (2008), et d'autre.

La méthode **B** a également été appliquée dans les systèmes embarques[1] avec, par exemple, des travaux sur le diagnostic de panne dans le domaine de l'automobile ou le développement de logiciels pour des modules d'aide aux malades diabétiques sous dépendance d'insuline.

Cependant, cette dernière expérience avait une contrainte de faible espace mémoire disponible pour l'exécution de systèmes sur de telles plateformes.

C'est la même limitation atteinte par **A. Requet** et **G. Bossu**, lors de leur tentative pour embarquer, sur carte à puce, du code traduit automatiquement depuis le langage B.

Suite à ces observations, le projet RNTL BOM[2] lancé un objectif de la création d'un traducteur B vers C optimisant l'utilisation de l'espace mémoire. L'efficacité de ce projet a permis d'embarquer sur une carte à puce un système d'exploitation JavaCard[3] développé en B.

En raison de leurs besoins en sécurité, les industriels de la carte à puce sont très présents dans le domaine de la méthode **B**, ce qui a motivé un grand nombre de travaux. Certains travaux se sont orientés vers le développement d'un vérificateur de bytecode[4] embarquable sur carte à puce.

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

---

[1] Est défini comme un système électronique et informatique autonome, souvent temps réel, spécialisé dans une tâche bien précise. Le terme désigne aussi bien le matériel que le logiciel utilisés [Wikipédia]

[2] Projet RNTL :B Optimisant la mémoire.Initialement lancé par GemPlus, il a été réalisé par Gemplus, Steria(Nouvellement ClearSy),l'équipe VASCO (LIG - Grenoble) et l'équipe TFC (LIFC-Besan_con).Ce projet vise à optimiser les programmes générés avec l'AtelierB, pour être embarqués sur des cartes à puce

[3] Est un système d'exploitation lancé en 1997 ,permettant la désinstallation ou l'installation d'applications, ces application permet de gérer plusieurs applications(écrites en JavaCard) sur une même carte et désinstaller des programmes

[4] Langage de bas niveau généré par un compilateur Java et pouvant être interprété par une machine virtuelle appelée Java Runtime Environment

Et les autres utilisateurs sont :

- **Des experts et spécialistes :** Les individus cherchant des informations sur les méthodes formelles à un niveau hautement technique ;
- **Des chercheurs spécialisés en R&D :** qu'ils agissent pour un développement durable des méthodes formelles afin de développer de nouvelles solutions pour le futur.
- **Des enseignants universitaires et chercheurs :** Ils enseignent B dans le milieu académique et étudient les évolutions possibles des méthodes formelles.
- **Des élèves :** Composés de chercheurs ou tout simplement d'élèves souhaitant utiliser les outils B.

3. Versions de langage et outils de la méthode B

On distingue deux types de version très connus (Classique, Event-B) :

3.1. Classique

Cette version est définit dans le B Book de **Abrial**, cet ouvrage est considérer comme un outil industriel permettant une utilisation opérationnelle de la **méthode B** pour des développements logiciels. En utilisant le langage de support **Atelier B**, ce langage est la version gratuite de cette méthode. B-Toolkit(Bart) est un autre outil efficace permet de raffinés les spécifications en B automatiquement.

3.2. Event-B

Event-B est une amélioration utilisant uniquement la notion d'événements pour décrire les actions et non plus les opérations, supporté par Atelier B et Rodin et peut s'appliquer à des différents domaines comme l'électronique.[28]

3.3. Outils du langage B

Chaque langage B comme Atelier B et Rodin ou B-Toolkit ou autre, contient plusieurs outils utilisés permettent de la spécification des problèmes par efficacité, et chaque outil contient: Un analyseur syntaxique des machines (au niveau de la spécification, raffinement ou implémentation), un générateur des obligations de preuves, démonstrateur automatique de preuves, démonstrateur interactif de preuves …etc.

4. Notions de base de langage B

4.1. Notion de Langage B

La méthode formelle B est divisée sur deux notions, une méthode et un langage. La notion de langage dans la méthode B est aussi constituée de trois langages complémentaire: le langage de la logique des prédicats et de la théorie des ensembles ; le langage des substitutions généralisées et le langage des machines abstraites.

4.1.1. Le langage de la logique de B

Il est basé sur la logique des prédicats (premier ordre) avec égalité et paires ordonnées. Ce langage est une simplification du langage de la théorie des ensembles classique(les ensembles en particulier des entiers, les relations entre ensembles de paires, les fonctions, les séquences), en utilisant aussi les opérateurs logique afin de permettre des spécifications très concises.

4.1.2. Le langage des substitutions généralisées

Ce langage comprend des instructions de spécification (utilisent les notations de pré-condition et d'indéterminisme pour la phase de spécification et des instructions de programmation (comme la séquence et la boucle pour les étapes finales de développement uniquement).

4.1.3. Le langage des machines abstraites

Le concept de machine abstraite est similaire aux concepts de module et de paquetage, classe et autres objets des langages de programmation. Il permet de construire les logiciels et valider les protocoles de manière modulaire et structurée. Une machine abstraite encapsule des données et les opérations qui les manipulent. Une machine peut être construite à partir d'autres machines.

4.2. Notion machine abstraite

La machine abstraite dans la méthode B est composé de :

- En-tête : nom de la machine;
- Données (ensembles, variables): constituent l'état de la machine, on ne peut pas les modifie directement de l'extérieur, mais à travers les opérations de la machine;
- Opérations: constituent l'interface entre les variables et l'extérieur de la machine abstraite.

L'état de la machine est restreint par un invariant (formule logique entre les variables de la machine). Cet invariant doit être vérifié après l'initialisation des variables, avant et après chaque opération. À chaque machine sont donc associées des obligations de preuve.[8]

4.3. Notion de Raffinage

L'opération du raffinage se fait en une ou plusieurs étapes successives selon la complexité de la machine à raffiner et consiste à reformuler les variables et les opérations de la machine abstraite.

Tout au long du raffinage on passera des données abstraites aux données plus concrètes (c'est-à-dire plus proche d'un langage de programmation), et la dernière étape de raffinage est l'étape d'implantation, le tableau ci-dessus résume les différentes actions de raffinage sur les données du système (aspect statique) et sur les traitements (aspect dynamique).

	Raffinage successifs	Dernier Raffinage(Implantation)
Données	- transformation des variables en variables plus concrètes - ajout de variables (détails de conception)	- transformation des variables en structures de données programmables - valuation des ensembles et des constantes
Opérations	- réduction du non déterminisme - affaiblissement des pré-conditions - introduction du séquencement - ajout des détails de conception	- suppression du non déterminisme - suppression des pré-conditions - suppression de la simultanéité - ajout des détails de conception - introduction des itérations

Tableau 2.1 – Les différentes actions de raffinage sur les données du système et sur les traitements

Une machine peut être raffinée en plusieurs étapes avec pour but de ne pas compliquer la réalisation des preuves à travers une succession de preuves incrémentales.

Le principe général du raffinement est montré par le schéma suivant[8] :

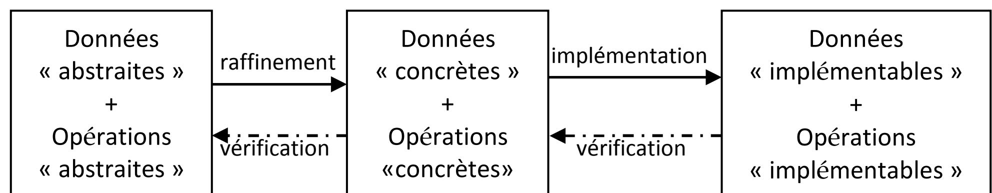


Figure 2.2 - Principe général du raffinement

Notons qu'il y a deux types de raffinement, le premier est le raffinement des opérations (dite algorithme), le deuxième est le raffinement des données, s'il introduit de nouvelles variables pour représenter, sous une forme plus proche des variables d'un langage de programmation, les variables du composant raffiné. Mais il peut aussi être une combinaison des deux types précédents.

5. Syntaxe des données d'une "machine abstraite"

La syntaxe des données d'une machine abstraite est décrite au moyen des clauses suivantes:

MACHINE	/*en tête de la machine*/
SETS	/*liste des ensembles abstraits et définition des ensembles énumérés*/
DEFINITIONS	/*liste des définitions */
VARIABLES	/* liste des variables */
INVARIANT	/* définition du type et des propriétés des variables */
CONSTANTS	/* un constante */
INITIALISATION	/* initialisation des variables */
PROPERTIES	/* introduire une formule sur un constante ou ensemble */

Tableau 2.2 – La syntaxe des données d'une abstraite

5.1. Les clauses "MACHINE" et "SETS"

La clause "**MACHINE**" introduit le nom qui identifie la machine. Et la clause "**SETS**" représente les ensembles introduits par la machine ce sont:

- Ensembles abstraits: utilisés pour désigner des objets dont on ne veut pas définir la structure au niveau de la spécification. Tous ensemble abstrait est défini non vide et fini, par exemple : **SETS** PERSONNES ;
- Ensembles énumérés: définis par les éléments de leur énumération, par exemple :
  SETS SEXE = {Féminin, Masculin}.

5.2. La clause "DEFINITIONS"

La clause "**DEFINITIONS**" constituent d'une liste d'abréviations pour un prédicat (une expression ou une substitution). Les définitions peuvent être utilisées dans la suite de la machine et sont locales à la machine où elles sont définies.

Exemple.

```
DEFINITIONS
        Composition (f, g) == f ; g ;
        AffectSeq (x, v) == x := 2 × v + 1 ;

CONCRETE_CONSTANTS
```

- *f* et *g* sont deux expressions
- Le corps de la définition *Composition* est «f ;g » où la dernière « ; » sépares cette définition de la définition suivant ;
- Le corps de la définition *AffectSeq* est «x := 2 × v + 1 » où la dernière « ; » c'est un partie de *AffectSeq*;

CONCRETE_CONSTANTS : La fin de la clause **DEFINITIONS**

5.3. La clause "VARIABLES"

Cette clause introduit la liste des variables de la machine. Elles représentent l'état de la machine. Les variables sont les seuls objets qui peuvent être modifiés directement dans

l'initialisation et les opérations de la machine. Cette clause aussi doit être accompagnée des clauses "**INVARIANT**" et "**INITIALISATION**". Elle peut être absente si la machine n'a pas de variables.

5.4. La clause "INVARIANT"

La clause "**INVARIANT**" regroupe un ensemble de prédicats. Ces prédicats qui définissent les propriétés mathématiques des variables de la machine, et permettent de typer les variables et de définir certaines contraintes que doit vérifier l'état de la machine à tout moment. Cette clause est obligatoire si la clause "VARIABLES" est présente.

5.5. La clause "INITIALISATION"

La clause "**INITIALISATION**" se compose des substitutions qui définissent les valeurs initiales de chaque variable propre à la machine. Toute variable propre à la machine doit être initialisée. Cette initialisation doit satisfaire l'invariant de la machine. Cette clause est obligatoire si la clause "**VARIABLES**" est présente.

5.6. Expressions de la théorie des ensembles de B

La syntaxe utilisée dans l'invariant et les définitions reprend la syntaxe mathématique utilisée dans la théorie des ensembles, on trouve les quantificateurs (universel et existentiel), les connecteurs (et, ou, implication, ...), les types de base (NAT, STRING), opérateurs ensemblistes (union, inclusion, intersection, appartenance, ...), les relations (inverse, composition, ...), et autres opérateurs relationnels. Nous avons récapitulé les principales expressions dans l'annexe B.

5.7. La clause "PROPERTIES"

Cette clause permet d'introduire une formule portant sur les constantes et ensembles déclarés qui est supposée être vérifiée, dans le cas ou, si les valeurs des constantes et ensembles ainsi introduits ne seront précisés qu'au moment de l'implantation (ils correspondront à des constantes du programme à exécuter), il peut être utile d'exprimer des contraintes sur ces objets.

```
MACHINE Mariages
SETS
    PERSONNES;
    STATUT= {marié, célibataire};
    SEXE= {homme, femme}
CONSTANTS
    max_pers
PROPERTIES
    max_pers ∈ N-{0} ∧ card(PERSONNES) = max_pers
VARIABLES
    personnes,
    sexe
INVARIANT
    personnes ⊆ PERSONNES ∧ sexe ∈ personnes → SEXE
INITIALISATION
    personnes, sexe := ∅, ∅
```

✓ Note : Les deux clauses "**INVARIANT**" et "**INITIALISATION**" sont obligatoires si la clause "**VARIABLES**" est présente.

6. Syntaxe des opérations d'une "machine abstraite"

Les opérations d'une machine abstraite sont définies après la clause "**OPERATIONS**", la syntaxe d'une opération se compose de deux parties suivant:

- Un En-tête ;
- Un corps.

6.1. En-tête d'une opération

L'en-tête d'une opération est constitué d'un identificateur désignant les paramètres suivant:

- Le nom de l'opération;
- Des éventuels paramètres formels d'entrée de l'opération;
- Eventuels paramètres formels de sortie l'opération.

En B, lors de l'appel d'une opération, la sémantique du passage des paramètres est la copie[8].

6.1.1. Des éventuels paramètres formels d'entrée de l'opération

Ces paramètres d'entrée sont représentés par une liste parenthésée d'identificateurs qui suit le nom de l'opération. Les paramètres d'entrée de l'opération permettent de paramétrer un appel d'opération à l'aide de valeurs. Et lors d'un appel d'opération, la valeur de chaque paramètre effectif d'entrée est recopiée dans le paramètre formel.

6.1.2. Des éventuels paramètres formels de sortie l'opération

Les paramètres de sortie sont représentés par une liste d'identificateurs précédents qui suit le nom de l'opération. Les paramètres de sortie de l'opération permettent de renvoyer les résultats d'un appel d'opération sous la forme de valeurs. Après un appel d'opération, la valeur de chaque paramètre formel de sortie est recopiée dans le paramètre effectif.

En fin, Les paramètres d'entrée et de sortie d'une opération doivent être deux à deux distincts, et la syntaxe de l'en-tête d'une opération est donnée sous la forme :

paramètres de sortie <-- nom-op(paramètres d'entrée

6.2. Corps d'une opération

Le corps d'une opération est constitué d'un ensemble d'instructions qui sont décrites par le langage des substitutions généralisées, ce langage est utilisé pour décrire le corps de l'initialisation et des opérations. Dans cette section on donne la description de quelque substitutions très utilisable. Ces substitutions données dans l'annexe B.

6.2.1. Substitution "Skip"

Cette substitution est la substitution identité, elle est utile pour exprimer que certaines branches d'une substitution IF ne font rien, elle aussi ne modifie pas les prédicats.

6.2.2. Substitution "Devient égal"

C'est la substitution de base à partir de laquelle seront définies les autres substitutions, cette substitution créé pour remplacer des variables par des expressions:

Exemple: La substitution : ***X*** := 9 remplace l'ancienne valeur de ***X*** par "9".

6.2.3. Substitution "Devient tel que"

La substitution devient tel que permet de remplacer des variables par des valeurs qui satisfont à un prédicat donné. Si plusieurs valeurs satisfont le prédicat, la substitution ne précise pas laquelle est effectivement choisie. Elle définit un comportement non-déterministe.

Exemple: La substitution : ***x : (x > x0)*** remplace l'ancienne valeur de x par des valeurs supérieure a sa valeur actuelle. La valeur avant substitution d'une variable x peut être référencée par ***x0*** dans le prédicat ***P***. Cette écriture évite d'introduire une variable intermédiaire.

6.2.4. Substitution "Devient un élément de"

La substitution devient un élément comme la substitution précédente mais avec un ensemble existe dans le système. Alors elle permet de remplacer des variables par des valeurs appartenant à un ensemble. Si l'ensemble a plusieurs valeurs, la substitution ne précise pas laquelle est effectivement choisie. Elle définit un comportement non-déterministe.

Exemple: On a l'ensemble EtatProcessus = {Active, Prêt, Désactive}, La substitution : Processus : dans EtatProcessus dit que la variable Processus prend comme valeur un élément quelconque de l'ensemble EtatProcessus. On ne sait pas à priori s'il s'agit de l'élément Active, Prêt, Désactive.

6.2.5. Substitution "Pré-condition"

La substitution pré-condition permet d'introduire le prédicat qui fixe les conditions sous lesquelles une opération est appelée.

Exemple: La substitution :

PRE $x \in$ **NAT1 THEN** $x: = x\text{-}1$ **END**

L'ensemble **NAT1** est l'ensemble des valeurs naturel **NAT1**, cette substitution dit que pré-condition $x \in$ **NAT1** est vraie, alors la substitution $x := x\text{-}1$ aura un sens.

6.2.6. Substitution "If"

La substitution if permet de définir pour une spécification plusieurs comportements possibles en fonction de la validité d'un ou de plusieurs prédicats. Le comportement défini par la substitution If est déterministe.

Exemple: Soit la substitution suivante:

```
IF (x> 10) THEN
        x := x-10 /*comportement 1 */
ELSEIF (x=0) THEN
        x := x+1 /* comportement 2 */
ELSE
        x := 1 /* comportement 3 */
END
```

Cette substitution définit trois comportements possibles selon la valeur de x.

6.2.7. Substitution "Any"

La substitution **ANY *X* WHERE *P* THEN *S* END** permet d'utiliser, dans la substitution S, les données abstraites déclarées dans la liste X et vérifiant le prédicat P.

Exemple: Soit la substitution suivant:

ANY *x* WHERE *x* ∈ NAT ∧ *x* ≤ *5* THEN y*: = x+1* END

Elle dit que pour n'importe quel x, tel que x soit un entier naturel, positif strictement et inférieur à 5, alors le comportement de la substitution est définie par "x := x+1".

6.2.8. Substitution "Appel d'opération"

Cette substitution remplaçant les paramètres formels par des paramètres effectifs. Les paramètres d'entrée éventuels sont des expressions et les paramètres de sortie éventuels sont des données accessibles en écriture.

Exemple:

opa ; // opération a
opb(*x+1, TRUE*) ; // opération b reçoit x+1
res1, res2 ← **opc ; //** résultat 1 et résultat 2affecter sur l'opération c

6.2.9. Substitution "Simultanée"

La substitution simultanée correspond à l'exécution simultanée de deux substitutions. Le caractère de simultanéité dénote le fait que les substitutions doivent pouvoir se réaliser indépendamment l'une de l'autre.

Exemple:

x := y ||
y := x

Dans cet exemple, les valeurs des variables ***x*** et ***y*** sont échangées

6.2.10. Notes

- La substitution :

  IF x>10 ***THEN x:= x***-5 ***END***

  est equivalent à:

  IF x>10 ***THEN x:= x***-5 *ELSE* ***skip END***;
- la substitution "***Any***", si plusieurs valeurs satisfont le prédicat *P*, la substitution ne précise pas laquelle est effectivement choisie .Elle définit alors un comportement non déterministe ;
- Toujours dans la substitution "***Any***", Les données abstraites de la liste ***X*** sont accessibles en lecture, mais pas en écriture dans ***S***, car ce ne sont pas des variables locales mais des données abstraites définies par le prédicat ***P*** ;
- la substitution "Appel d'opération", l'appel d'opération décrit sous quatre formes différentes, selon la présence de paramètres d'entrée et de sortie ;
- La substitution simultanée est commutative et associative ;
- La substitution simultanée de deux substitutions ***S*** et ***T*** modifie des variables différentes.

7. Implantation

L'implantation est un cas particulier de raffinement en une machine qui ne pourra plus être raffinée, La forme générale d'une machine d'implantation est :

```
IMPLEMENTATION nomImp  /* nom de la machine d'implantation */
REFINES nomMachine  /* nom de la machine raffinée */
.
.
.
END
```

En outre, l'implémentation suit les règles suivantes:

- Une implantation n'a pas de variables propres;
- Une implantation utilise les opérations de machines qu'elle importe et dont elle ne peut voir les variables sauf pour exprimer les invariants;
- Restrictions sont appliquées aux substitutions apparaissant dans une implantation pour assurer qu'elle pourra être traduite en un programme exécutable :
    - pas de choix non déterministe ou de composition parallèle,
    - les conditions dans les substitutions ***IF-THEN-ELSE*** sont limitées à des expressions booléennes simples,
    - opérations ne s'appliquent qu'à des variables locales ou paramètres,
    - les affectations mettent en jeu des expressions simples,
    - ….
- Une implantation peut introduire des substitutions correspondant à des boucles ***WHILE***, il faudra par contre indiquer pour chaque boucle ***WHILE***, un invariant et un variant (quantité qui décroit à chaque passage dans la boucle);
- Au moment de l'implantation, toutes les constantes déclarées ou les ensembles non énumérés doivent être instanciés (surtout dans une clause **VALUES** ou dans une des machines importées).

