Université de Montréal

Méthodes formelles de haut niveau pour la conception de systèmes électroniques fiables

par
Nicolas Gorse

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Thèse présentée à la Faculté des Études Supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Décembre 2005

Université de Montréal
Faculté des Études Supérieures

Cette thèse intitulée:

Méthodes formelles de haut niveau pour la conception de systèmes électroniques fiables

présentée par:

Nicolas Gorse

a été évaluée par un jury composé des personnes suivantes:

Julie Vachon – *Université de Montréal*
présidente – rapporteuse

El Mostapha Aboulhamid – *Université de Montréal*
directeur de recherche

Yvon Savaria – *École Polytechnique de Montréal*
co-directeur de recherche

Jean-Pierre David – *École Polytechnique de Montréal*
membre du jury

Dominique Borrione – *Université Joseph-Fourier, Grenoble, France*
examinatrice externe

Petko Valtchev – *Université de Montréal*
représentant du doyen de la FES

Thèse acceptée le :

RÉSUMÉ

Cette thèse porte sur l'emploi de méthodes formelles pour l'amélioration de la qualité dans le cycle de développement matériel-logiciel, visant ainsi la production de systèmes électroniques fiables.

Nous appuyant sur une revue de littérature détaillée au cours de laquelle nous identifions les lacunes du cycle de développement, nous nous concentrons sur deux axes de recherche conjoints, l'un portant sur l'ingénierie des exigences, l'autre sur la vérification des techniques de tolérance aux pannes.

Sur le plan de l'ingénierie des exigences, nous présentons une méthodologie complète allant de leur représentation abstraite à l'aide d'un formalisme de haut niveau jusqu'à la vérification formelle du système représenté par ces derniers. Nous appuyons notre théorie par un ensemble d'études de cas sur des exemples industriels représentatifs dont les résultats démontrent la pertinence de l'approche proposée.

En ce qui a trait à la tolérance aux pannes, nous présentons une méthodologie de vérification formelle des techniques dédiées à la détection des altérations du flux de contrôle par des transferts erronés (conséquences de *Single Upset Events*). Cette présentation est suivie par une explication de son implémentation ainsi que le détail des expérimentations effectuées. Les résultats obtenus démontrent, ici encore, la puissance et la pertinence de notre approche.

En guise de conclusion, nous rappelons la liste des contributions apportées et élaborons sur l'ensemble des axes de recherche futurs.

Mots clefs : Systèmes électroniques, méthodes formelles, qualité, fiabilité, ingénierie des exigences, tolérance aux pannes.

ABSTRACT

This thesis relates to the use of formal methods for the improvement of quality in hardware-software development cycle, thus aiming the production of reliable electronic systems.

Based on a detailed literature review during which we identify the gaps of the development cycle, we concentrate on two united research orientations, one bearing on requirements engineering , the other on the verification of fault-tolerance techniques.

Regarding requirements engineering, we present a complete methodology going from their abstract representation using a high level formalism to the formal verification of the system represented by the latter. We support our theory by a number of case studies on representative industrial examples whose results show the relevance of the approach we suggest.

On the fault-tolerance side, we present a methodology for the formal verification of the techniques dedicated to the detection of corruptions of the control flow caused by erroneous transfers (consequences of *Single Upset Events*). This presentation is followed by an explanation of its implementation as well as the detail of the experiments carried out. The results obtained show, again, the power and the relevance of our approach.

As conclusion, we summarize the list of contributions brought and suggest future research orientations.

Keywords: Electronic systems, formal methods, quality, reliability, requirements engineering, fault tolerance.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

L'aboutissement de cette thèse est le résultat de trois années de travail mais aussi de trois années de collaboration et de support de la part d'un grand nombre de personnes qui ont, de près ou de loin, toutes contribué à ma réussite.

J'aimerais en premier lieu remercier sincèrement mes directeurs de thèse, El Mostapha Aboulhamid et Yvon Savaria, pour leur support théorique, technique, financier[1] et moral, sans oublier leur disponibilité constante au cours de ce chapitre de ma vie passé à leur coté. Je tiens à leur témoigner l'expression de ma profonde et sincère amitié.

Je tiens également à remercier les membres du jury pour l'intérêt qu'ils ont porté à cette thèse. La pertinence de leurs remarques a permis d'apporter de nombreuses améliorations au manuscrit ainsi qu'à nos travaux de recherche. Je voudrais en particulier exprimer ma profonde gratitude à Dominique Borionne pour m'avoir fait l'honneur d'être l'examinatrice externe de ce jury.