7.1. Restrictions

Les substitutions autorisées dans une implantation sont :

La substitution ***VAR IN END***: pour introduire les variables locales, ces variables doivent être typées dans la substitution en apparaissant dans une substitution d'initialisation;

- Substitution ***BEGIN END*** qui permet de grouper une séquence de substitutions ;
- Substitution ***IF THEN ELSE END*** et ses variantes : ***IF THEN END, IF THEN ELSEIF THEN ELSE END*** …etc. La condition doit correspondre à une expression booléenne calculable;
- La substitution **CASE** qui est une forme de **SELECT** mais dont les prédicats testent l'appartenance à un ensemble. La substitution s'écrit : ***CASE e OF EITHER*** E_1 ***THEN*** S_1 ***OR*** E_2 ***THEN*** S_2 ***END END*** (Dans cette substitution, ***e*** est une expression simple, E_i un

ensemble et S_i une substitution, le nombre de branches peut être quelconque et une clause ***ELSE*** peut également apparaître);

- La séquence de deux substitutions;
- Une boucle ***WHILE*** précisant un variant et un invariant;
- L'affectation d'une expression simple à une variable qui peut être une variable locale ou une variable de sortie de l'opération;
- Une opération d'une machine vue ou importée. La sortie de l'opération utilisée doit être formée de variables locales ou de variables de sortie de l'opération dans laquelle apparaît la substitution.

Il y a des substitutions comme **ANY**, **SELECT**, **LET** ..., ne sont pas autorisées dans les implantations. Comme il y a des expressions autorisées contiennent deux types. Le premier est les expressions simple, ce sont des variables locales, ou des variables d'entrées ou bien constantes ou un littéral (constante entière, valeur booléenne **TRUE** ou **FALSE**). Le deuxième type est les expressions générales qui peuvent être utilisées dans une implantation (expression simple ou expression arithmétique) construite à partir des opérateurs d'addition, de soustraction, de multiplication, de division, de reste modulo ou exponentiation.

Il est à noter que les conditions des substitutions ***IF*** ou ***WHILE***, les expressions peuvent être formées à partir des connecteurs ¬, ∧, ∨ et ⇒ et des formules de comparaison d'expressions simples.

7.2. La clause IMPORTS

La clause "**IMPORTS**", permet d'importer des machines qui, elles, seront raffinées. Cette clause se comporte dans les implémentations de manière analogue (semblable) à la clause "**INCLUDES**" utilisée dans les spécifications. Quoique, les variables de la machine importée seront complètement cachées dans les opérations de la machine qui importe.

7.3. La clause VALUES

La clause "**VALUES**" permet de donner une valeur aux constantes concrètes de la clause "**CONSTANTS**" et aux ensembles abstraits de la clause "**SETS**" de la machine raffinée. Cette clause est suivie d'un ensemble de déclarations ***id=valeur*** séparées par des points virgules. ***id*** dans ce cas est le nom de la donnée à valuer et valeur sa valeur. L'ordre des valuation est important.

Lorsqu'une constante ou un ensemble abstrait d'une implantation *M* apparaît avec le même nom dans une machine *M'* importée ou bien vue par *M*, alors la donnée de *M* est implicitement valuée par l'objet de même nom de la machine *M'*.

Exemple. Une constante de tableau peut être valuée : par un tableau dont tous les éléments ont la même valeur, exprimé sous la forme $D \times \{e\}$ avec D est le domaine du tableau et e la valeur des éléments du tableau, par exemple un tableau $tab \in (0..2) \rightarrow \mathbb{N}$ pourra être implanté par la clause : **VALUES** $tab = (0..2) \times \{0\}$.

7.4. La substitution WHILE

La substitution "***WHILE***" utilisé pour effectuer un nombre non borné d'itérations, la syntaxe de cette substitution est la suivante:

WHILE *P* **DO** *S* **INVARIANT** *I* **VARIANT** *V* **END**

D'où:

- P et I sont des prédicats (P la condition de la boucle);
- S est une substitution (le corps de la boucle);
- l'invariant et V est une expression, le variant qui représente une grandeur qui diminue à chaque passage dans la boucle mais reste positive.

Dans autre côté, L'action de cette substitution sur un prédicat R est défini comme la conjonction de plusieurs propriétés :

- L'invariant I est satisfait à l'entrée de la boucle: I ;
- L'invariant I est préservé par le passage dans la boucle: $I \land P \Longrightarrow [S]\, I$;
- Le variant est un entier: $I \Longrightarrow V \in N$;
- Le variant décroit dans le passage dans la boucle: $I \land P \Longrightarrow [n:=V]\ [S](V < n)$;
- En sortie de boucle, la propriété R découle de l'invariant et de la non-satisfaction de la condition: $I \land \neg P \downarrow R(\downarrow \textit{symbole de non satisfaction d'une prédicat})$

Exemple. On a l'exemple de machine représentant un ensemble fini (entiers non nuls), qui permet d'ajouter des éléments mais également de calculer le maximum de l'ensemble.

La spécification de la machine est la suivante :

```
MACHINE MaxEns
VARIABLE ens
INVARIANT ens ∈ F(N1)
INITIALISATION ens := ∅
OPERATIONS
        enter(n) =
              PRE  n ∈ NAT1
              THEN ens: = ens ∪ {n}
              END;
        m ← maximum =
             PRE ens ≠ ∅
             THEN m: = max(ens)
             END
END
```

Comme la seule opération à réaliser sur l'ensemble stocké dans cette machine est de trouver le maximum, il n'est pas utile de garder un ensemble arbitraire, on peut se contenter d'une variable entière. On obtient donc le raffinement :

```
REFINEMENT MaxEns1
REFINES MaxEns
VARIABLES mEns
INVARIANT mEns=max(ens ∪ {0})
INITIALISATION mEns := 0
OPERATIONS
    enter(n) =
    PRE n ∈ NAT1
    THEN mEns: = max({mEns, n})
    END;
    m ← maximum =
    PRE z ≠ 0
    THEN m: = mEns
    END
END
```

8. Le langage Event-B

8.1. Notions générales

Event-B est un langage formel qui a été développé pour spécifier correctement et modéliser itérativement des systèmes complexes par un mécanisme de raffinement. Le modèle formel d'Event-B repose sur la représentation des événements qui marquent l'évolution du système. Dans cette section, nous présentons les notations d'Event-B, et les éléments principaux qui constituent un modèle décrit avec Event-B.

8.1.1. Notation du langage Event-B

Les machines abstraites et le raffinement sont au cœur du langage Event-B. Une machine abstraite se compose d'un ensemble de constantes et de variables globales. L'état du système est défini par des variables. Chaque événement dans le modèle abstrait est composé par un ensemble de gardes et d'actions.

Les notations d'Event-B sont généralement sous-forme de notations relationnelles ou fonctionnelles. On peut résumer les notations relationnelles dans le tableau suivant:

Notation	Description
$\mapsto$	Tuple
dom(R)	Domaine de la relation **R**
ran(R)	Codomaine de la relation **R**
$\lhd$	Domaine restrictive
$\rhd$	Codomaine restrictive
R [A]	Image de la relation **R** par rapport à l'ensemble **A**

Tableau 2.3 – Notations relationnelles dans Event-B

8.1.2. substitutions généralisées

Les substitutions généralisées permettent d'exprimer l'aspect dynamique des spécifications d'évent-B. Ces substitutions généralisées sont basées sur le calcul de la plus faible pré-condition de Dijkstra.

La sémantique des substitutions est définie sous la forme de transformateurs de prédicats. Soit la formule: ***P[S] R*** où, ***P*** exprime la post-condition de la substitution, et ***R*** la post-condition de l'instruction ***S***. Cette formule signifie que si le prédicat ***P*** est vrai et que l'instruction ***S*** se déclenche, alors l'assertion ***R*** est vraie après l'exécution de ***S***.

8.2. Modèle du système avec le langage Event-B

Dans le langage Event-B, l'état d'un système est représenté par un ensemble des variables typés. Les événements marquent les changements d'état du système. Les événements se composent des gardes sur des actions qui peuvent se produire spontanément au lieu d'être invoqués. Les modèles d'Event-B sont décrits par deux concepts de base: les contextes et les machines.

8.2.1. Contexte

Un contexte représente la partie statique du modèle, il est composé de constantes et d'axiomes décrivant les propriétés de ces constantes. Un contexte peut être étendu par un autre contexte, et il peut être visible pour une machine en utilisant la clause "**SEES**".

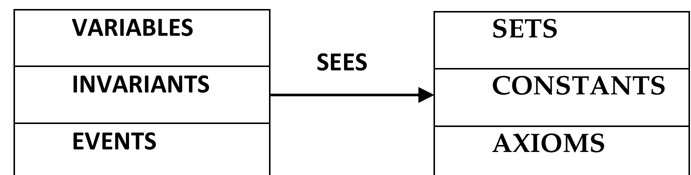


Figure 2.3 **-** Relation entre un contexte et une machine

On a le contexte suivant :

```
CONTEXT ctx_0
SETS D
CONSTANTS n , f, v
AXIOMS
        axm1 : n ∈ N
        axm2 : f ∈ 1..N → D
        axm3 : v ∈ ran(f)
THEOREMS thm1: n ∈ N1
END
```

On décrit dans le contexte "ctx_0 " les éléments suivants:

- L'ensemble ***D*** ;
- Trois constantes sont définies ***n***, ***f*** et ***v*** ;

- La clause axiomes on définit trois axiomes qui décrivent:
  - ***n*** est un entier positif (***axm*** 1),
  - est une fonction totale de l'intervalle ***1... n*** à l'ensemble ***D*** (***axm2***),
  - ***v*** est supposé appartenir à l'image de la fonction ***f*** (***axm3***) ;
- Un théorème est proposé à la fin du contexte: ***n*** est un entier positif (***thm 1***).

8.2.2. La machine Event-B

La machine contient des éléments dynamiques qui décrivent l'état du système. Une machine est constituée de trois sections principales : les invariants, les variables et les événements. Les événements permettent de changer l'état des variables tout en respectant les conditions décrites dans la clause des invariants.

```
MACHINE m_0a
SEES ctx_0
VARIABLES x
INVARIANTS Inv1: n ∈ N
EVENTS
    ...
END
```

8.2.3. L'événement

Les événements jouent le rôle des opérations de la machine. Chaque événement est composé d'une condition et une action. Une fois les conditions satisfaites, l'événement peut être déclenché à tout moment. Cependant, dès que la condition ne tient plus, l'événement ne peut plus être déclenché.

L'action, comme son nom l'indique, détermine la façon d'évoluer les variables au moment où l'événement se produit. L'action permet de spécifier et de déduire les propriétés du système. L'événement d'une machine abstraite est correcte lorsque celle-ci préserve l'invariant de la machine. La figure ci-dessus présente un exemple d'événement décrit avec l'outil **Rodin**.

```
RECHERCHE =
STATUS ordinary
ANY k
WHERE
    grd1: k ∈  1 .. N
    grd1: f(k) =v
THEN act1: i := k
END;
```

8.3. Les outils ProB et Rodin

8.3.1. L'outil de vérification ProB

ProB s'avère un outil qui automatise la vérification de la cohérence des machines via un modèle de vérification. Nous pouvons utiliser l'interpréteur de **ProB** pour exécuter directement les scénarios de tests possibles pour vérifier la couverture des événements de la machine, et pour détecter les interblocages (deadloacks).

L'outil **ProB** permet de détecter les expressions indéfinies comme l'application partielle d'une fonction à des arguments en-dehors de son domaine.

8.3.2. Rodin

Rodin est une Plate-forme open source basée sur Éclipse qui offre un soutien efficace pour le raffinement et la preuve mathématique. La plate-forme Rodin est l'acronyme de (***Rigorous Open Development Environment for Complex Systems***). Il s'agit d'un outil de développement, permettant d'intégrer les outils de support à la méthode B.

La plate-forme **Rodin** a certaines caractéristiques qui en font d'elle un outil unique (efficace, fiable et réutilisable) pour le développement de modèles. Ainsi, il aide à la compréhension du système dans son ensemble.

8.4. Les avantages de Event-B

Un des avantages de la méthode B-événementiel est qu'elle permet plus de flexibilité que les langages d'entrées des model-checkers tout en nécessitant moins d'interaction avec l'utilisateur que les prouveurs de théorèmes tel que **PVS**[1]. Aussi, le raffinement dans cette méthode, permet d'enrichir le modèle graduellement, et de renforcer peu à peu l'invariant, alors qu'en général en model-checking on passe directement vers un modèle très complexe du système à vérifier. Le raffinement aussi permet d'assurer un meilleur dialogue entre la personne créant le modèle et son client, ainsi de réparer les erreurs en cours de développement. Aussi on valide les modèles étape par étape, plutôt que sur un unique modèle final construit directement.

8.5. Les limites de Event-B

La méthode B-événementiel permet d'établir des preuves de propriétés d'invariance, par conséquent, on peut vérifier la correction d'une spécification et sa complétude en faisant appel à la théorie des ensembles.

Cependant, la méthode B-événementiel ne permet pas d'exprimer explicitement des propriétés temporelles (fatalité, équité...).

8.6. Différence entre Event-B et B classique

Les principales différences entre Event-B et B classique sont les suivantes[29]:

- Notion d'opération est remplacée par la notion d'événement. Un événement a une garde (sans pré-condition), et peut être déclenché si la garde est vraie;

[1] PVS ou Prototype vérification System : est un système pour spécifier et vérifier des propriétés sur des systèmes logiciels. Il repose sur la logique d'ordre supérieure classique

- Le remplacement des notions de structuration de machines (**USES, SEES, INCLUDES, IMPORTS** ...) par la seule notion de contexte, qui regroupe les constantes et les ensembles de bases, ainsi que des axiomes associés;
- La disparition de certains types de données et leurs opérateurs associés, notamment les séquences (**seq**), les structures (**struct**) et les arbres (**btree**);
- La disparition de certaines substitutions. Chaque événement en Event-B a en fait une forme très simple. La forme la plus compliquée peut-être exprimée par un seul **ANY** contenant des assignations (déterministes et non déterministes) parallèles. Les variables de **ANY** sont les «paramètres» de l'événement. Il existe deux formes simplifiées, une pour des événements sans paramètre et une pour des événements sans paramètre et sans garde;
- Une notion adaptée du raffinement, permettant l'introduction de nouveaux événements (qui doivent raffiner ***skip***), de séparer un événement en plusieurs événements dans la machine raffinée, et inversement, de fusionner plusieurs événements;
- l'apparition de certains nouveaux opérateurs, comme par exemple, la relation totale (**< <->**);
- Les substitutions autorisées en B évènementiel sont moins nombreuses. Cette réduction est liée au fait que le B événementiel oblige à raisonner en termes d'activation d'événement plutôt qu'en termes d'appel de sous-routine.

9. Conclusion

Plusieurs méthodes ont été proposées pour valider les protocoles sur les différents côtés de specification et modélisation d'un protocole ou programme, et la méthode formelle B est la plus adéquate solution pour atteindre ce but, car elle est basée sur la verification en utilisant les théorèmes et les preuves mathématiques qui sont mettre notre specification plus forte qu'avec autre type de verification.

Donc, nous avons présenté dans ce chapitre la méthode formelle B. Et en particulière l'Event-B le langage formelle basé inventer pour la modélisation et le raisonnement mathématique pour les systèmes répartis. Ce langage est basé sur la théorie des ensembles et la logique des prédicats du premier ordre.

Chapitre 3

LOGICIELS ET OUTILS B UTILISÉS

1. Introduction

Dans ce chapitre, nous utiliserons l'**Atelier B** et **Rodin** et **ProB** comme des outils de spécification et animation.

1.1. L'Atelier B

L'**Atelier B** est un logiciel développé par **Clearsy** en collaboration avec Jean-Raymond ABRIAL et **Alstom**, et avec le financement et la participation de la **RATP**, la **SNCF** et l'**INRETS**.

Ce logiciel permet une utilisation opérationnelle de la méthode B. Il offre de nombreuses fonctionnalités permettant de gérer des projets en langage B. Ces fonctionnalités se regroupent en quatre grandes catégories : les tâches automatisables lors du développement d'un projet, une aide à la preuve, une aide au développement, et des outils pour l'utilisateur : comme par exemple représentation graphique de projets, navigateur hypertexte pour se diriger dans un projet, affichage de l'état et des statistiques d'un projet.[30]

1.2. Le ProB

ProB est un animateur et modèle de vérification pour les méthodes B, il permet de générer automatiquement des animations totales de plusieurs spécifications B. Ainsi, il permet de vérifier les erreurs de spécification. **ProB** est maintenir par **Michael Leuschel** basé sur les recherches et l'effort d'implémentation de **Michael Leuschel**, **Michael Butler**, **Carla Ferreira**, **Leonid Mikhailov**, **Edward Turner**, **Phil Turner** et **Laksono Adhianto**.[31]

ProB permettent de simuler l'exécution d'une spécification sous la forme d'une animation de dessins. Chaque état de chaque composant physique modélisé est associé à un état de la description formelle et à une disposition graphique des dessins. Chaque événement du modèle est associé à une ou des animations graphiques. L'animation du modèle formel permet alors de générer une séquence de dessins. L'utilisation d'un animateur permet d'évaluer quelles gardes sont vérifiées dans un état donné et donc de choisir une action à activer. Cette approche permet une bonne compréhension des évolutions possibles d'un système, mais nécessite un temps de développement assez long, puisqu'il faut dessiner tous les composants et prévoir les différentes transitions possibles du système.

2. Atelier B

2.1. Lancement de l'Atelier B et création d'un nouveau projet

Lors de la création d'un nouveau projet dans l'**Atelier B**, nous pouvons remarquer la création de deux nouveaux répertoires : "**bdp**" et "**lang**". Ces deux répertoires vont être utilisés par l'**Atelier B** de la manière suivante :

2.1.1. "bdp"

Ce répertoire contient les différents fichiers outils nécessaires à l'**Atelier B** pour gérer et appliquer la méthode B à notre «projet». Il contient également les fichiers de documentations qui pourront être générés par l'**Atelier B** à la demande.

2.1.2. "lang"

Ce répertoire contient quant à lui l'ensemble des fichiers sources générés par l'**Atelier B** dans le langage spécifié pour le projet.

2.2. Création d'une machine abstraite

2.2.1. Machine "*Rech_Element*"

On considère une fonction de recherche d'un élément "***n***" dans un tableau "***tableau***" d'entiers de taille "***x***". L'objectif de cette question est de créer une machine abstraite qui représente cette fonction. Nous avons réalisé la machine suivante.

```
MACHINE
    Rech_Element
CONSTANTS
    xx,
    tableau
PROPERTIES
    xx: NAT &
    tableau: 0..xx-1 --> NAT
OPERATIONS
    yy <-- RechercheOp(nn) =
    PRE nn:NAT
    THEN
        IF nn : ran(tableau)
        THEN
            yy := TRUE
        ELSE
            yy := FALSE
        END
    END
END
```

Figure 3.1 **-** La machine abstraite *Rech_Element*

Avec :

- ***xx, tableau*** : sont des constates
- ***RechercheOp(nn)***: l'opération qui réalise la recherche de l'élément "***nn***" dans le tableau.

2.2.2. obligation de preuve

L'état d'une machine abstraite doit vérifier en tout temps l'invariant. Une "obligation de preuve" est ainsi associée à chaque prédicat défini dans l'invariant. Cette obligation de preuve doit s'assurer du respect de ce prédicat lors de toute modification effectuée sur les variables considérées dans le prédicat.

Dans l'exemple suivant:

```
MACHINE
    Rech_Maximum
VARIABLES
    xx, yy, zz
INVARIANT
    xx : INT &
    yy : INT &
    zz : INT &
    (zz = xx or zz = yy) &
    zz >= xx & zz >= yy
INITIALISATION
    xx, yy, zz := 0, 0, 0
OPERATIONS
    Recherche (nn) =
    PRE
        nn : INT & nn >= 0
    THEN
        xx := nn ||
        yy := xx ||
        zz := max ({xx, nn})
    END
END
```

Figure 3.2 **-** La machine abstraite *Rech_Maximum*

Il faudra s'assurer que le prédicat ***P*** ("***nn*** : **INT &** ***nn >= 0***") soit vrai comme "pré-condition" à la réalisation de la substitution ***S*** dans l'opération "***Recherche(nn)***":

xx := nn ||

yy := xx ||

zz := **max** ***({xx, nn})***

Si ***P*** est vérifié, l'obligation de preuve veillera à ce que la substitution ***S*** préserve l'invariant ***I*** (***xx*** : **INT** & ...) à l'issue de sa réalisation. Si ce n'est pas le cas, ***S*** ne sera réalisé.

2.2.3. Génération des obligations de preuves *POs*

En ce qui concerne les obligations de preuves de notre machine, nous n'avons remarqué aucun problème.

- ✓ **Note :** En réalisant la machine selon la description de l'énoncé, nous n'avons aucune clause "**INVARIANT** ", de ce fait aucune "obligation de preuve" n'a pu être générée.

2.2.4. Statut du projet

Figure 3.3 **-** Le statut de la machine abstraite ***Rech_Maximum***

Signification des acronymes du statut :

- **TC**: Type Checked : Vérification du typage des variables utilisées.
- **POG** : PO Genereated : État de la génération des preuves (terminée et réussie ou non).
- **nPO** : Proof Obligations : Nombre d'obligations de preuve générées suite à l'invariant.
- **nUn** : Unproved : Nombre de PO non prouvées.
- **Pr** : Proved : Nombre d'obligations de preuves vérifiées.
- **BOC** : BO checked : État au niveau B0.

2.3. Machine abstraite et implémentation

2.3.1. Machine abstraite "*Operations*"

Nous avons utilisé deux exemples de typage des variables. Dans la première opération, nous avons défini les variables "***aa***" et "***bb***" et "***cc***" séparément. Dans la seconde, et puisque "***aa***", "**bb**" et "cc" et "dd" appartiennent au même ensemble **NAT**, nous avons pu utiliser la notation par produit cartésien.

```
MACHINE
    Operations
OPERATIONS
    xx <-- maxOp(aa,bb,cc) =
    PRE
        aa : NAT &
        bb : NAT &
        cc : NAT
    THEN
        xx := max({aa,bb,cc})
    END;
    yy <-- minOp(aa,bb,cc,dd) =
    PRE
        aa,bb,cc,dd : NAT*NAT*NAT*NAT
    THEN
        yy := min({aa,bb,cc,dd})
    END
END
```

Figure 3.4 **-** La machine abstraite ***Operations***

- ✓ **Note 1:** Les fonctions intégrées « **min** » et « **max** » travaillent sur des ensembles d'entiers, d'où la nécessité d'utiliser les accolades « ***{}*** ».
- ✓ **Note 2:** Nous n'avons pas utilisé d'invariant explicite et donc il n'y a aucune obligation de preuve à cet instant.

2.3.2. Implémentation "*Operations_i.imp*"

Pour générer ces implémentations, nous nous sommes laissés influencée très fortement par la syntaxe et la logique de programmation du langage ***C***. De ce fait, nous pouvons remarquer l'utilisation de doubles conditions dans les clauses ***IF***.

```
IMPLEMENTATION
   Operations_i
REFINES
   Operations
OPERATIONS
    xx <-- maxOp(aa,bb,cc) =
    IF (aa >= bb) & (aa >= cc)
    THEN xx:= aa
    ELSIF (bb >= aa) & (bb >= cc)
    THEN xx:= bb
    ELSE xx:=cc
    END;
    yy<-- minOp(aa,bb,cc,dd) =
    IF (aa <= bb) & (aa <= cc) & (aa <= dd)
    THEN yy:= aa
    ELSIF (bb <=aa) & (bb <= cc) & (bb <= dd)
    THEN yy:= bb
    ELSIF
        (cc <=aa) & (cc <= bb) & (cc <= dd)
    THEN
        yy:= cc
    ELSE yy:= dd
    END
END
```

Figure 3.5 **-** La machine abstraite *Operations_i*

- ✓ **Note 1 :** A ce niveau, l'**Atelier B** nous a généré 14 preuves pour vérifier que notre implémentation effectue bien les spécifications énoncées dans la machine abstraite.

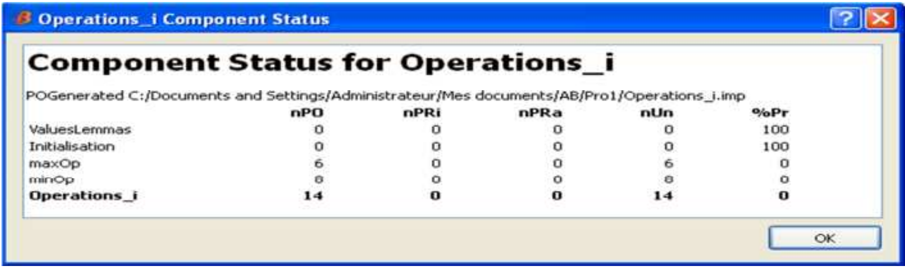


Figure 3.6 **-** Le statut de la machine abstraite *Operations_i*

- ✓ **Note 2 :** Dans les 14 preuves, on a 6 preuves qui ne sont pas prouvées concernant l'opération ***maxOp*** et le reste concernant l'opération ***minOp***, Avec :
  - **nPRi**: Cette colonne contient le nombre d'obligations à prouver par le prouveur interactive **Interactive Proof**.
  - **nPRa**: Cette colonne contient le nombre d'obligations insensibles à prouver par le prouveur automatique **Automatic Proof**.
- ✓ **Note 3 :** La dernière entrée de ligne (on gras), collecte toutes les informations de chaque opération de la composante.

2.3.3. Utilisation du "*prouveur*"

Nous avons essayé de prouver ces obligations de preuve à l'aide du "*prouveur*" automatique de force successives **0**, **1**, **2** et **3**. Nous avons ainsi pu remarquer que l'utilisation de "force" de niveau supérieur permet, de temps à autre, de prouver certaines preuves qui ne l'étaient pas au niveau inférieur.

Cependant, nous pouvons ajouter que nous n'avons pas le même comportement selon les conditions que nous indiquons dans notre implémentation.

Par exemple, lors de l'utilisation de symboles de comparaison « **strict** », le "*prouveur*" cherche à montrer que (***b+1 <= a***). Or « ***b*** » appartenant à l'ensemble **NAT** et ce dernier étant borné, il se peut que ***b+1*** dépasse cette limite si "*b*" égale ***MAXINT***. De ce fait, il ne pourra prouver que « ***a*** » appartient bien à ***NAT*** avec la condition (***b+1 <= a***).

Nous avons également pu remarquer que le prouveur rencontre quelques difficultés lorsque nous indiquons plusieurs conditions dans les clauses ***IF***.

Et enfin, dans le cas où tous les nombres, "***aa***", "***bb***", "***cc***" et "dd" sont égaux, nous devons en choisir un arbitrairement pour le désigner comme valeur maximale de l'ensemble. Ceci est également un élément que le prouveur a du mal à vérifier.

Ainsi, nous pouvons dire que pour qu'un maximum d'obligation de preuves soit vérifié, il faut les détaillées un maximum dans l'implémentation pour faciliter le travail "automatique" du prouveur.

2.4. Le prouveur interactif

2.4.1. Prouveur automatique

On a la machine suivant:

```
MACHINE
    Exemple1

VARIABLES
    ens1,
    ens2
INVARIANT
    ens1 <: NATURAL &
    ens2 <: NATURAL &
    ens1 <: ens2
INITIALISATION
    ens1,ens2 :∣= {1,2,3},{2,3,4}
END
```

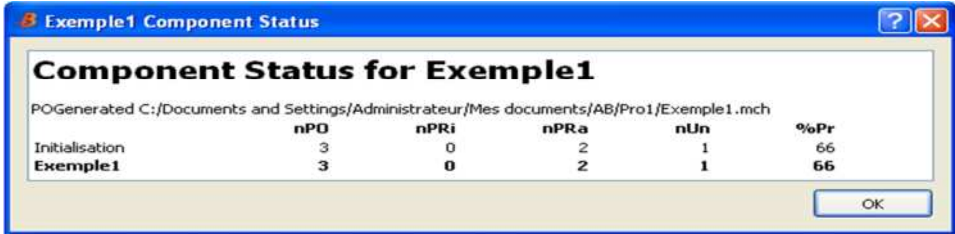
Figure 3.7 **-** La machine abstraite ***Exemple1***

Si nous regardons cet exemple, nous pouvons lire :

- Variables : La déclaration de deux variables "***ens1***" et "***ens2***".

- Invariants : Les prédicats d'invariant tel que "***ens1***" et "***ens2***" sont des sous ensemble de NATURAL. Et "***ens1***" est même un sous ensemble de "***ens2***".
- Initialisation : L'initialisation spécifie les valeurs comprises dans les intervalles "***ens1***" et "***ens2***".

Le prouveur automatique de force **0**, **1**, **2** et **3** cherche donc à vérifier l'invariant restrictif : ***ens1*** ⊆ ***ens2***. Pour ce faire, il cherche à vérifier que chaque élément de "***ens1***" est bien inclus dans "***ens2***".


Exemple1 Component Status

Component Status for Exemple1

POGenerated C:/Documents and Settings/Administrateur/Mes documents/AB/Pro1/Exemple1.mch

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	2	1	66
Exemple1	**3**	**0**	**2**	**1**	**66**

OK

Figure 3.8 **-** Le statut de la machine abstraite *Exemple1*

2.4.2. Prouveur interactif

En utilisant le prouveur interactif[1], nous remarquons que le prouveur n'arrive pas à vérifier l'invariant tel que :

```
dd dd0 dd1 mp pr PP0 PP1 t0 t1 ss ch ct rp1 te te+ Force 0
---------------------
        "`Check that the invariant (ens1 <: ens2) is established by the initialisation - ref
3.3'" => 1: {2,3,4}
```

Figure 3.9 **-** Un extraire de Exemple1.po de la machine abstraite *Exemple1*

En saisissant la commande "***pr***" nous obtenons le message suivant :

```
---------------------
        "`Check that the invariant (ens1 <: ens2) is established by the initialisation - ref
3.3'" => 1: {2,3,4}
---------------------
        bfalse
```

Figure 3.10 **-** Utilisation La commande "*pr*" sur la machine abstraite *Exemple1*

En effet, le résultat de l'initialisation est faux et ne vérifie pas l'invariant. Si nous ne changeons pas les valeurs fournies, nous ne pourrons rien faire d'autre.

[1] Ce prouveur utilise une liste continent beaucoup théorèmes qui sont utilises selon le cas d'erreur

2.4.3. Preuve d'un lemme mathématique

Prouvez à l'aide du prouveur interactive le lemme suivant :

Si a* ∈ *1, 2, ..., 10* et *b* ∈ *2, ..., 10* et *c* ∈ *3, ..., 10

Alors* max*(a, b, c)* ∈ *1, 2, ..., 10

Pour traduire ce lemme, nous avons déclaré les variables ***aa***, ***bb***, ***cc*** et ***dd*** qui représente la valeur du **max**.

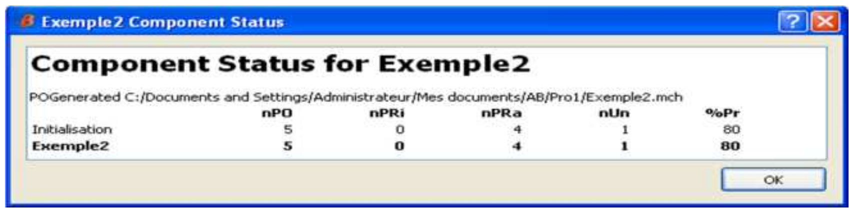
Dans l'invariant, nous avons directement traduit les conditions d'appartenance des variables aux différents ensembles.

Nous avons ensuite défini la condition sur "***dd***" qui devra appartenir au même ensemble que "***aa***" et qui représentera le max de ces trois variables.

```
MACHINE
      Exemple2
VARIABLES
      aa,  bb,  cc,dd
INVARIANT
      aa  :  1..20  &
      bb  :  2..20  &
      cc  :  3..20  &
      dd:1..20 &
      dd=max({aa,bb,cc})
INITIALISATION
      aa:=1||
      bb:=2||
      cc:=3||
      dd:=2
END
```

Figure 3.11 **-** La machine abstraite ***Exemple2***

Pour tester notre lemme, nous avons initialisé les variables et lancer le prouveur interactif. De manière automatique au niveau **0**, il réussie à nous prouver **4** obligations sur **5** : les initialisations de "**aa**", "**bb**", et "**cc**".



Exemple2 Component Status

Component Status for Exemple2

POGenerated C:/Documents and Settings/Administrateur/Mes documents/AB/Pro1/Exemple2.mch

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	5	0	4	1	80
Exemple2	5	0	4	1	80

OK

Figure 3.12 **-** Le statut de la machine abstraite ***Exemple2***

Nous utilisons ensuite la commande "***pr***" sur les deux obligations successives et obtenons :

```
--------------------
        "`Check that the invariant (dd = max({aa,bb,cc})) is established by the initialisation -
ref 3.3'" => 3 = max({1,2,3})
--------------------
        "`Check that the invariant (dd = max({aa,bb,cc})) is established by the initialisation -
ref 3.3'" => 3 = max({1,2,3})
```

Figure 3.13 **-** Utilisation La commande "*pr*" sur la machine abstraite ***Exemple2***

En effet, "***dd***" appartient bien à l'intervalle spécifié mais ne satisfait pas la condition d'être le max de "***aa***", "***bb***" et "***cc***". En modifiant la valeur de "***dd***" à **3** par exemple, il n'y a plus de problème.

2.5. Machine abstraite de réservation

2.5.1. Réalisation de la machine

Nous avons compléter l'exemple fourni par les instructions en gras.

```
MACHINE
    Reservation
VARIABLES
    nbrePlaceLibre
INVARIANT
    nbrePlaceLibre : NATURAL
INITIALISATION
    nbrePlaceLibre := 10
OPERATIONS
    Reserver =
    PRE
        nbrePlaceLibre > 1
    THEN
        nbrePlaceLibre := nbrePlaceLibre - 1
    END ;
    Annuler =
    nbrePlaceLibre := nbrePlaceLibre + 1
END
```

Figure 3.14 **-** La machine abstraite ***Reservation***

Dans l'opération "***Reserver***", aucune valeur n'est passée en paramètre, il faut donc juste s'assurer qu'il reste encore une place disponible avant d'effectuer la soustraction, bien que cela soit vérifié par l'invariant (en effet, si ***nbrePlaceLibre = 0***, la réservation réalisera la substitution et ***nbrePlaceLibre*** prendra la valeur de ***-1***(***nbrePlaceLibre*** est un entier naturel qui ne peux être négatif).

Dans l'opération "***annuler***", nous n'avons aucune pré-condition à vérifier puisque NATURAL n'est pas borné.

2.5.2. Générer les *POs* et les prouver

A. Obligations de preuves de "***Reserver***" :

```
--------------------
        "`Check that the invariant (nbrePlaceLibre: NATURAL) is preserved by the operation - ref
3.4'" => nbrePlaceLibre-1: INTEGER
--------------------
        "`Check that the invariant (nbrePlaceLibre: NATURAL) is preserved by the operation - ref
3.4'" => 0<=nbrePlaceLibre-1
```

Figure 3.15 **-** Obligations de preuves de *Reserver*

B. Obligations de preuves de "***Annuler***" :

```
--------------------
        "`Check that the invariant (nbrePlaceLibre: NATURAL) is preserved by the operation - ref
3.4'" => nbrePlaceLibre+1: INTEGER
--------------------
        "`Check that the invariant (nbrePlaceLibre: NATURAL) is preserved by the operation - ref
3.4'" => 0<=nbrePlaceLibre+1
```

Figure 3.16 **-** Obligations de preuves d'*Annuler*

2.5.3. Résultat

Il n'y eu aucun problème lors de la vérification des obligations de preuves. Lors du raffinement de notre machine et avec l'utilisation d'ensembles concrets comme **NAT**, nous devrons nous assurer lors de l'opération "**annuler**" que nous ne dépasserons pas le nombre de place originale qui pourra être défini dans une constante.

3. ProB

Nous avons réalisé dans la section suivante, les machines définirent précédemment sur l'outil **ProB**. Et montrer quelques options notamment les graphes par (sur "***dot***","***dotty***" ...etc.), et surtout l'option intégré récemment dans l'**Atelier B**, cette option a pour but de travailler directement de l'**Atelier B** avec **ProB** :

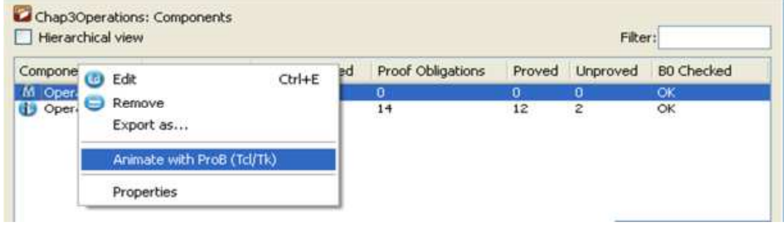


Figure 3.17 **-** Option d'animer un spécification avec ProB(**Tcl/Tk**)

3.1. L'animateur **ProB** et contrôleur du modèle pour la méthode B

3.1.1. Caractéristiques de ProB

On utilise la version la plus récente, est la version **ProB 1.3.1-final5**, le dernier changement a été le 11-01-2011.

Cette version a l'avantage de s'installer dans l'**Atelier B**. **ProB** utilisée avec deux modes d'utilisation (normale, débutant).

Il avait plusieurs options, comme les vérifications des formules de langage **LTL**, **CSP** et **XTL**. Et cette version aussi de **ProB** offre la possibilité d'utiliser d'autres outils extérieurs (**Spin/Promela**, **FDR/CSP**).

3.1.2. Utilisant ProB avec machine B simple

Avant de commencer tout travail avec **ProB** il est important que vous vous assurez que **ProB** opère correctement, Pour vérifier ceci, vous devriez exécuter self check: **Debug → Perform Self Check**. S'il n'y a pas de problèmes, vous devriez voir la fenêtre qui confirme ce **ProB** est passer l'autotest (**self-check**) suivant:

Figure 3.18 - Self-checker

Commencez en chargeant (**File → Open**) la machine ***Rech_Maximum***. Vous devriez maintenant être présentés avec la vue suivante de **ProB**:

```
/* Rech_Maximum
 * Author: Administrateur
 * Creatio
 */
MACHINE
    Rech_Maximum
VARIABLES
    xx, yy, zz
INVARIANT
    xx : INT &
    yy : INT &
    zz : INT &
    (zz = xx or zz = yy) &
    zz >= xx & zz >= yy
INITIALISATION
    xx, yy, zz := 0, 0, 0
OPERATIONS
    Recherche (nn) =
    PRE
        nn : INT & nn >= 0
    THEN
        xx := nn ||
        yy := xx ||
        zz := max ({xx, nn})
    END
END
```

StateProperties | EnabledOperations | History
initialise_machine(0,0,0)

Figure 3.19 **-** La machine abstraite ***Rech_Maximum***

L'état courant de la machine est montré dans la vitre des propriétés de l'état (gauche inférieure). Notez que la vitre des Propriétés de l'état est vide. C'est parce que la machine n'a aucun état. Confirmer ceci, produisez un diagramme de l'état courant (**Animate** → **View Current State**).

Vous devriez voir un diagramme comme celui montré ici:

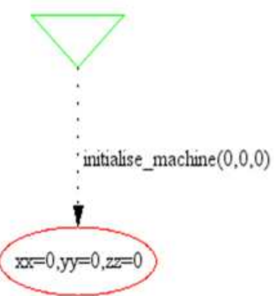


Figure 3.20 **-** Diagramme de la machine abstraite ***Rech_Maximum***

A. Current State: L'état courant de la machine est une racine (root) (or quel que soit la forme au sommet).
B. Enabled Operations : De l'état courant, il y a une seule opération (dans notre exemple). Vous avez pu remarquer que cela paraît aussi dans la vitre des: **Enabled Operations**. Dans l'état courant (c.-à-d. vide) l'opération qui est montrée est l'opération ***initialise_machine***.
C. Open State : Au fond du diagramme un autre nœud (état) est montré. **ProB** a automatiquement montré que le les états suivants qui sont accessible de l'état courant en appliquant (**one of**). Les Nœuds qui ne sont pas visités sont ouverts. Par défaut, les nœuds ouverts sont colorés en rouge.

Ensuite, un double clic sur **Enabled Operations** (milieu inférieur). Nous devrions voir :

Figure 3.21 - Après double clic sur ***initialise_machine***

D. State Properties : Montre les valeurs actuelles du variables ***xx, yy*** et ***zz***. Et aussi s'il y a un invariant qui a été violé ou pas. **Enabled Operations** montre que scinque opérations sont maintenant possibles, et la vitre **Historique** montre la dernière opération qui a été appliquée (***initialise_machine(0, 0, 0)***).

En plus des opérations qui sont maintenant possible de la spécification de la machine, **ProB** fournit aussi la capacité d'annuler le dernier que cela a été appliqué. Cette option est inscrite comme **BACKTRACK**. Si vous sélectionnez (double clic) sur **BACKTRACK**, vous reviendrez à l'état que la machine était chargée en premier.