Merci à André Baron de PMC-Sierra[2] et Bogdan Nicolescu de l'École Polytechnique de Montréal, pour leur collaboration respective aux deux axes de recherche développés dans cette thèse. Et, bien entendu, merci à Eduard Cerny qui a beaucoup influencé nos travaux de recherche à travers un grand nombre de discussions toutes aussi agréables qu'enrichissantes.

Merci, encore et toujours, à mes collègues du LASSO et autres laboratoires de recherche du DIRO qui ont supporté avec patience et humour les diverses crises existentielles que j'ai traversées au cours de mes trois années de thèse.

Last but not least, merci à ma conjointe et collaboratrice Pascale Bélanger qui a, sans toujours le savoir, beaucoup contribué à l'aboutissement de cette thèse, ne serait-ce qu'en provoquant régulièrement mon esprit de contradiction.

[1] Une partie de cette thèse a été subventionnée par le RESMIQ.

[2] Une autre partie de cette thèse a été subventionnée par PMC-Sierra via Micronet.

AVANT-PROPOS

Montréal, trois heures du matin.

Enflammé par le soleil de midi, le sable me brûle la plante des pieds. Guidé par mon instinct, je galope vers la délivrance que me promet le bleu de la mer et c'est en narguant palmiers et cocotiers qui me toisent d'un regard sévère que je plonge dans l'eau avec un sourire rassasié.

Ce n'est que bien plus tard, lorsque le soleil rejoint l'horizon et qu'une odeur de poisson grillé erre le long de la plage, que je quitte avec amertume ce rivage. Fatigué, la peau tannée par le soleil et le sel de l'océan indien, je regagne la ville par la route qui longe la corniche.

Brusquement, la chaussée qui défile devant moi s'efface, les rochers du front de mer disparaissent et le bruit des flots est rapidement remplacé par le silence de la nuit.

Un à un, les meubles et objets qui jonchent l'étendue restreinte de la pièce refont surface tandis que le monde fictif qui s'était mis en place sous mes yeux s'éclipse graduellement. Enfermé entre quatre murs, mon regard à présent vide de toute expression rationnelle se perd dans les cendres de mes songes.

Juchée à l'extrémité sud-ouest de la table qui me sert de bureau, une lampe imbibe de sa lumière tamisée les quelques papiers privilégiés qui n'ont pas encore leur place parmi le fatras des classeurs habitant l'étagère à laquelle je tourne le dos avec une feinte obstination.

Il fait une chaleur inhabituelle pour un début de septembre. La nuit est calme et les lumières de la ville scintillent sous le hululement régulier du vent. En éternel habitué de ces heures inavouables, j'ai laissé mon esprit se perdre en vagabondage nocturne dans les méandres de l'irréel.

N.

Asinus equum spectat.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

Le rythme actuel des innovations technologiques se traduit par une croissance exponentielle de la complexité des circuits intégrés. Confirmant les prédictions de la loi de Moore, le nombre de transistors que l'on peut intégrer sur une puce double tous les dix huit mois. Cet accroissement repousse les limites des méthodes de conceptions actuelles au delà de leur capacité, ce qui a pour résultat d'affecter fortement la production de systèmes sur puce (*Systems on Chip* – (SoC)) complexes, fiables et performants [1]. Il devient, entre autres, de plus en plus difficile de s'assurer qu'un SoC correspond aux spécifications données. Un ensemble de problèmes touche aujourd'hui le cycle de développement matériel-logiciel. Ces derniers sont regroupés en deux champs, respectivement relatifs au développement et à l'utilisation des systèmes.

Concernant le développement nous identifions trois sous-problèmes. Le premier a trait à la spécification des exigences. L'utilisation de documents d'exigences qui regroupent ces dernières sous forme textuelle accompagnée de tableaux, schémas et automates, est de plus en plus répandue. Ces exigences sont utilisées pour dériver les spécifications exécutables mais ne sont cependant vérifiées qu'au travers de la vérification des spécifications. Une vérification intervenant plus tôt, consistant à vérifier la cohérence et la complétude de ces exigences, permettrait d'éviter, du moins en grande partie, de dériver des spécifications erronées [2] [3].

Une seconde préoccupation est celle de la vérification des systèmes, que ce soit au sujet de leur spécification ou de leur fonctionnement. Leur taille actuelle pose problème, tant au niveau des méthodes de vérification formelles qui atteignent leurs limites avec les problèmes d'explosion combinatoire, qu'au niveau des méthodes basées sur la simulation dont l'utilisation nécessite des temps de calcul beaucoup

trop importants [4].