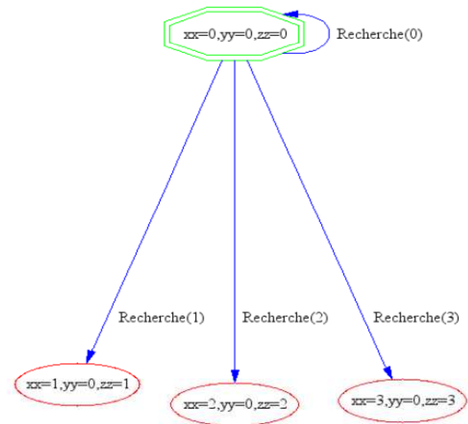


Figure 3.22 - View Current State de la machine ***Rech_Maximum***

Maintenant régénérez la visualisation de d'état courant (**Animate → View Current State**). Vous noterez que le nœud courant est maintenant un double octogone ligné dans le diagramme.

Le nœud courant est assigné sa propre forme. Il y a maintenant trois nœuds ouverts. Sur chaque arc l'opération qui prend la machine de l'état courant (***xx=0***, ***yy=0***, ***zz=0***) à l'état montré est donnée. Par exemple, une opération de ***Recherche(1)*** apportera l'état courant à un état où (***xx=1***, ***yy=0***, ***zz=1***), et ***Recherche(2)*** apportera l'état courant à un état où (***xx=2***,***yy=0***,***zz=2***)...etc. Les opérations qui sont possibles dans un état particulier sont toujours montrées dans **Enabled Operations**. En effet, **Enabled Operations** inclut maintenant ***Recherche(1)*** et ***Recherche(2)*** et ***Recherche(3)***.

Maintenant sélectionnez encore l'opération ***Recherche(1)***, la vitre **State Properties** des propriétés de l'état montrera maintenant que (***xx=1***, ***yy=1***, ***zz=1***). La vitre de **History** montrera maintenant aussi l'opération de ***Recherche(1)*** aussi bien que l'initialisation. La vitre de **History** montre des opérations dans l'ordre inverse parce que l'historique est un tas.

Pour comprendre le but des trois types de visualisation que **ProB** offre, on examine les trois diagrammes ci-dessous.

- **Animate → View Visited States** : Montre tous les états qui ont été visités avec tous les états ouverts rencontrés.
- **Animate → View Shortest To Current State** : Montre l'état courant et tous les états adjacents. Une existence de l'état adjacent est accessible par une opération sur la machine.
- **Animate → View Visited States** : Montre le chemin le plus court de l'état initial (c.-à-d. racine) à l'état courant. Les cycles Survenus sont omis.

Dans tous les cas, les nœuds ouverts (unvisited) sont montrés (par défaut en rouge) aussi bien que nœuds fermés.

Maintenant nous reviendrons à l'initialisation de machines, et on prend l'exemple de machine ***Etage.mch*** suivante:

```
MACHINE
    Etage
ABSTRACT_VARIABLES
    Indice
INVARIANT
    Indice : 0..5
OPERATIONS
    inc =
    PRE Indice < 5
    THEN
        Indice := Indice + 1
    END;
    dec =
    PRE Indice > 5
    THEN
        Indice := Indice - 1
    END

END
```

Figure 3.23 **-** La machine abstraite ***Etage.mch***

Nous recevrons un message d'erreur suivant :

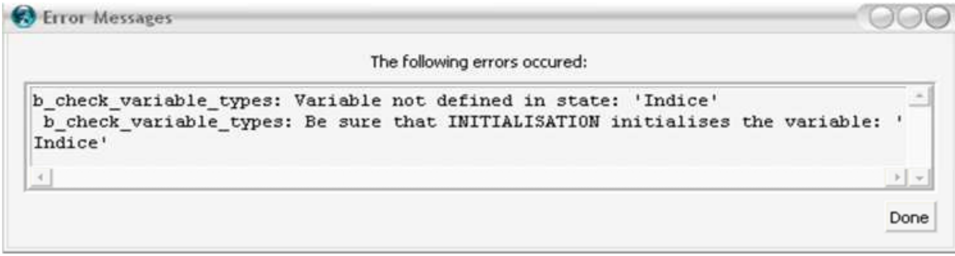


Figure 3.24 **-** Erreur de La machine abstraite ***Etage.mch***

La cause de l'erreur dans cette machine est ce que ***Indice*** est omis de la section de l'initialisation, parce que **ProB** ne peut pas déterminer (ou estimer) une valeur initial pour ***Indice***.

Maintenant si nous chargeons la machine ***Etage_a.mch*** :

```
MACHINE
    Etage_a
ABSTRACT_VARIABLES
    Indice
INVARIANT
    Indice : 0..5
INITIALISATION
    Indice := 6
OPERATIONS
    inc =
    PRE Indice < 5
    THEN
        Indice := Indice + 1
    END;
    dec =
    PRE Indice > 5
    THEN
        Indice := Indice - 1
    END

END
```

Figure 3.25 **-** La machine abstraite *Etage_a.mch*

La section de **INVARIANT** déclare que les ***Indice*** doivent inclure dans le domaine de ***0*** à ***5***. Si nous initialisons la machine, **Enabled Operations** sera faite la mise à jour pour montrer que l'invariant est violé.

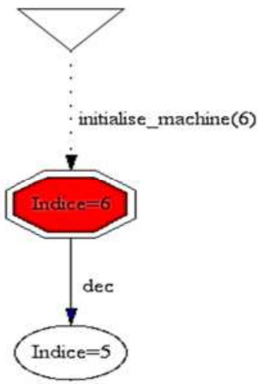


Figure 3.26 **-** Une violation de l'invariant dans *Etage_a.mch*

Dans la figure on note que l'invariant a été violé (nœud en rouge), et que le nœud au-dessous de l'état courant (***Indice = 5***) est en rouge (dehors ligne) parce que c'est un ouvert nœud (***unvisited node***). **Enabled Operations** pour le nœud courant (***Indice=6***) inclut seulement deux opérations: **BACKTRACK** et ***dec***.

3.1.3. Avantages de ProB

- **ProB** offre les deux fonctions en même, la première est l'animation et la deuxième est de jouer le rôle comme d'un model-checking.
- **ProB** capable de spécifier en graphe le modèle Event-B.
- **ProB** fait automatiquement la technique de ***constraint-solving*** en trouvant les propres valeurs qui sont satisfaisantes a tous les axiomes de système (machine).

3.1.4. Inconvénients de ProB

- **ProB** manque d'adresser quelques problèmes d'animations connues. Par exemple, il exige que tous les ensembles doivent être donnés une cardinalité finis ou nombres mathématiques entiers ne peuvent pas être énumérés à l'extérieur de leurs bornes supérieures et inférieures.

3.2. Exemple de system Event-B

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

Pour illustrer l'approche Event-B, nous proposons de détailler la démarche de conception d'un modèle. Partant de l'idée de modéliser un canal de communication, un concepteur Event-B sera généralement amené à en représenter informellement les comportements (figure 3.27). Il souhaite, dans un premier temps, considérer trois actions principales : l'envoi d'un message, son traitement et l'abandon d'un traitement. Il choisit par exemple de caractériser l'événement ***Envoyer***, qui émet, en une fois, un message composé de ***TailleEnvoi*** éléments, qui sont ensuite reçus un par un par l'événement Traiter. Une réception peut être annulée à tout moment par l'événement ***Reset***.

La phase de modélisation B peut alors commencer. La spécification est un exemple de spécification répondant au cahier des charges initiales. Cet exemple sera utilisé dans les chapitres suivants pour illustrer nos travaux portant sur le Event-B.

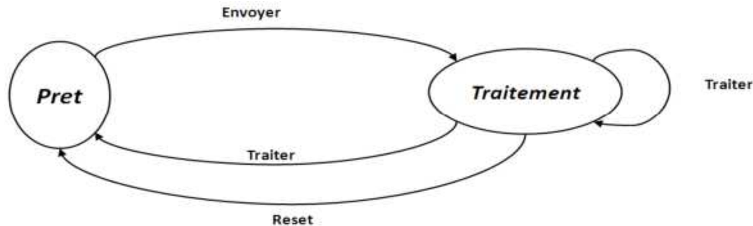


Figure 3.27 **-** Dessin informel représentant les comportements attendus

```
SYSTEM CanalComm
VARIABLES TailleEnvoi
INVARIANT TailleEnvoi : NAT
INITIALISATION TailleEnvoi :=0
EVENTS
    Envoyer =
    SELECT TailleEnvoi = 0
    THEN TailleEnvoi :: NAT1
    END ;
    Traiter =
    SELECT TailleEnvoi > 0
    THEN TailleEnvoi :=TailleEnvoi - 1 END ;
    Reset =
    SELECT TailleEnvoi > 0
    THEN TailleEnvoi :=0
    END
END
```

Figure 3.28 **-** La machine abstraite ***CanalComm***

Dans cet exemple, si la variable ***TailleEnvoi*** n'est pas nulle, donc s'il reste des données envoyées et non traitées, alors les gardes des deux événements ***Traiter*** et ***Reset*** sont vérifiés. Le choix de l'événement à exécuter est alors fait de manière non-déterministe.

4. Rodin

Rodin est une Plate-forme basée sur **Eclipse**[1] pour Event-B, cette Plate-forme fournit un support efficace pour le raffinement et les preuves mathématiques. La plate-forme est une open source, contribue à la structure d'**Eclipse** [32].

Rodin permet de créer un modèle formalisé Event-B avec un éditeur. Il génère des obligations ***POs***[2] soit automatiquement ou interactivement.

4.1. L'éditeur de l'Event-B

Une fois un contexte ou une machine est créé, une fenêtre apparaît dans la zone de l'édition montrée dans la figure suivante[3].

```
CONTEXT
    Cntxt1
CONSTANTS
    NbreProcs
AXIOMS
    axm1:   NbreProcs ⊆ ℕ          not theorem
    axm2:   NbreProcs ≠ ∅          not theorem
    axm3:   finite(NbreProcs)      not theorem
    axm4:   ∃ a · ∀ x · x ∈ NbreProcs ⇒ a ≥ x     not theorem
END
```

Figure 3.29 **-** L'éditeur de l'Événement-B

L'éditeur nous permet de modifier les éléments de la modélisation du machine (ou du contexte), variables, invariants et les constantes …etc.

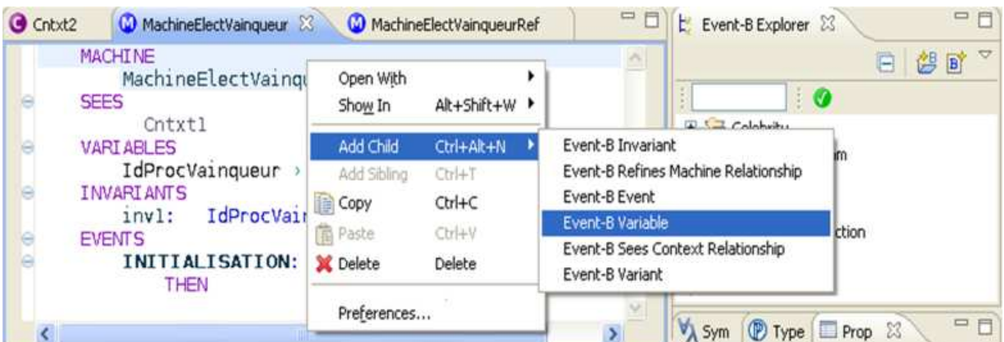


Figure 3.30 **-** Ajouter un nouvel élément de la modélisation

[1] **Eclipse :** est un environnement du développement de logiciel multi-langage qui comprend un environnement du développement automatisé (IDE) et un système de plug-in extensible. Il est écrit principalement par Java .Eclipse est Plate-forme open source.

[2] ***POs***: Proof obligations: Une obligation est doit prouver pour montrer la consistance de la machine, la convenance des théorèmes, etc. cet obligation consiste un nom(**Label**) et le nombre des hypothèses utilisées dans les preuves et dans le but(**Goal**).

[3] L'éditeur décrit ici est l'éditeur par défaut dans Rodin 2.4

Et la suppression d'un élément de la modélisation, fait par un clic sur l'élément de la modélisation et choisie élimination(**Delete**).

C'est aussi possible d'ajouter des éléments de la modélisation en utilisant des assistants.

4.2. L'éditeur structurel de Event-B

L'éditeur décrit ici était l'éditeur par défaut de **Rodin 2.3**. C'est encore disponible dans **Rodin 2.4**. Pour utiliser cet éditeur, On fait un clic sur notre machine ou fichier du contexte dans l'Explorateur de l'Événement-B et choisit **Open with → Event-B Machine Editor**.

Une fenêtre apparaître dans la zone de l'édition montrée dans la figure suivante :

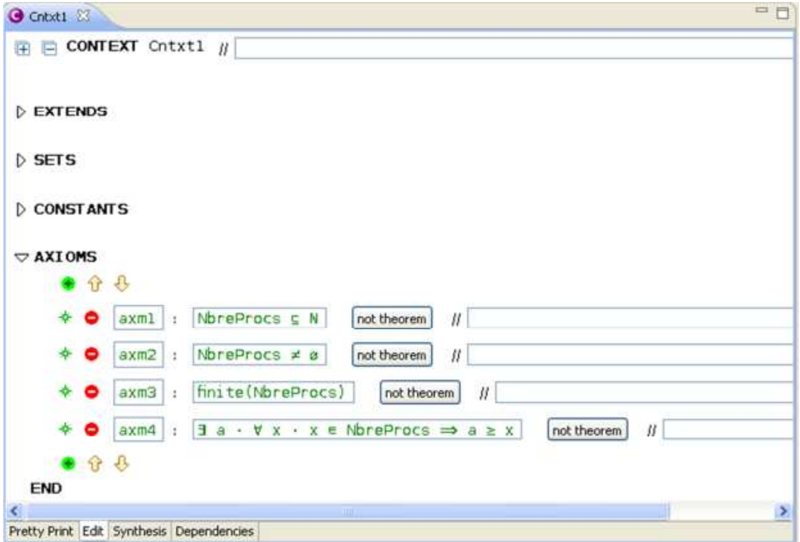


Figure 3.31 - L'éditeur de l'Événement-B Structuré

Par default, nous somme dans le mode **Edit** qui nous permet de modifier les éléments de la modélisation. Par un clic sur le triangle ⊳ qui apparaît avant chaque élément (voire la figure précédente).

Nous pouvons voir les différents éléments de la modélisation et aussi les options de : Ajouter (**add**), supprimer (**remove**), ou les déplacer (**move them**).

4.2.1. Dépendances (Dependencies)

En sélectionnant la tabulation **Dependencies** au fond de l'éditeur de l'Événement-B, nous obtenons la fenêtre dans la figure suivante :

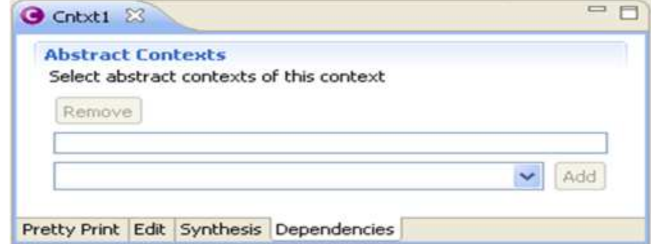


Figure 3.32 - Les Dépendances tabulent de l'éditeur de l'Événement-B d'un contexte

Cette tabulation permet de contrôler les autres machines que la machine courant étend (**Extend**). Pour ajouter la machine que nous voulons étendre, sélectionnons le nom de la machine dans le menu déroulé (**Abstract Machine**) au-dessus de la fenêtre et alors cliquons sur le bouton de l'Addition.

Il y a aussi une autre façon de créer un nouveau. Sélectionnons le contexte dans la fenêtre du projet et pressons la clé droite de la souris, choisi **Extend** dans le menu qui ouvre. Cela devrait amener la fenêtre comme la figure suivante montre.

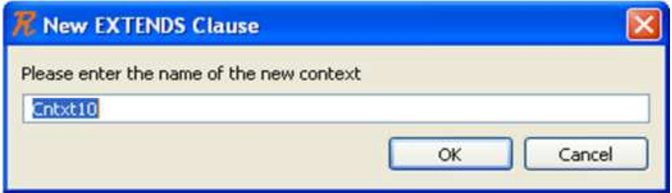


Figure 3.33 **-** Fenêtre de La clause **EXTENDS**(**Include**)

Nous pouvons entrer alors le nom du nouveau contexte qui étendra notre contexte choisi automatiquement.

De la même façon, les **Dependencies** pour une machine que celui dans le contexte. Avec La principale différence, qu'il y a deux genres de dépendances qui peuvent être établies: les machines dans lesquelles la machine courante dépend sont listées dans la partie supérieure de la fenêtre, et les contextes dans lesquels la machine courante dépend est listée dans la partie inférieure.

Figure 3.34 **-** Les Dépendances tabulent de l'éditeur de l'Événement-B d'une machine

4.2.2. Synthèse (Synthesis)

Sélectionner la tabulation de la Synthèse amène a un affichage global des éléments de notre contexte (ensemble **set** / constante **cst** / axiome **axm**) comme il est démontré dans la figure suivante.

Figure 3.35 **-** La tabulation de la Synthèse de l'éditeur de l'Événement-B d'un contexte

En pressant l'ensemble, **cst**, ou l'**axm** boutonne dans le coin supérieur droit, nous pouvons éliminer respectivement les ensembles, constantes ou axiomes de notre contexte respectivement.

Si nous sélectionnons un élément, nous pouvons changer sa priorité en pressant le bouton ⇧ ou le bouton⇩. Nous faisons ceci pour axiomes, ensembles, constantes et contextes étendus.

Le clic du droit dans cet affichage amènera a un menu du contexte qui nous permet d'ajouter un nouvel ensemble, constante, axiome ou contexte étendu.

La tabulation de la Synthèse pour les machines est fait de la même façon qu'avec les contextes mais nous avons un affichage global des éléments de notre machine (variables **var**, invariants **inv**, gardes **grd**, les paramètres des éléments **prm**).

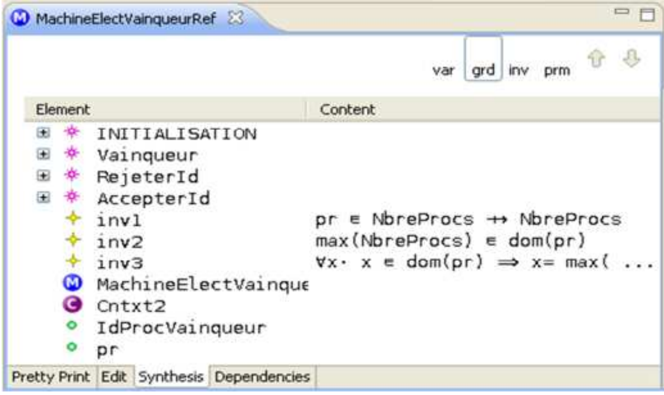


Figure 3.36 **-** La tabulation de la Synthèse de l'éditeur de l'Événement-B d'une machine

4.2.3. Pretty Print

Sélectionner **Pretty Print** nous donne un affichage global de notre contexte ou machine, été entré à travers d'un fichier de textes de l'entrée comme vu dans la figure suivante.

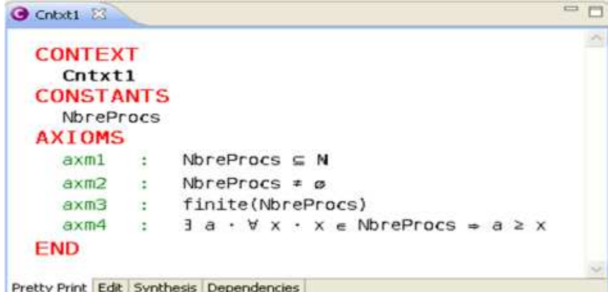


Figure 3.37 **-** La tabulation **Pretty Print** de l'Impression de l'éditeur de l'Événement-B

4.3. Assistants(Wizards)

On utilise pour être capable d'introduire directement des éléments de la modélisation dans l'éditeur, c'est aussi possible de les ajouter en utilisant des assistants (**Wizards**).

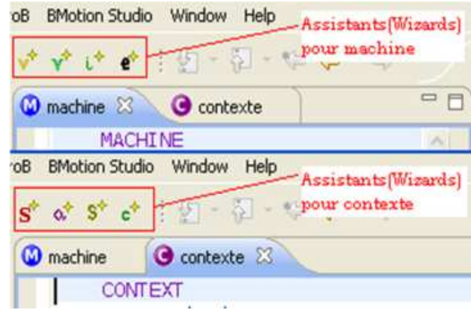


Figure 3.38 **-** Assistants pour les machines et les contextes

4.3.1. Création d'un Nouvel assistant des ensembles

Pour activer le nouvel opérateur assistant des ensembles, pressons le bouton dans la barre d'outils. Cette opération amène la fenêtre montrée dans la figure suivante :

Figure 3.39 **-** Nouvel opérateur assistant des ensembles

Et la même façon pour activer le nouvel opérateur assistant des ensembles énumérés, en pressons le bouton :

Figure 3.40 - Nouvel opérateur assistant des ensembles énumérés

L'avantage d'utiliser cet assistant est de créer l'ensemble et ses éléments, l'assistant crée automatiquement l'axiome qui est nécessaire pour le contexte pour travailler. Par exemple, quand nous ajoutons un nouvel ensemble ***Tampon*** et les deux éléments (**Plein, Vide**), l'assistant crée automatiquement l'axiome ***partition (Tampon, {Plein}, {Vide}).***

4.3.2. Création d'un Nouvel assistant des constants

Pour activer le nouvel assistant des constantes, pressons le bouton (voire la figure suivante).

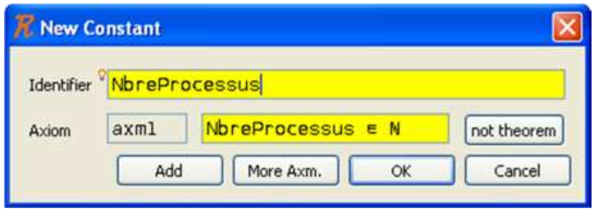


Figure 3.41 - Nouvel assistant des constantes

Dans l'exemple, on créée une constante ***NbreProcessus***, sous l'axiome que ***NbreProcessus*** ∈ ℕ

4.3.3. Création d'un Nouvel assistant des axiomes

Pour activer le nouvel assistant des axiomes, pressons le bouton de la barre d'outils. Presser le bouton amène a la fenêtre dans la figure suivante :

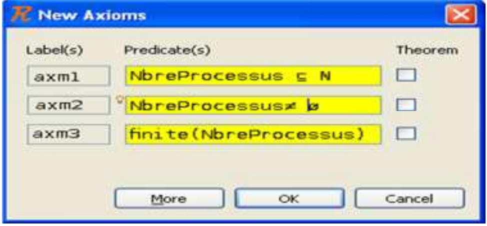


Figure 3.42 - Nouvel Assistant des axiomes

Nous pouvons entrer alors les axiomes que nous voulons. Si plus d'axiomes sont exigés, pressez le bouton **More**. Et puis vérifions la case **Theorem,** si l'axiome que nous avions créé devrait être marqué comme un théorème ou non.

Dans la figure, il y a un ensemble finie non vide des processus appartient.

4.3.4. Création d'un Nouvel assistant des variables

Pour activer le nouvel assistant variable, pressons toujours le bouton dans la barre d'outils. Presser le bouton amène a la fenêtre dans la figure suivante :

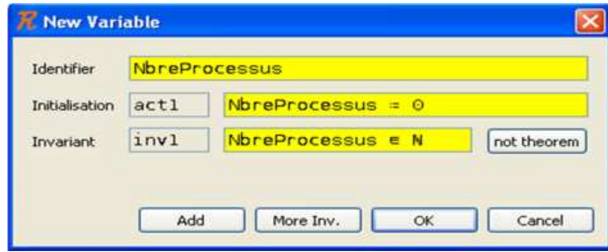


Figure 3.43 **-** Nouvel Assistant des variables

Nous pouvons entrer alors le nom de la variable, et son état d'initialisation (dans l'exemple `NbreProcessus` initialisé a 0), et un invariant lequel définit son type (dans l'exemple `NbreProcessus` est nombre naturel). On peut ajouter d'autres variables en pressons sur **Add** (pour les variables) et **More Inv** (pour plus d'invariants).

4.3.5. Création d'un Nouvel assistant des invariants

Pour activer le nouvel assistant des invariants, pressons le bouton dans la barre d'outils. Presser le bouton amène la fenêtre dans la figure suivante :

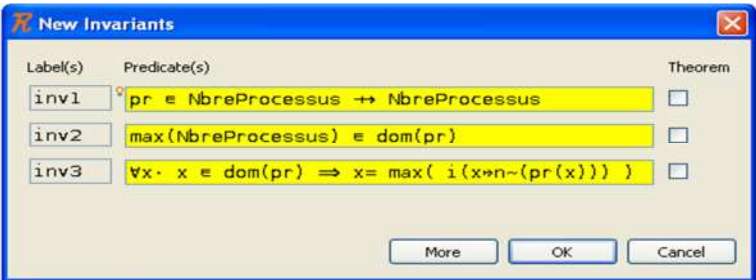


Figure 3.44 **-** Nouvel Assistant des invariants

Nous pouvons entrer alors les invariants que vous voulez. Si plus d'invariants sont exigés, pressez le Plus bouton. Vérifiez la case à cocher du Théorème pour indiquer que l'invariant correspondant devrait être marqué comme un théorème.

Dans l'exemple, il y a la fonction ***pr*** appartient à l'ensemble ***NbreProcessus*** vers ***NbreProcessus***.

Avec :

- ***NbreProcessus*** ⇸ ***NbreProcessus:*** Est l'ensemble des fonctions partielles de l'ensemble ***NbreProcessus*** dans ***NbreProcessus***. D'une autre façon c'est un cycle, de sorte que le processus qui finit l'exécution de code va directement appartenant de l'ensemble d'arrive ***NbreProcessus*** pour éviter la boucle dans le cycle.
- **max(*NbreProcessus*) ∈ dom(*pr*)** : Signifie que l'identité de chaque nouveau processus doivent appartient de l'ensemble de départ de la fonction ***pr***.
- $\forall \boldsymbol{x}.\ \boldsymbol{x} \in \boldsymbol{dom}(\boldsymbol{pr}) \Rightarrow \boldsymbol{x} = \mathbf{max}(\boldsymbol{i}(\boldsymbol{x} \mapsto \boldsymbol{n} \sim (\mathbf{pr}(\mathbf{x}))))$: signifie que, pour tous les valeurs de $\boldsymbol{x}$ ∈ **dom (*pr*)** implique que $\boldsymbol{x}$ pu soit le maximum de tous les identités dans **dom(*pr*).** Sachant que $\boldsymbol{n} \sim (\mathbf{pr}(\mathbf{x}))$ signifie que $\boldsymbol{n}$ est application inverse de la fonction ***pr(x),*** et $\boldsymbol{x} \mapsto \boldsymbol{n} \sim (\mathbf{pr}(\mathbf{x}))$ est tous les valeurs de $\boldsymbol{x}$ qui vérifiant cette application inverse. et $\boldsymbol{i}$ la fonction appartient à l'ensemble ***NbreProcessus*** × ***NbreProcessus*** vers ***NbreProcessus*** × ***NbreProcessus***.

4.3.6. Création d'un Nouvel assistant d'un événement

Pour activer le nouvel assistant des événements, pressons le bouton dans la barre d'outils. Presser ce bouton amène la fenêtre dans la figure suivante :

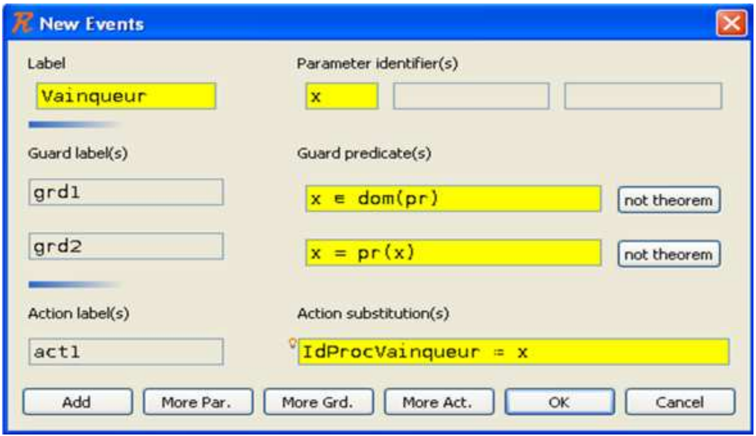


Figure 3.45 **-** Nouvel Assistant d'un événement

Nous pouvons entrer donc les événements que nous voulons. Comme les éléments suivants indiqués: nom(*Vainqueur*), paramètres ($\boldsymbol{x}$), gardes ($\boldsymbol{x} \in \boldsymbol{dom}(\boldsymbol{pr}), \boldsymbol{x} = \boldsymbol{pr}(\boldsymbol{x})$), et actions (***IdProcVainqueur***). Plus de paramètres, de gardes et d'actions peuvent être entrées en pressant les boutons correspondants.

✓ **Note :** L'ordre de l'emplacement des axiomes et les théorèmes est très important, parce que la démonstration d'un théorème ou bien le degré de chaque expression est bien définit **(well-defined : WD)** dépend des axiomes et les théorèmes qui sont déjà écrites. C'est nécessaire pour éviter le raisonnement circulaire.

4.4. La perspective de preuve

Quand obligations (***POs***) n'est pas exécuté automatiquement, dans Rodin on peut essayer de l'exécuter en utilisant l'option de **Prove Interactively** en cliquant par le bouton droit de souris sur le contexte.

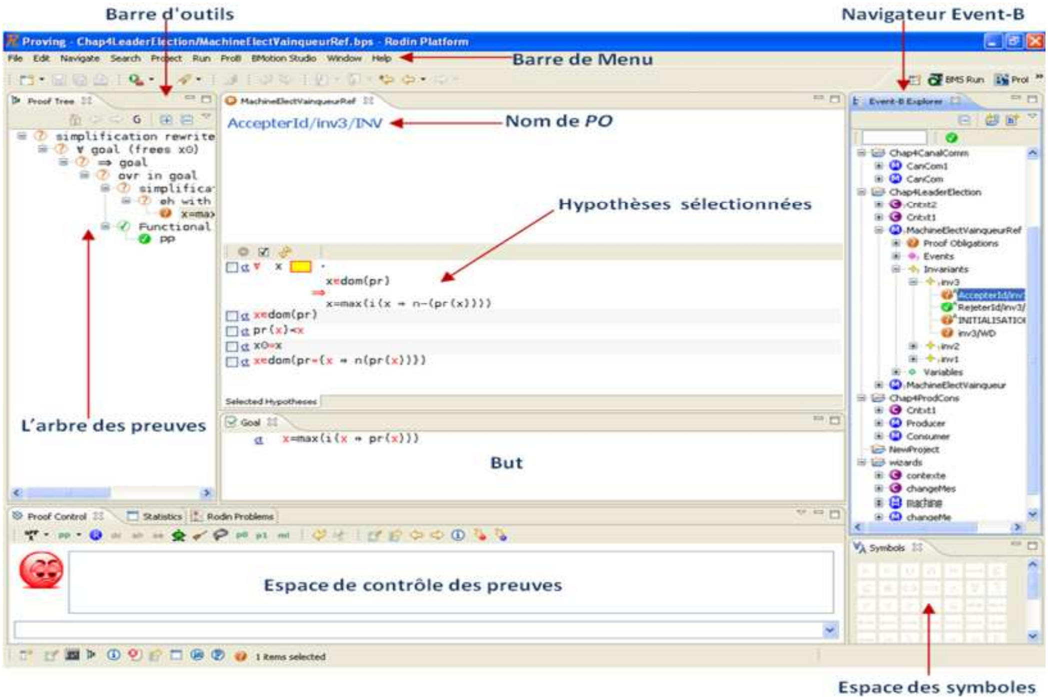


Figure 3.46 **-** Présentation de la **Prove Interactively**

4.4.1. L'arbre de démonstrations

Il faut d'abord charger le ***PO,*** Sélectionnons le projet du navigateur de l'Événement-B et choisi le composant (contexte ou machine). Choisir par la suite (double-cliquez) sur l'obligation ***PO*** qui nous intéresse.

L'affichage de l'arbre des preuves (démonstrations) fournit une représentation graphique de chaque preuve :

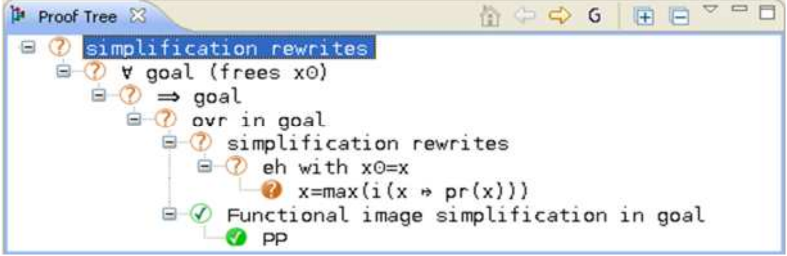


Figure 3.47 **-** L'arbre de démonstrations

L'arbre est composé des séquences. Chaque ligne de l'arbre est une descendante d'un autre. Chaque séquence est étiquetée avec un commentaire qui indique quelle

règle s'est appliquée ou quels prouveur ont chargé la preuve. Avec un clic sur une séquence de l'arbre les hypothèses du cette séquence sont chargées à la fenêtre des **Selected Hypotheses**.

Voici quelques significations sur symboles utilisées dans **Prover Perspective** :

- indique que cette séquence est bien prouvée.
- indique que cette séquence n'est pas définit et non prouvée.
- indique que cette séquence doit examiner ou réviser au début parce que la séquence n'est pas chargée par le prouveur.

4.4.2. But et hypothèses sélectionnées

Dans la barre de **Goal** on trouve quelques hypothèses le prouveur doit les prouvée, et on a la possibilité d'éviter certaines d'elles selon nos choix. On les trouvées a coté de **Search Hypotheses** , on décrire quelques boutons les plus utilisés :

- Sélectionnée tous les hypothèses.
- Inversée la sélection.
- Démonstration par contradiction 1: On a la possibilité de démontré les hypothèses par la négation dans cette cas la, la négation de **Goal** viennent d'être une hypothèse sélectionnée et **Goal** soit "⊥".

4.4.3. Contrôleur de démonstrations

La barre du Contrôleur de démonstrations contient les boutons avec lesquels nous pouvons exécuter une preuve interactive.

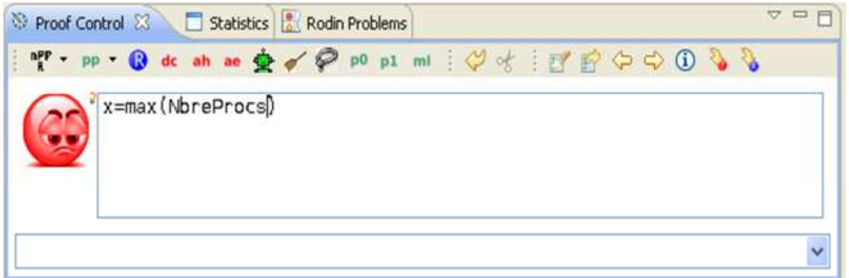


Figure 3.48 **-** Contrôleur de démonstrations

Les boutons suivants sont disponibles dans l'affichage du Contrôle Insensible :

- invoque le nouveau prouveur du prédicat. Une liste déroulante indique des stratégies alternatives ;
- Les prouveurs externes peuvent être invoqués de la liste déroulante pour tester les séquences ;

- ah Ajoute une nouvelle hypothèse. Le prédicat dans la zone de l'édition devrait être prouvé par l'utilisateur. Il est ajouté alors comme une nouvelle hypothèse sélectionnée ;
- Est le prouveur automatique, appliqué automatiquement sur toutes les obligations de la preuve après un la machine ou le contexte a enregistré ;
- Charge les hypothèses cachées qui contiennent des identifiants en commun avec le but et avec les hypothèses sélectionnées dans la fenêtre des **Selected Hypotheses** ;
- Déplacements au prochain nœud en attente de l'arbre de démonstrations courant ;
- Charge le prochain nœud examiné de l'arbre de démonstrations courant.
- Le **Simley** : dans trois couleurs :

    - Le rouge indique que l'arbre de démonstrations contient un ou plus séquences non prouvée.
    - Le bleu indique que tous les séquences de l'arbre de démonstrations doit examiner au début.
    - Le vert indique que tous les séquences de l'arbre de démonstrations sont prouvées.

5. Conclusion

L'**Atelier B**, **ProB** sont deux logiciels permettant la verification de la specification d'un programme ou un protocole écrit par le langage B. **ProB** il permet aussi de détecter les interblocages et permet de détecter les expressions indéfinies.

Rodin, il a la possibilité de compromise ces deux derniers dans la même plate-forme, et même autres options supplémentaires, comme **AnimB**, **B2Rodin**, et **Brama**…etc.

Chapitre 4

APPLICATIONS ET DISCUSSION DE RÉSULTATS

1. Introduction

Dans ce chapitre nous allons voir, à titre d'application de spécification et vérification avec les outils **Atelier B** et **ProB** et **Rodin** et le langage B les deux algorithmes suivants: Le premier est celui de l'élection de leader de **Herman**, le deuxième est l'algorithme de consommateur et producteur de **Lamport**. Mais avant ces deux spécifications on a choisi de traiter un exemple simple d'un protocole de communication de la façon générale pour montrer comment spécifier un algorithme de systèmes répartis.

2. Un protocole de communication simple

On commence ce chapitre par une spécification d'un protocole de communication simple entre un nombre finie des agents, pour montrer comment spécifier un problème d'un système distribué par les méthodes B.

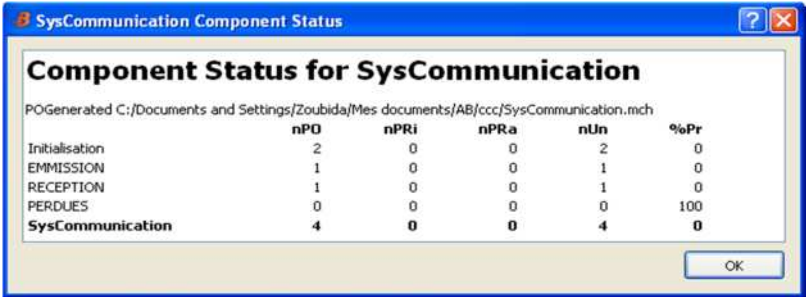
2.1. Principe de protocole

Le principe de ce protocole(ou système), est un échange des messages entre les agents de ce système. La partie dynamique contient trois opérations (***EMISSION, RECEPTION, PERDUES***).

Un agent ***agnt1*** communique un message ***msg*** à un autre agent ***agnt2*** en ajoutant le triplet (***agnt1, msg, agnt2***) à l'ensemble des valeurs envoyées c'est-à-dire ***envoyer***, l'événement d'envoi est appelé ***EMMISSION***. Aussi, on peut modéliser la réception, en ajoutant ce triplet à l'ensemble ***recue***. On suppose qu'il n'y a pas de perte de message.

2.2. Résultat de la spécification

Après l'utilisation d'**Atelier B** la vérification par ***Tc*** et ***P0*** et ***B0 check*** nous donne le statut de la ***SysCommunication***:



SysCommunication Component Status

Component Status for SysCommunication

POGenerated C:/Documents and Settings/Zoubida/Mes documents/AB/ccc/SysCommunication.mch

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	0	2	0
EMMISSION	1	0	0	1	0
RECEPTION	1	0	0	1	0
PERDUES	0	0	0	0	100
SysCommunication	**4**	**0**	**0**	**4**	**0**

OK

Figure 4.1 - La statut de la machine abstraite ***SysCommunication***

Notons qu'il y a quatre(04) ***POs*** non prouvées: deux dans l'initialisation et un dans l'émission et l'autre dans l'opération de la réception. Et la figure 4.2 suivante montre les détails de ces ***POs*** (**nUn**).

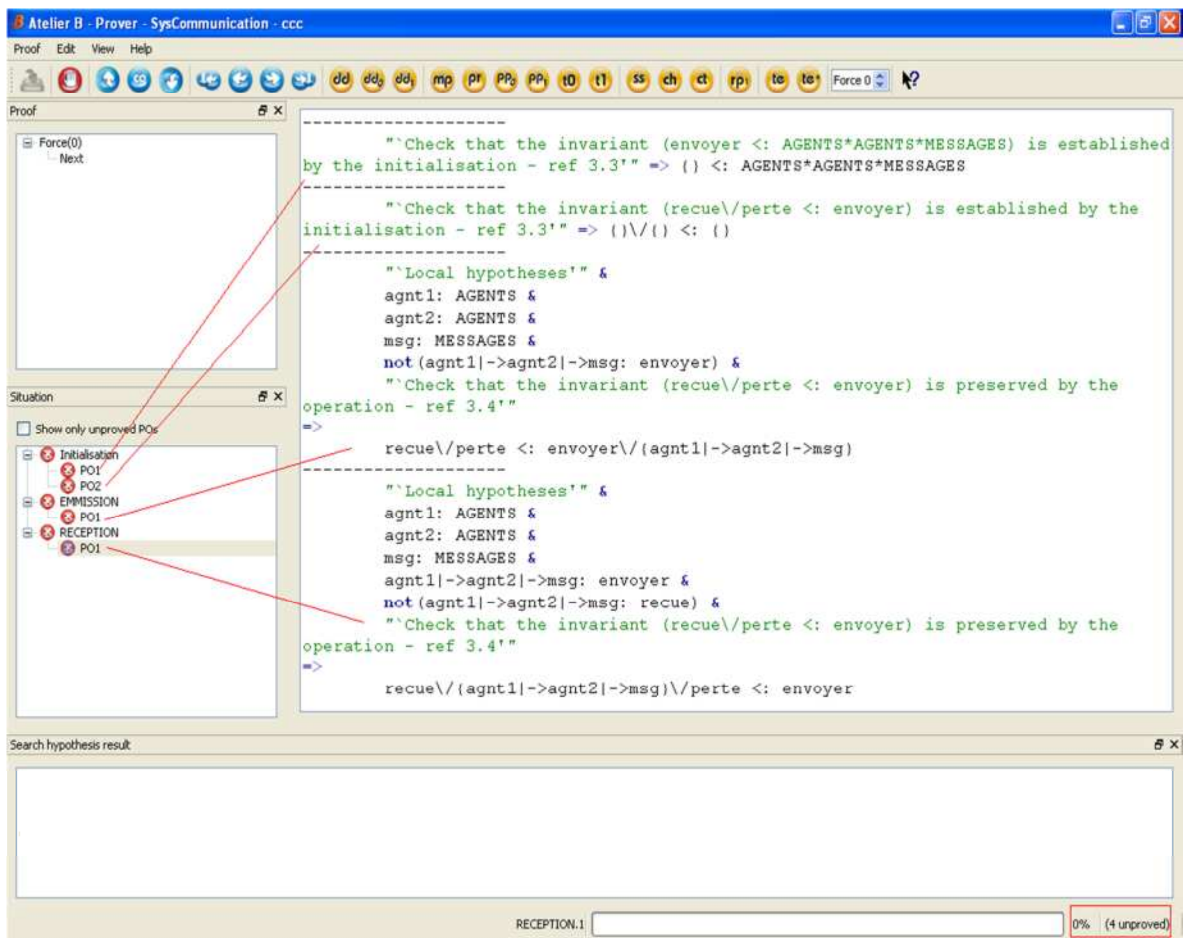


Figure 4.2 **-** Un extraire de SysCommunication.po de la machine abstraite *SysCommunication*

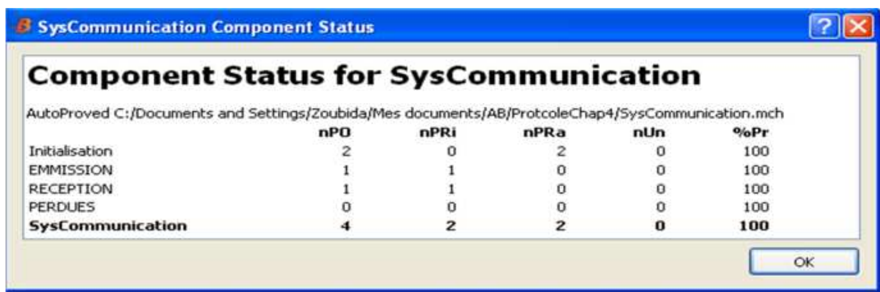
✓ Note : Il y a réellement (02) ***POs*** **(uUN)** : ***P01*** et ***P02*** dans l'initialisation, sont répétées dans les deux opérations : ***EMMISIION***, ***RECEPTION***

2.3. Discussion du résultat

La première exécution de la vérification, par l'outil **Atelier B**, nous indique que la spécification comporte 04 obligations ***POs*** non prouvées (***P0*** et ***P1***).

- Pour ***P01*** : L'initialisation de l'ensemble ***envoyer*** par l'ensemble vide fait la contradiction avec les propriétés des agents et les messages qu'on a (AGENTS $\neq\emptyset$, MESSAGES $\neq\emptyset$).
- Pour ***P02*** : L'initialisation des deux ensembles : ***recue*** et ***perte*** par l'ensemble aussi vide, fait la contradiction avec l'ensemble ***envoyer*** qui est initialisé par l'ensemble vide ($\emptyset \cup \emptyset \subset \emptyset$: l'union de deux ensembles vides est sous-ensemble l'ensemble vide), selon les caractères de l'union et le sous-ensemble c'est une erreur mathématique.

Pour corriger ces deux obligations, on utilise le prouveur "*pr*", pour chaque ***PO(P01,P02)***. Et la figure 4.3 suivante montre le statut des composantes du ***SysCommunication*** :


SysCommunication Component Status

Component Status for SysCommunication

AutoProved C:/Documents and Settings/Zoubida/Mes documents/AB/ProtcoleChap4/SysCommunication.mch

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	2	0	100
EMMISSION	1	1	0	0	100
RECEPTION	1	1	0	0	100
PERDUES	0	0	0	0	100
SysCommunication	**4**	**2**	**2**	**0**	**100**

OK

Figure 4.3 - La statut de la machine abstraite ***SysCommunication*** après la démonstration par "*pr*"

Maintenant en utilise le **ProB**, pour vérifier sil y a de violation ou non. Et on vérifie par **Model Check** : le résultat montre que il n'ya pas d'erreur, et tous les nœuds sont visités.

Figure 4.4 - La statut de la machine abstraite ***SysCommunication*** après la démonstration par **Model Check**

3. Algorithme d'élection d'un leader

De nombreux algorithmes de contrôle bien que dit distribués sont en faite centralisée dans le sens où un processus joue un rôle particulier ou coordinateur. Il effectue certains services à la demande des autres.

Un inconvénient majeur peut arriver, la panne de processus en fonctionnement peut s'entende entre eux pour décider le quel d'entre eux jouera désormais le rôle de coordinateur. Ceci passera des algorithmes d'élection.[3 3]

Nous avons pris le problème de l'élection du leader[34]. Dans l'exemple courant, La machine de l'événement-B ***MachineElectVainqueur*** montre la spécification de cet algorithme. Pour un nombre fini de processus arrangés dans un anneau, chaque processeur a une **IDENTIFICATION** numérique. Le processeur avec la plus haute valeur numérique est choisi comme un ***leader***.

3.1. Principe de L'algorithme de l'élection d'un leader

Cet algorithme s'intéresse au cas où les processus sont placés dans un anneau unidirectionnel. Chaque processus P_i est noté de manière unique par un numéro qui l'identifie.

Le principe de l'algorithme est simple. Il s'agit de trouver le maximum d'un ensemble de numéros. Pour cela chaque processus P_i transmet son propre numéro j à son voisin de gauche, celui-ci, à la réception du message, compare son numéro j au numéro reçu i. Donc si i est la plus grande valeur, il le transmet à son voisin, sinon c'est son propre numéro j qui est envoyé. Le processus qui reçoit un message d'élection portant son numéro, on conclut que son numéro a fait le tour de l'anneau, et est le plus grand numéro. Il est élu et diffuse son identité aux autres processus si cela est nécessaire. De manière formelle l'algorithme est défini comme dans la machine ***MachineElectVainqueur***.

3.2. Résultat de la spécification

Après la spécification de problème d'élection d'un leader, on note quelques obligations qui ne sont pas prouvées par le prouveur automatique. Donc nous avons besoin d'utiliser les autres prouveurs que **Rodin** offre.

Les résultats de l'exécution de spécification de l'algorithme d'élection de leader sont illustrés par les captures d'écran que présente la figure 4.5 suivantes :

3.2.1. Création du contexte *Cntxt1* et la machine abstraite *MachineElectVainqueur*

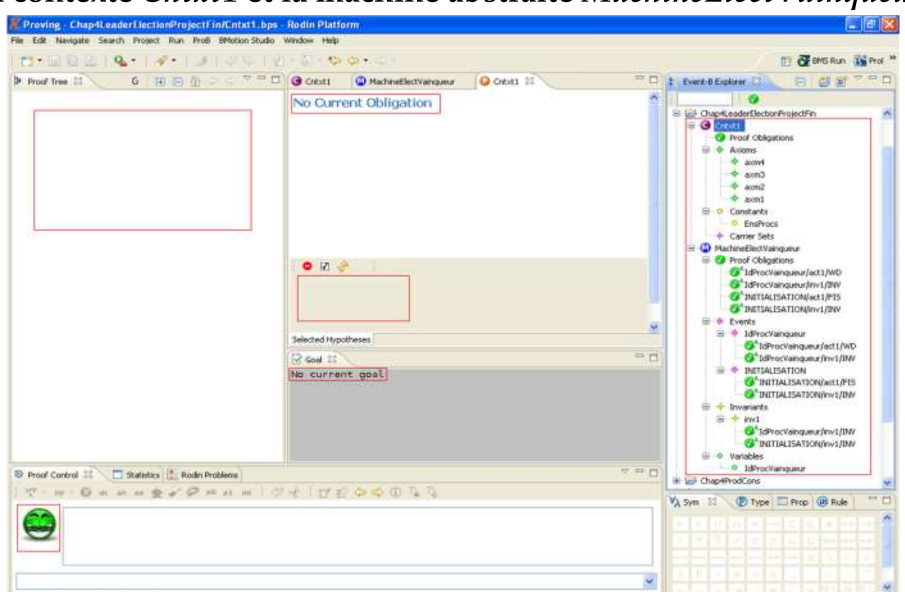


Figure 4.5 **-** Résultats de spécification après création ***Cntxt1*** et ***MachineElectVainqueur***

Cette capture d'écran montre six résultats concernant :

- Le navigateur de Event-B : Tous les éléments du contexte et la machine aussi sont mentionnés par ;

- Le contrôleur des preuves : vient on couleur de verte ;
- L'arbre de démonstration : Notons qu'il ya aucun arbre, aucune obligation et pas des hypothèses (donc pas de **Goal**).

3.2.2. Création du contexte *Cntxt2* et la machine abstraite *MachineElectVainqueurRaf*

A. Création du contexte *Cntxt2*

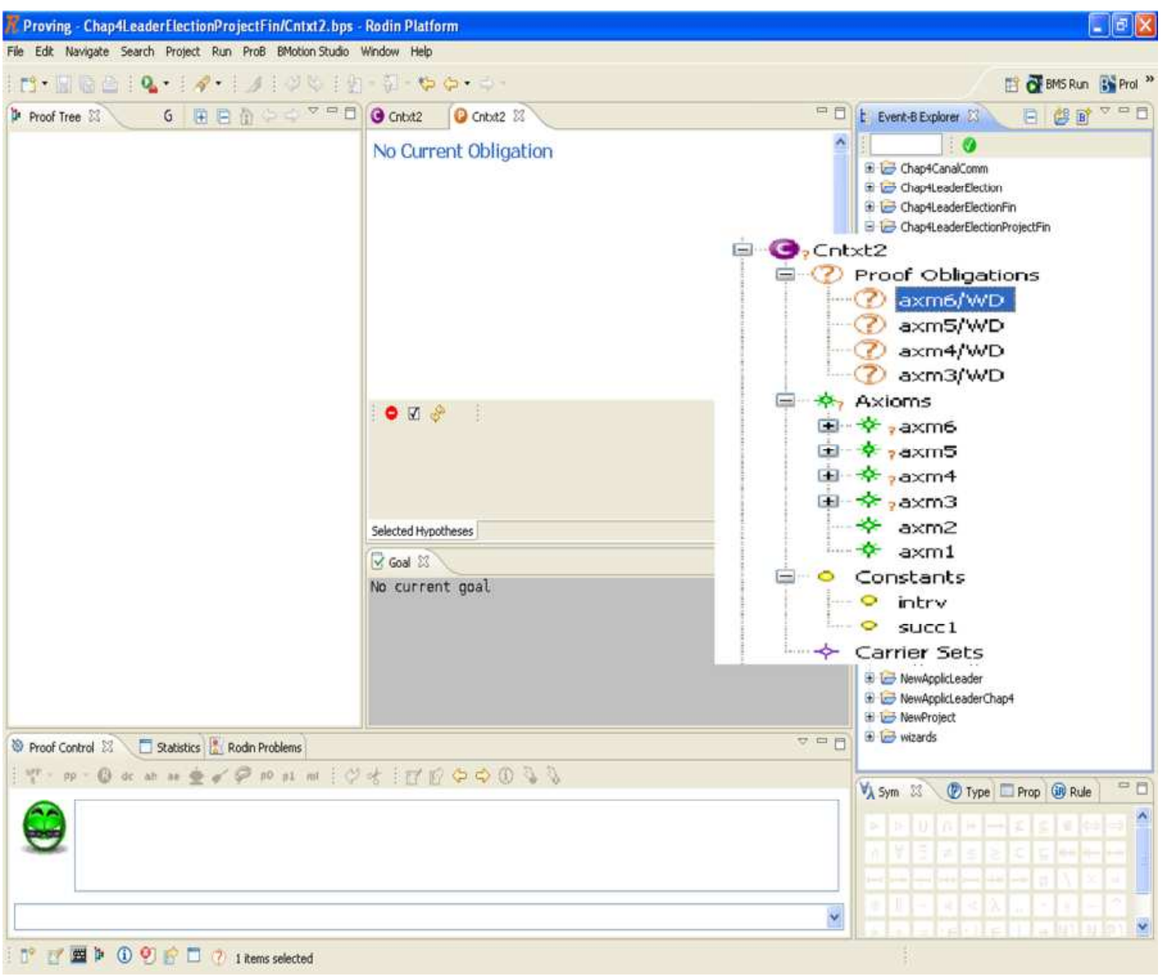


Figure 4.6 **-** Résultats de spécification après création *Cntxt2*

La figure 4.6 représente les mêmes résultats précédents mais avec une différence qui concerne le navigateur de **Event-B**. Alors les obligations ***POs*** ne sont pas vérifiés par le prouveur automatique au niveau des axiomes **axm(3 ; 4 ; 5 ; 6)/WD**[1].

[1]**WD**: well-definedness proof.

B. Création du contexte ***MachineElectVainqueurRaf***

Figure 4.7 **-** Résultats de spécification après la création de ***MachineElectVainqueurRaf***

La figure 4.7 montre les mêmes résultats que les précédents, mais au niveau de **inv2/INV** et **inv3/WD** .Donc les obligations ***POs*** ne sont pas générés car il y a des éléments dans l'événement (***IdProcReject***) et les invariants dans le même événement.

3.3. Discussion du résultat

3.3.1. Pour le contexte *Cntxt1* et la machine abstraite *MachineElectVainqueur*

Pour ces deux (***Cntxt1, MachineElectVainqueur***), la spécification est bien définit sur les ensembles utilisés.

D'un côté du contexte ***Cntxt1***, on crée un ensemble fini non vide ***EnsProcs***. Et d'un autre côté de la machine ***MachineElectVainqueur***, on définit un processus qui soit le vainqueur ***IdProcVainqueur*** par l'application de la fonction prédéfinie **max()**.Il n'y a pas de complication dans ces deux composantes, alors le prouveur automatique exécuter sur elles sans problème de démonstration.

3.3.2. Création du contexte *Cntxt2* et la machine abstraite *MachineElectVainqueurRaf*

A. Pour le contexte ***Cntxt2***

- **axm6/WD** :

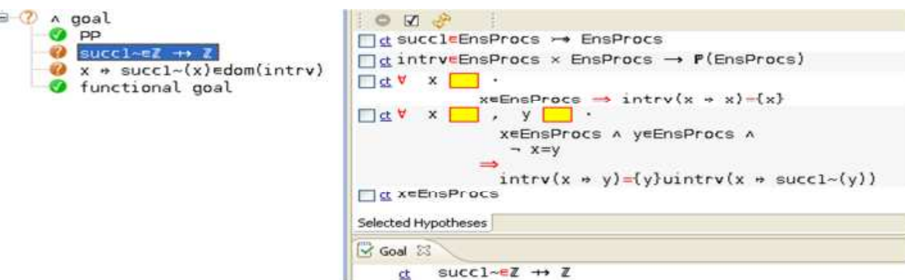


Figure 4.8 **-** Obligation POs (**axm6/WD**) non prouvée

Comme la figure 4.8 montre, il y a deux obligations ***PO*** non prouvées concernant l'axiome 6 **axm6** :

$$\forall x \cdot x \in EnsProcs \Rightarrow intrv(x \mapsto succ1{\sim}(x)) = EnsProcs$$ …(voir le code source)

Notons que la fonction **succ1** est une fonction dans les intervalles de nombres naturels ℕ, mais le prouveur automatique ne peut pas vérifier le cas que **succ1** appartienne à l'ensemble ℤ et même le cas pour la deuxième obligation. Alors on utilise manuellement les prouveurs extérieures **p0**, **p1**, **ml**… etc. successivement[1].

Dans ce cas, on essaye le premier prouveur **p0** deux fois successivement et l'obligation a été démontée dans le premier temps comme la figure 4.9 suivante montre.

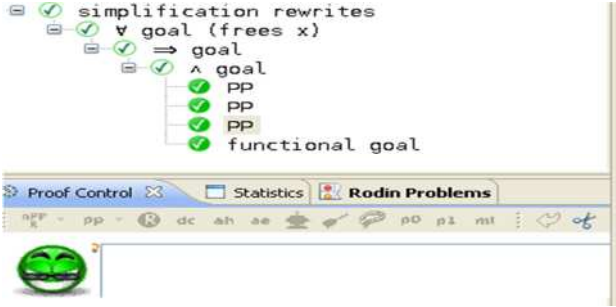


Figure 4.9 **-** Après l'utilisation du prouveur **p0**

Et pour les deux axiomes restants (**axm5, axm4**), on fait le même. Voir les figures suivantes.

- **axm5/WD** :

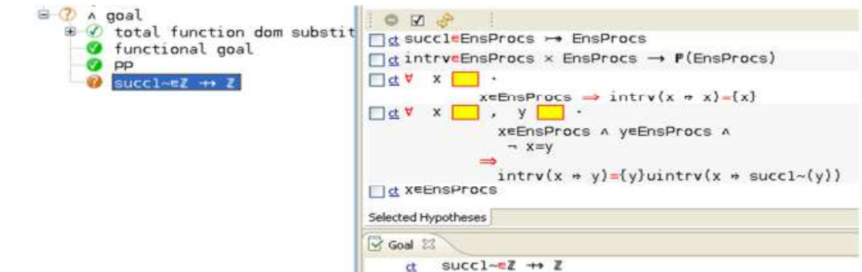


Figure 4.10 **-** Obligation POs (**axm5/WD**) non prouvée

[1] Ces prouveurs sont utilisées par ordre, car les démonstrations sont faites par le degré de chaque prouveur.

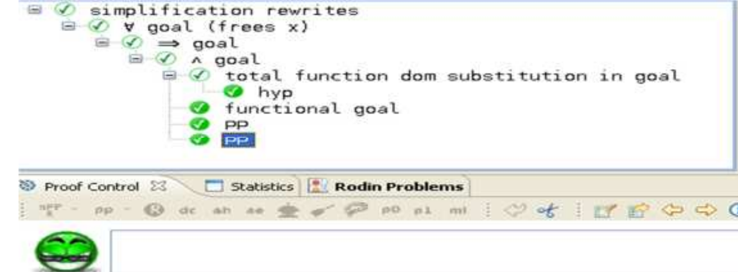


Figure 4.11 **-** Après l'utilisation du prouveur **p0**

- **axm4/WD** :

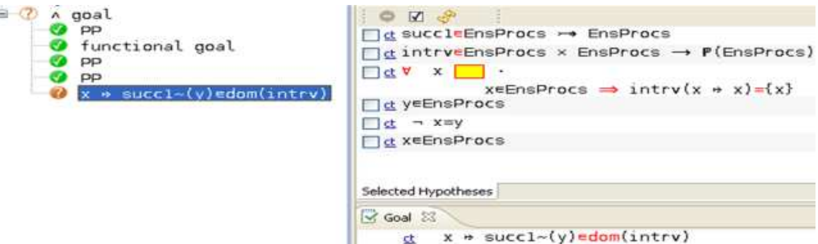


Figure 4.12 **-** Obligation POs (**axm4/WD**) non prouvée

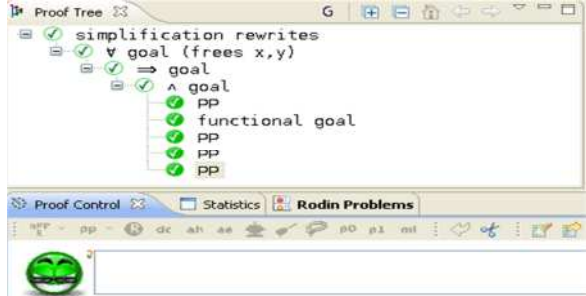


Figure 4.13 **-** Après l'utilisation du prouveur **p0**

B. Pour la machine ***MachineElectVainqueurRaf***

- **inv2/INV** :

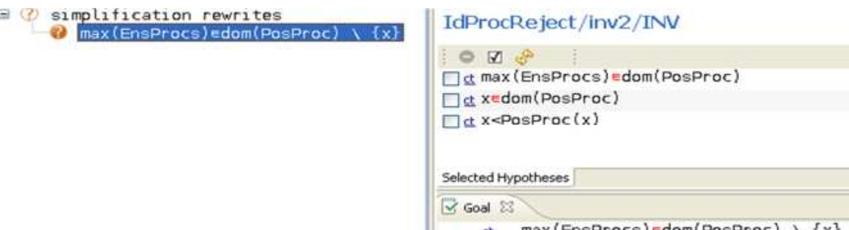


Figure 4.14 **-** Obligation POs (**inv2/INV**) non prouvée

La figure 4.14 montre l'invariant 2 suivant:

$$\mathbf{max}(\boldsymbol{EnsProcs})\in \mathbf{dom}(\boldsymbol{PosProc})\backslash\{y\}$$

Alors, on a un invariant non prouvé **inv2** dépend directement avec l'opération ***IdProcReject***, cette dernière doit prouver avec le prouveur extérieur.

Dans ce cas-là, on utilise le prouveur **p0** deux, mais cette fois rien changer et on observe que ce prouveur n'a pas pu réaliser la démonstration.

On essaye une autre fois avec le prouveur **p1**, et cette fois la démonstration se fait correctement. La figure 4.15 suivante montre le contrôleur de démonstration en vert.

Figure 4.15 **-** Après l'utilisation du prouveur **p1**

- **inv3/WD** :

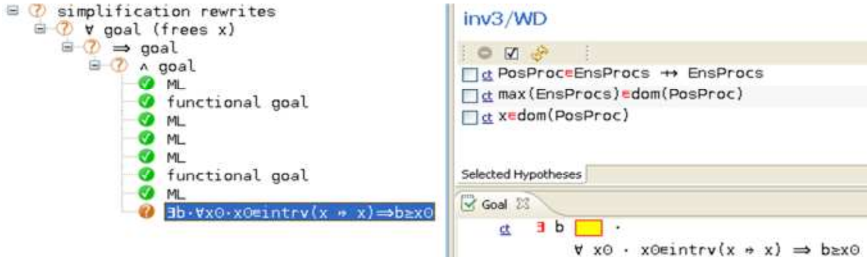


Figure 4.16 **-** Obligation POs (**inv3/WD**) non prouvée

Cette fois ci, la figure 4.16 montre que l'invariant **inv3** est non prouvé parmi les obligations ***POs*** de la machine ***MachineElectVainqueurRaf***.

$$\forall x \cdot x \in \mathrm{dom}(PosProc) \Rightarrow x = \max(\, intrv(x \mapsto succ1 \sim (succ1(x))))$$

Le prouveur doit vérifier que cet invariant doit vérifier l'invariante suivant (la barre de **Goal**) :

$$\exists\, \boldsymbol{b}.\forall\, \boldsymbol{x0}\,.\boldsymbol{x0} \in \boldsymbol{intrv}(\boldsymbol{x} \longmapsto \boldsymbol{x}) \Longrightarrow \boldsymbol{b} \geq \boldsymbol{x0}$$

Il doit vérifier qu'il existe un nombre $\boldsymbol{b} \in \mathbb{N}$, et pour les valeurs de $\boldsymbol{x} \in (\boldsymbol{x} \longmapsto \boldsymbol{x})$ alors $\boldsymbol{b} \geq \boldsymbol{x0}$.

Le prouveur automatique générer cette obligation ***PO***, il n'est plus capable de la prouver. Donc le prouveur extérieure **p0** a pu le faire cette fois.

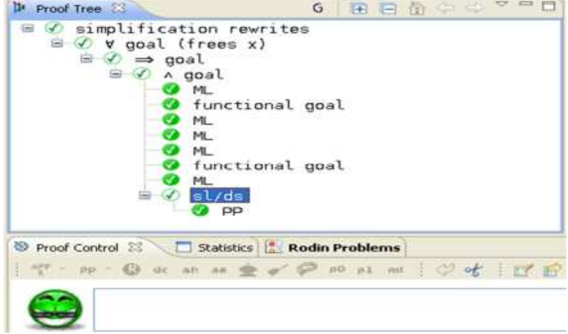


Figure 4.17 **-** Après l'utilisation du prouveur **p0**

4. Algorithme de consommateur et de producteur

Nous prenons l'exemple classique du producteur-consommateur. C'est un exemple simple et largement utilisé pour illustrer les différents concepts de concurrence et de synchronisation.

Lamport est le premier qui proposa cet algorithme qui évite les problèmes de la synchronisation dans le cas d'un seul producteur et un seul consommateur.[35]

4.1. Principe de L'algorithme de producteur/consommateur

L'idée principale est l'exécution concurrente d'un processus consommateur qui retire une donnée d'une zone mémoire (tampon ou buffer) à une place. Cette dernière est remplie par un autre processus producteur. Le consommateur ne peut pas consommer un tampon vide et le producteur ne peut pas remplir un tampon plein. Donc chacun d'entre eux (producteur, consommateur) doivent permuter (alterner) leurs tâches.

4.2. Décomposition du système

Dans la suite nous montrons comment ce système peut être construit en composant de deux sous-systèmes distincts.

4.2.1. Répartition des variables et les invariants

Les variables et les invariants du système abstrait ***ProdCon*** peuvent être répartis sur la base des variables utilisées par les événements ***produire*** et ***consommer***. Il y a des variables communes entre les deux événements précédents (voir la figure 4.18 suivante).

```
MACHINE
    Producteur ›
SEES
      Cntxt1
VARIABLES
    Tampon ›
    etatTampon ›
INVARIANTS
    inv1:   Tampon ∈ DATA      not theorem  ›
    inv2:   etatTampon ∈ Etat      not theorem  ›
EVENTS
    INITIALISATION:     not extended ordinary ›
        THEN
            act1:   Tampon :∈ DATA ›
            act2:   etatTampon = vide  ›
        END
```

Figure 4.18 **-** Variables et invariants partagés

4.2.2. Variables partagés et substitutions multiples

Lorsque les substitutions partagent le même espace de variables la composition de systèmes abstraits correspond à l'opérateur plus général de fusion de **BACK** et **BUTLER**.

4.2.3. Les deux sous systèmes

On peut décomposer notre système ***ProdCon***, en deux autres sous-systèmes (***Producteur, Consommateur***).

Et à chaque sous-système, il y a une opération propre avec les variables partagés qui sont existent dans les deux sous-systèmes.

4.3. Résultat de la spécification

4.3.1. Le système abstraite *ProdCon*

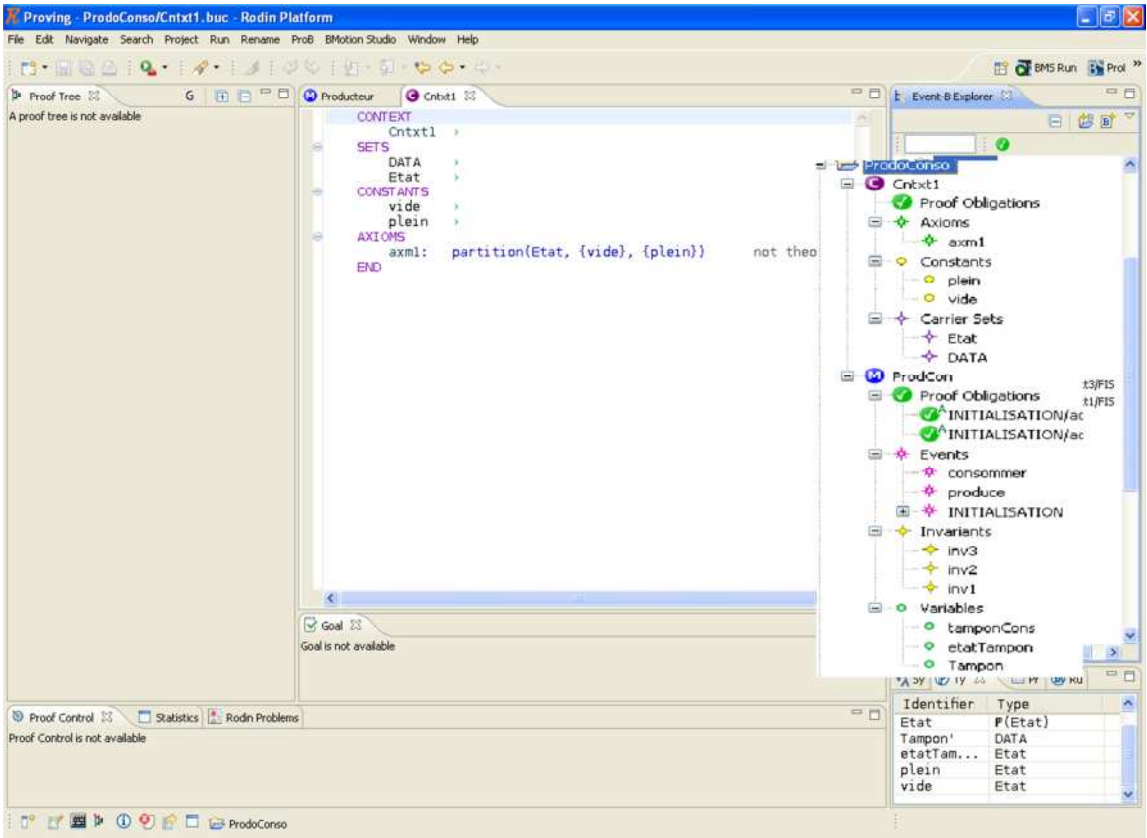


Figure 4.19 - Résultats de spécification après la création de ***ProdCon***

Cette figure (figure 4.19) montre le résultat de la spécification après la création du ***ProdCon.***

4.3.2. Les sous systèmes abstraites *Producteur, Consommateur*

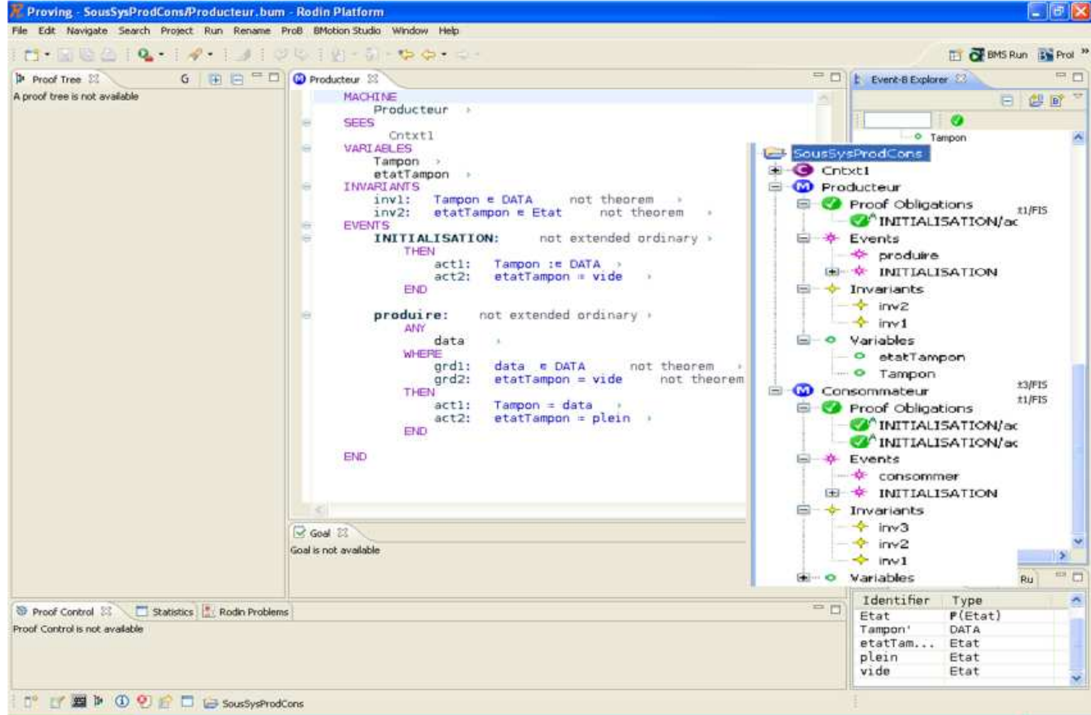


Figure 4.20 - Résultats de spécification après la création de ***Producteur et Consommateur***

4.4. Discussion de résultat

Nous avons présenté dans cette brève spécification une technique générale pour structurer des systèmes abstraits concurrents en les composants parallèlement en utilisant **Rodin**.

Notre spécification traite plus généralement les cas de variables partagés et d'échanges de messages.

Donc la figure 4.20 indique, la spécification dans les deux cas (***ProdCon ; Consommateur*** et ***Producteur***) est bien définit, et les obligations ***POs*** sont générées correctement avec le prouveur automatique.

5. Conclusion

Dans ce chapitre, nous avons traité comment spécifier un algorithme d’un système réparti avec la méthode B, en particulière Event-B. Ces deux algorithmes sont: L’algorithme de l’élection et l’algorithme de producteur/consommateur. Le premier est déjà spécifier et vérifier mais avec plus des détails dans [36]. Mais nous voulons juste de démontré l’utilisation des outils de la méthode B pour avoir comment valider les protocoles des systèmes répartis.

L’un de points importants à noter dans ce chapitre, concerne les deux algorithmes de l’élection et producteur/consommateur qui ne présentaient aucunes anomalies lors de la démonstration par les outils utilisées dans **Rodin** par rapport les deux autres outils(**Atelier B**, **ProB**).

CONCLUSION ET PERSPECTIVES

Le travail de ce mémoire s'est déroulé dans le cadre d'utilisation des méthodes formelles pour les systèmes repartis. Il s'intéresse plus particulièrement aux méthodes formelles B.

Nous avons commencé par une étude générale concernant ces méthodes et leurs existants. Nous avons étudié les différents types : les méthodes informelles, les méthodes semi-formelles et les méthodes formelles qui contiennent elle-même des diverses méthodes basées sur des langages.

Nous avons basé sur la méthode B en particulier la méthode Event-B, et nous avons trouvé que l'Event-B est une méthode et un langage adéquat pour l'ingénierie de domaine. Sa philosophie générale est bien adaptée à notre objectif dans la specification, , verification, validation de protocoles dans les systèmes repartis.

Nous avons ensuite proposé que l'outil **Rodin** soit meilleur que **ProB** et **Atelier B**, car il est plus adéquat dans les spécifications des protocoles dans les systèmes répartis qui sont basés sur les événements, en outre est un outil encapsulé les deux outils précédents dans son corps, en plus il contient autres outils **AnimB**, **B2Rodin**, … .

Pour pouvoir traiter l'utilisation de Event-B, nous avons décomposée le 4eme chapitre en trois parties : la première est de traité un protocole de communication simple pour voir comment écrite un protocole de communication entre deux agents communicant une message avec la méthode classique B en utilisant l'**Atelier B**. Cette specification ne résulte aucune erreur.

La deuxième et la troisième partie, étudient les deux algorithmes connus : l'algorithme de l'élection d'un leader dans un cycle contient un nombre fini des processus, et l'algorithme de producteur/consommateur qui doit vérifier les deux tâches par la synchronisation entre la production des données et leur consommation dans un tampon (zone mémoire), ces deux algorithmes sont écrites en utilisant **Rodin**. Le résultat de specification ne donne aucune anomalie, sauf des erreurs dans l'ordonnancement des invariants.

Nous savons que ces algorithmes ne sont qu'une 1ère étape de specification et il reste les raffinements de chaque machine, les modifications et les améliorations dans ces derniers sont toujours possibles. C'est la raison pour laquelle nous avons pensé à des améliorations et des perspectives futures:

1. L'utilisation des méthodes formelles est exploitable pour les réseaux avec tous ces types, et surtout le domaine de VANET, Ad-Hoc.

2. Il faut se concentrer dans les prochains travaux sur le raffinage, ce dernier est une étape importante dans les spécifications, car avec cette étape notre outil généré le code source de plusieurs langages de programmation comme Ada et C.

BIBLIOGRAPHIE

[1] Denis Oddoux. Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéraires. Thèse doctorat. Université Paris 7 - Denis Diderot UFR d'Informatique. Décembre 2003. (Cité page 03)

[2] J.L.Lions et autres. Inquiry Ariane 5 : Flight 501 Failure. Rapport de commission. 19 Juillet 1996. (Cité page 04)

[3] Sare Robinson. Beyond 2000: Further Troubles Lurk in the Future of Computing. The New York Times. 19 juillet 1999. (Cité page 04)

[4] Michel Héon et Josianne Basque et Gilbert Paquette. Validation de la sémantique d'un modèle semi-formel de connaissances avec OntoCASE. 21èmes 21es Journées Francophones d'Ingénierie des Connaissances. France (2010). (Cité page 05)

[5] Martin Gogolla et Uwe Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. ACM Transactions on Database Systems. Mars 1976 . (Cité page 07)

[6] Habrias H. Dictionnaire encyclopédique du Génie Logiciel. Masson. 1997. (Cité page 07)

[7] John Miles Smith et Diane C. P. Smith. Database Abstractions: Aggregation and Generalization, ACM Transactions on Database Systems. Juin 1977. (Cité page 08)

[8] Hong phuong nguyen. Dérivation de spécifications formelles B à partir de spécifications semi-formelles. Thèse doctorat. Conservatoire national des arts et métiers CNAM. Décembre 1998. (Cité pages 08, 09, 27,28 et 30)

[9] Petko Valtchev. Méthode Z. Université de Montréal. Octobre2003. (Cité page 09)

[10] Pascal André. Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions. Thèse doctorat. Université de Rennes1. Juillet 1995. (Cité page 09)

[11] Phil Stocks et Kerry Raymond et David Carrington et Andrew Lister. Modeling Open Distributed System in Z. Livre : Computer Communications, Volume 15 1992. (Cité page 09)

[12] R. Milner. A Calculus of Communicating Systems.1980. (Cité page 11)

[13] Hubert Garavel. Introduction au langage LOTOS. Revue de l'Association des Anciens Elèves de l'ENSIMAG. 1990. (Cité page 12)

[14] Éric Jaeger. Study of the Benefits of Using Deductive Formal Methods for Secure Developments. Thèse doctorat. 2010. (Cité page 16)

[15] Abdelhafid Zitouni. Utilisation des design Patterns et des méthodes formelles dans le développement des systèmes d'information. Thèse doctorat. Université Mentouri Constantine. Décembre 2008. (Cité page 16)

[16] Bilel Gargouri et Mohamed Jmaiel et Abdelmajid Ben Hamadou. Vers l'utilisation des méthodes formelles pour le développement de linguiciels. 2002. (Cité page 16)

[17] Sébastien Bardin. Introduction au Test Logiciel. Tunisie 2011. (Cité page 17)

[18] Z.Mammeri. Introduction aux méthodes et cycle de développement du logiciel. 2011. (Cité page 17)

[19] Tomas Barros. Formal specification and verification of distributed component systems. Thèse doctorat. Université de Nice-Sophia Antipolis - UFR Sciences. Novembre 2005. (Cité page 17)

[20] Régine Laleau. Conception et développement formels d'applications bases de données. Thèse d'une habilitation à diriger des recherches. Université d'Evry Val d'Essonne. Décembre 2002. (Cité page 18)

[21] J. R. Abrial. The B-Book: Assigning Programs to Meanings. 1996. ISBN 0-521-49619-5. (Cité page 21)

[22] Frédéric Gervais. EB4 : Vers une méthode combinée de spécification formelle des systèmes d'information. Université de Sherbrooke Conservatoire National des Arts et Métiers. Examen de spécialité. Juin 2004. (Cité page 21)

[23] fr.wikipedia.org. consulter en Mai 2012. (Cité page 21)

[24] www.methode-b.com. Consulter Mai 2012. (Cité page 22)

[25] Nicolas Stouls. Systèmes de transitions symboliques et hiérarchiques pour la conception et la validation de modèles B raffinés. Thèse doctorat. Institut polytechnique de Grenoble. décembre 2007. (Cité page 23)

[26] Jean-Marie Portal. Les méthodes formelles garantes de la sécurité des systèmes. Magazine : 01Informaique numéro 1646. 14 septembre 2001. (Cité page 24)

[27] Les méthodes formelles sonnent le glas des tests unitaires. Enquête. Magazine : 01Informaique numéro 1662. 18 janvier 2002. (Cité page 24)

[28] http ://www.methodeb.com. Consulter Mai 2012. (Cité page 26)

[29] Jens Bendispost et Michael Leuschel et Olivier Ligot et Mireille Samia, La validation de modèles Event-B avec le plug-in ProB pour RODIN. 2008. (Cité page 40)

[30] ClearSy. The B formal Method: from Research to Teaching. Nantes : Juin 2008. (Cité page 43)

[31] Michael.A. Leuschel et M.Butler et S.Lo Presti. ProB User Manuel of ProB. Version 1.1.4. En collaboration de l'université Southampton(Angleterre) et l'université Düsseldorf (Allemagne). Avril 2005. (Cité page 43)

[32] Michael Jastram. Rodin: User's Handbook. Version 2.4. Deploy Project. Janvier 2012. (Cité page 59)

[33] M.Djoudi. Support de cours: Systèmes répartis : Algorithme d'élection. université de Laghouat. 2012. (Cité page 73)

[34] T. Herman. Probabilistic Self-Stabilization: Information Processing Letters, vol.35. (Cité page 73)

[35] Leslie Lamport. Specifying concurrent program modules. ACM Trans. 1983. (Cité page 80)

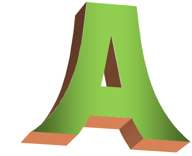
[36] Alexei Iliasov et Linas Laibinis et Elena Troubitsyna et Alexander Romanovsky. Formal Derivation of a Distributed Program in Event B. October 2011. (Cité page 82)

ANNEXES:

A. Interview Avec **J**ean-**R**aymond **A**brial: L'inventeur de la méthode formelle B….. **89**

B. Eléments du langage de la méthode B ……………………………………………….. **90**

C. Codes sources des algorithmes ……………………………………………………….. **92**



A.1. Interview Avec **J**ean-**R**aymond **A**brial: L'inventeur de la méthode formelle B

Jean-Raymond Abrial, consultant indépendant et inventeur de la méthode formelle B

« Le zéro défaut n'existe pas mais on peut tout de même s'en approcher »

Les méthodes formelles sont en mesure de garantir des logiciels plus fiables. Pourtant, elles ont encore du mal à percer.

Bien que permettant d'accroître la fiabilité des logiciels, les méthodes formelles sont encore peu utilisées. Comment l'expliquez-vous ?

Il est possible que certains éditeurs n'aient pas intérêt à ce que leurs logiciels fonctionnent correctement. Je pense, en effet, que c'est ce qui se passe pour certains logiciels grand public. Entre autres, parce que beaucoup d'utilisateurs aiment bien bricoler. Et quand quelque chose marche bien, cela n'est pas très intéressant pour eux. Les dysfonctionnements d'un logiciel peuvent donc être voulus, car considérés comme un plus pour le produit. Mais je n'évoque ici que le bon côté des choses. Je ne parle pas des professionnels qui le font exprès pour que le client paye ensuite la maintenance. Utiliser les méthodes formelles avec cette idée en tête ne présente donc aucun intérêt. Car le logiciel tournerait correctement. Par contre, s'en servir pour garantir qu'un train sans conducteur, ou un satellite, remplit sa mission peut avoir un sens. Les enjeux ne sont ici, à l'évidence, plus du tout les mêmes.

Les méthodes formelles impliquent de s'attarder sur les spécifications au détriment des tests. C'est une remise en cause du cycle de développement...

Dans d'autres disciplines de l'ingénieur, comme la construction aéronautique ou le génie civil, il existe l'équivalent des méthodes formelles. C'est la notion de bureau d'étude. On y réalise un dessin – le modèle du futur objet – sur lequel on raisonne selon des théories. Ce sont les techniques propres à l'ingénieur. L'idée de raisonner longtemps avant sa finalisation fait partie de l'ingénierie moderne. Utiliser les méthodes formelles revient à faire du bureau d'étude pour les logiciels : le dessin d'un programme est un objet formel que l'on ne peut pas exécuter, mais sur lequel on peut raisonner. La plupart des programmeurs prétendent qu'il suffit d'exécuter un logiciel et de faire des tests pour s'assurer qu'il fonctionne. Comme si, pour faire un avion, vous fabriquiez quelque chose qui ressemble à un fer à repasser avec des ailes, et que vous le testiez. Cela s'est pratiqué dans l'histoire des technologies, mais c'est une approche très primitive. Quand une technologie prend de l'ampleur, on se met à raisonner sur un dessin ou un modèle.

Quelles difficultés présente l'utilisation des méthodes formelles ?

Il est difficile de vendre une nouvelle technologie à des personnes qui n'en ont pas besoin. Un industriel qui n'a pas rencontré de gros problèmes au cours de la mise en œuvre de son processus de développement est généralement peu enclin à le remettre en question. Les méthodes formelles ne se vendent bien qu'à des personnes qui ont souffert. Leur intégration dans une pratique industrielle présente deux grosses difficultés. Elles nécessitent de revoir le processus de développement pour y intégrer les éléments caractéristiques de l'approche formelle (affinement et preuve) et elles renversent le profil de financement des projets : au lieu d'avoir des dépenses qui culminent au moment des tests, un investissement non négligeable a lieu beaucoup plus en amont, dès les spécifications et la conception, voire dès l'étude système. Pour un décideur, c'est un choix difficile à prendre. Par contre, la maîtrise technique d'une méthode formelle ne pose pas de difficultés particulières.

Dans quels types de développements logiciels les méthodes formelles prennent-elles le plus facilement ?

Il y en a beaucoup. Par exemple, pour la conception des systèmes distribués. La complexité de ces derniers réside dans le fait que la notion de maître n'existe pas. Dans un protocole de routage, un nœud du réseau ne prend pas de décision pour les autres. L'information progresse et, à chaque nœud, redétermine son chemin en fonction de la configuration locale du réseau. Il devient alors extrêmement difficile de faire des tests puisque les conditions d'exécution ne sont jamais répétitives. De fait, les méthodes formelles sont déjà en partie utilisées pour l'élaboration de protocoles de transmission, de routage ou cryptographiques. Elles sont également de plus en plus appliquées dans un domaine plus vaste que le développement logiciel proprement dit. En l'occurrence, pour l'étude des systèmes complexes. Là aussi, il faut fabriquer un objet dont il est, à l'évidence, nécessaire de pratiquer des raisonnements très rigoureux sur le modèle.

Quels sont les pays les plus avancés dans l'utilisation des méthodes formelles ?

L'Europe est plutôt en avance dans le domaine. En particulier la France, l'Angleterre et l'Allemagne. Les Américains, quant à eux, ont beaucoup d'argent et de bonnes méthodes de travail. Les gens sont extrêmement responsables dans les projets. Tout cela compense en partie. Ce sont des technologues. Sur le plan de la sociologie professionnelle, ils accordent beaucoup d'importance à la valeur du technicien. D'ailleurs, dans les grandes compagnies, il est fréquent que de très hauts responsables soient encore des techniciens. Il existe une vraie reconnaissance de la valeur technique des individus, considérée comme quelque chose d'irremplaçable. Cela leur permet de construire des systèmes remarquables.

Les méthodes formelles sont-elles vraiment la solution miracle ?

Le zéro défaut n'existe pas. Il peut toujours se passer quelque chose. Si on place un ordinateur dans un tunnel avec plein de bruits électroniques, la valeur d'un bit peut changer spontanément. Ceux qui prétendent pouvoir atteindre le zéro défaut ne sont pas de vrais ingénieurs. Mais, on peut s'en approcher toujours plus. Un ingénieur raisonne constamment en termes de probabilité. Et c'est en connaissance de cause qu'il accepte le risque, aussi faible soit-il.

Propos recueillis par Jean-Marie Portal

G. FORLUIT

« Certains éditeurs n'ont pas intérêt à ce que leurs logiciels fonctionnent correctement »

« Raisonner sur un modèle coûte beaucoup moins cher »

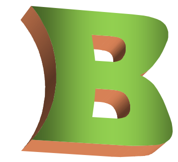
« L'Europe est plutôt en avance dans l'utilisation des méthodes formelles »

LE CONTEXTE

De nos jours, il est devenu courant, de parler de qualité logicielle, en entreprise. Paradoxalement, l'utilisation des méthodes formelles, qui garantissent une plus grande fiabilité des programmes, a du mal à se généraliser. Jean-Raymond Abrial, l'inventeur de la méthode B, donne son point de vue sur la situation.

NOTRE ANALYSE

« Les méthodes formelles si nécessaire, mais pas nécessairement. » C'est, en gros, le raisonnement tenu actuellement par les entreprises. Car cette technologie, lorsqu'elle est connue, dérange a priori quelque peu. Remettant en cause les processus habituels de développement, elle tend aussi à reléguer au placard les programmeurs. En somme, une petite révolution. Si les utilisateurs continuent à se contenter des produits qui leur sont délivrés, la situation mettra du temps à évoluer.



ANNEXE

Eléments du langage de la méthode B

B .1. Opérateurs de B (format mathématique et ASCII)

Math	Opérateur(ASCII)	Interprétation
$=, \neq$	=, /=	égalité, inégalité
$\in, \notin$	: , / :	inclusion ensembliste
$\subseteq$	< :	union ensembliste
$\cap$	/\	intersection ensembliste
$\cup$	\/	différence ensembliste
$-$	–	restriction de domaine d'une relation
$\lhd$	<|	restriction de codomaine d'une relation
$\rhd$	|>	paire ordonnée
$\mapsto$	|->	surcharge de relation
⩤	<+	Surcharge de relation
$\otimes$	><	produit direct de deux relations
r^{-1}	r~	relation inverse de r
;	;	composition en avant de relations
*	*	produit cartésien de deux ensembles
&	&	et logique
Or	Or	ou logique
$\forall$	!	quantificateur universel
$\exists$	#	quantificateur existentiel
$\leftrightarrow$	<->	ensemble des relations
$\rightarrow$	-->	ensemble des fonctions totales
⇸	+->	ensemble des fonctions partielles
:=	:=	substitution simple
||	||	substitution parallèle

B .2. Les expressions principales de la théorie des ensembles de B

Nom	Syntaxe	Définition	Pré-condition
Produit cartésien	$s \times t$	$\{x \mapsto y \mid x \in s \wedge y \in t\}$ *	
Relation	$s \leftrightarrow t$	$\mathbb{P}(s \times t)$ avec $\mathbb{P}(S)$: ensemble des parties de S	
Inverse	r^{-1}	$\{y,x \mid (y,x) \in t \times s \ (x \mapsto y) \in r\}$	$r \in s \leftrightarrow t$
Domaine	dom(r)	$\{x \mid x \in s \wedge \exists y. (y \in t \wedge (x \mapsto y) \in r)\}$	
Codomaine	ran(r)	$dom(r^{-1})$	
Composition	q;r	$\{x,z \mid (x,z) \in s \times u \wedge \exists y. (y \in t \wedge (x \mapsto y) \in q \ (y \mapsto z) \in r)\}$	
Restriction	$u \lhd r$	$\{x,y \mid (x,y) \in r \ x \in u\}$	$r \in s \leftrightarrow t \wedge u \subseteq s$
Corestriction	$r \rhd v$	$\{x,y \mid (x,y) \in r \ y \in v\}$	$r \in s \leftrightarrow t \wedge v \subseteq t$
Anti-restriction	u ⩤ r	$\{x,y \mid (x,y) \in r \ x \notin u\}$	$r \in s \leftrightarrow t \ u \subseteq s$
Anti-Corestriction	r ⩥ v	$\{x,y \mid (x,y) \in r \wedge y \notin v\}$	$r \in s \leftrightarrow t \wedge v \subseteq t$
Image	r[w]	$\{y \mid y \in t \wedge \exists x. (x \in w \wedge (x \mapsto y) \in r)\}$	$r \in s \leftrightarrow t \wedge w \subseteq s$
Produit direct**	$f \otimes g$	$\{x,(y,z) \mid (x,(y,z)) \in s \times (u \times v) \wedge (x \mapsto y) \in f \wedge (x \mapsto z) \in g\}$	$f \in s \leftrightarrow u \wedge g \in s \leftrightarrow v$
Fonction partielle	s ⇸ t	$\{f \mid f \in s \leftrightarrow t \wedge \forall x,y,z. ((x \mapsto y) \in f \wedge (x \mapsto z) \in f \Rightarrow y = z)\}$	

Fonction totale	s ⟶ t	{f | f ∈ s ⇸ t ∧ dom(f) = s}
Injection partielle	s ⤔t	{f | f ∈ s ⇸ t ∧ f^{-1} ∈ t ⇸ s
Injection totale	s ↣t	s ⤔t ǀ s ⟶ t
Surjection partielle	s ⤀t	{f | f ∈ s ⇸ t ∧ ran(f) = t}
Surjection totale	s ↠ t	s ⤀t ∧ s ⟶ t
Bijection	s ⤖t	s ↣t ∩s ↠ t

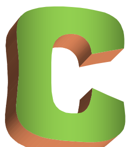
* : se lit ensemble des couples (x,y) telque x appartient à s et y appartient à t la notion "↦" correspond à la notion de couple.

B .3. Les principales substitutions généralisées

Nom	Notation mathématique	Pré-condition
Skip	Skip	
Devient égal	**X** := **E**	**X** est une variable **E** est une expression quelconque
Devient tel que	**X** : (**P**)	**X** est une variable **P** est un prédicat
Devient un élément de	**X** : ∈ **S**	**X** est une variable **S** est un ensemble
Pré-condition	PRE **P** THEN **S** END	**P** est un prédicat **S** est une substitution quelconque
Begin	BEGIN **S** END	**S** est une substitution

IF	IF **P** THEN
	S
	ELSEIF **Q** THEN
	T
	ELSE
	U
	END
		P, **Q** sont des prédicats
		S, **T**, **U** sont des substitutions quelconques

ANY	ANY **X** WHERE **P** THEN **S** END	**X** est une liste de noms simples deux à deux distincts **P** est un prédicat **S** est une substitution quelconque
Appel d'opération	[**u** ←] OP [(**v**)] la partie entre [ ] est optionnelle	
Simultanée	‖	



ANNEXE

Codes sources des algorithmes

B .1. Code source de protocole de communication avec **Atelier B**

```
/* SysCommunication
 * Author: Zoubida
 * Creation date: sam. avr. 21 2012
 */
MACHINE
   SysCommunication
SETS
   AGENTS;
   MESSAGES
PROPERTIES
   AGENTS /= {} &
   MESSAGES /= {}
VARIABLES
   envoyer,
   recue,
   perte
INVARIANT
   envoyer <: AGENTS * AGENTS * MESSAGES & /* Emmission d'un msg entre deux agents */
   recue <: AGENTS * AGENTS * MESSAGES &
   perte <: AGENTS * AGENTS * MESSAGES &
   (recue \/ perte) <: envoyer &
   perte = {} /* pas de pertes*/
INITIALISATION
   envoyer := {} ||
   recue := {} ||
   perte := {}
OPERATIONS
   EMMISSION =
   ANY agnt1, agnt2, msg
   WHERE
      agnt1 : AGENTS &
      agnt2 : AGENTS &
      msg : MESSAGES &
      agnt1 |-> agnt2 |-> msg /: envoyer
   THEN
      envoyer := envoyer \/ {agnt1 |-> agnt2 |-> msg}
   END;
   RECEPTION =
   ANY agnt1, agnt2, msg
   WHERE
      agnt1 : AGENTS &
      agnt2 : AGENTS &
      msg : MESSAGES &
      agnt1 |-> agnt2 |-> msg : envoyer &
      agnt1 |-> agnt2 |-> msg /: recue
   THEN
      recue := recue \/ {agnt1 |-> agnt2 |-> msg}
   END;
   PERDUES =
      BEGIN
         Skip  END END
```

B .2. Code source de l'algorithme de l'élection d'un leader avec Rodin

B.2.1. Le contexte **Cntxt1**

```
CONTEXT
	Cntxt1
CONSTANTS
	EnsProcs
AXIOMS
	axm1 :EnsProcs ⊆ ℕ       // EnsProcs sous ensemble de ℕ
	axm2 :∃ n,m · m ∈ 1‥n ⤖ EnsProcs       // EnsProcs un ensemble fini
	axm3 :EnsProcs ≠ ∅       // EnsProcs un essemble contient au moin 01 processus
	axm4 :∃a ·∀ x· x ∈ EnsProcs ⇒ a ≥ x
END
```

B.2.2. Le contexte **Cntxt2**

```
CONTEXT
	Cntxt2
EXTENDS
	Cntxt1
CONSTANTS
	succ1
	intrv
AXIOMS
	axm1 :succ1 ∈ EnsProcs ⤖ EnsProcs
	axm2 :intrv ∈ EnsProcs×EnsProcs → ℙ(EnsProcs)
	axm3 :∀ x · x ∈ EnsProcs ⇒ intrv(x↦x)={x}
	axm4 :∀ x, y · x ∈ EnsProcs ∧ y ∈ EnsProcs ∧ x ≠ y ⇒ intrv(x↦y) = {y} ∪ intrv(x↦succ1∼(y))
	axm6 :∀ x ·(x ∈ EnsProcs ⇒ succ1 ∼ (succ1(x))=x)
	axm7 :∀x·x ∈ EnsProcs ⇒ intrv(x↦succ1∼(x))= EnsProcs
END
```

B.2.3. La machine abstraite **MachineElectVainqueur**

```
MACHINE
	MachineElectVainqueur
SEES
	Cntxt1
VARIABLES
	IdProcVainqueur
INVARIANTS
	inv1 :IdProcVainqueur ∈ EnsProcs
EVENTS
	INITIALISATION ≙
		STATUS
	  ordinary
BEGIN
	  act1 :IdProcVainqueur :∈ EnsProcs
END
	Vainqueur ≙
		STATUS
	  ordinary
BEGIN
	  act1 :IdProcVainqueur ≔ max(EnsProcs)
END
END
```

B.2.3. La machine abstraite **MachineElectVainqueurRaf**

```
MACHINE
     MachineElectVainqueurRaf
REFINES
     MachineElectVainqueur
SEES
     Cntxt2
VARIABLES
     IdProcVainqueur
     PosProc
INVARIANTS
     inv1 :PosProc ∈ EnsProcs⇸EnsProcs
     inv2 :max(EnsProcs)∈dom(PosProc)
     inv3 :∀x·x∈dom(PosProc) ⇒ x=max( intrv(x↦succ1∼(succ1(x))))
EVENTS
     INITIALISATION ≙
       extended
          STATUS
       ordinary
BEGIN
       act1 :IdProcVainqueur :∈ EnsProcs
       act2 :PosProc ≔ succ1
END
     Vainqueur ≙
       extended
          STATUS
       ordinary
REFINES
       Vainqueur
BEGIN
       act1 :IdProcVainqueur ≔ max(EnsProcs)
END
     AcceptIdProcs ≙
          STATUS
       ordinary
ANY
       x
WHERE
       grd1 :x∈dom(PosProc)
       grd2 :PosProc(x) < x
THEN
       act1 :PosProc(x) ≔ succ1(PosProc(x))
END
     RejectIdProc ≙
          STATUS
       ordinary
ANY
       x
WHERE
       grd1 :x ∈ dom(PosProc)
       grd2 :x < PosProc(x)
THEN
       act1 :PosProc ≔ {x} ⩤ PosProc
END
END
```

B .3. Code source de l'algorithme de producteur/consommateur avec Rodin

B.3.1. Le contexte Cntxt1

```
CONTEXT
    Cntxt1
SETS
    DATA
    Etat
CONSTANTS
    vide
    plein
AXIOMS
    axm1 :partition(Etat, {vide}, {plein})
END
```

B.3.2. La machine abstraite **ProdCon**

```
MACHINE
    ProdCon
SEES
    Cntxt1
VARIABLES
    Tampon
    etatTampon
    tamponCons
INVARIANTS
    inv1 :Tampon ∈ DATA
    inv2 :etatTampon ∈ Etat
    inv3 :tamponCons ∈ DATA
EVENTS
    INITIALISATION ≙
        STATUS
      ordinary
BEGIN
      act1 :Tampon :∈ DATA
      act3 :tamponCons:∈ DATA
      act2 :etatTampon ≔ vide
END
    produce ≙
        STATUS
      ordinary
ANY
      data
WHERE
      grd1 :data ∈ DATA
      grd2 :etatTampon = vide
THEN
      act1 :Tampon ≔ data
      act2 :etatTampon ≔ plein
END
    consommer ≙
        STATUS
      ordinary
WHEN
      grd1 :etatTampon = plein
THEN
      act1 :tamponCons ≔ Tampon
      act2 :etatTampon ≔ vide
END
END
```