Un troisième problème des méthodes de développement concerne les aspects de productivité, lesquels sont bridés par les méthodes de conception actuelles. Sans une amélioration des méthodes afin de permettre une augmentation significative de la productivité, il sera impossible d'exploiter au maximum le nombre croissant de portes logiques des circuits sans dépasser les budgets de développement, contrôlés par des contraintes économiques.

Sur le plan de l'utilisation, les problèmes se rapportent principalement à la miniaturisation des circuits. L'aspect positif de cette miniaturisation est qu'elle entraîne une réduction significative de la consommation d'énergie, permettant de ce fait d'étendre l'autonomie des systèmes embarqués. L'aspect négatif est que les circuits miniatures sont considérablement plus sensibles aux agressions extérieures, résultant en des taux d'erreurs importants [5]. L'apport de techniques de tolérance aux pannes permet de maintenir ces systèmes à un niveau de fiabilité acceptable [6]. Cependant, les méthodologies de développement de ces techniques présentent des limites certaines vis-à-vis des phases de vérification.

1.2 Problématique

Jusqu'à récemment, les outils de conception assistée par ordinateur (*Computer Aided Development (CAD) tools*) supportaient uniquement les spécifications de type "logique de transfert entre registres" (*Register Transfer Logic (RTL)*). Le niveau d'abstraction de ces types de spécifications est très peu élevé et n'est absolument plus adéquat en ce qui concerne les systèmes actuels. En effet, le niveau de détail dont les outils doivent alors tenir compte dégrade les performances des tâches de conception et de vérification, créant ainsi un goulot d'étranglement dans le cycle de développement.

L'atteinte du niveau de productivité désiré passe nécessairement par une élévation du niveau d'abstraction utilisé dans le processus de développement [1]. En effet, la complexité des systèmes actuels et le niveau de détails avec lequel ils

sont conçus engendrent des problèmes d'explosion combinatoire irréductibles. Or, l'emploi d'un niveau d'abstraction plus élevé, et donc d'un niveau de détails moins important, permet une importante diminution de la complexité, réduisant alors en grande partie les problèmes d'explosion combinatoire. De plus, les phases de spécification et vérification à haut niveau d'abstraction permettent une amélioration de productivité certaine en ce qui a trait aux phases subséquentes. En effet, à un niveau d'abstraction élevé, la conception et la vérification sont bien plus rapides et permettent de détecter malgré tout une grande quantité de problèmes de conception.

Le cycle de développement matériel-logiciel est qualifié de cycle *descendant (top-down cycle)*. Élever le niveau d'abstraction dans un tel cycle consiste à lui superposer de nouvelles couches d'un niveau d'abstraction plus élevé que la phase initiale actuelle. Cela ne consiste en aucun cas à remplacer les phases déjà existantes. À titre d'exemple, le développement d'applications matériel-logiciel complexes tend de plus en plus à démarrer par l'expression des fonctionnalités sous une forme logicielle en utilisant un langage de haut niveau. Cette forme logicielle, appelée aussi "spécification", subit différentes évaluations (simulations, vérifications) et de raffinement jusqu'à l'atteinte du niveau de détail et de fiabilité voulu. Dès lors, cette dernière est utilisée comme point d'entrée aux phases subséquentes de conception à plus bas niveau tel que le RTL.

Sur le plan de l'utilisation, la miniaturisation constante des dispositifs entraîne une sensibilité croissante aux "erreurs transitoires" (*transient errors*). Un exemple illustratif est celui des SEU (*Single Event Upset*) [7], appelées aussi *bit-flips* ou erreurs douces. Ces erreurs sont la conséquence des processus de miniaturisation combinés avec les effets physiques extérieurs tels que les radiations ionisantes. Ces erreurs sont responsables de la modification du contenu des cellules mémoire (RAM, registres, etc.) entraînant ainsi des conséquences sur le comportement des systèmes embarqués, conséquences pouvant aller de simples erreurs de calcul à des *crashs* complets.

De plus, la sensibilité aux SEU entre en corrélation directe avec la quantité de

mémoire d'un dispositif. Ainsi, le cas des SEU dans les SoC est un problème grandissant étant donné l'accroissement des quantités de mémoire dans les systèmes embarqués. Les SoC deviennent de plus en plus sensibles aux radiations ionisantes produisant des changements de valeurs sur les bits. Ainsi, des taux non négligeables d'erreurs logicielles autrefois observées seulement dans les applications spatiales commencent à être observés au niveau des applications terrestres et aériennes. Nombre de méthodes dites "techniques de tolérance aux pannes" permettent de détecter et de corriger les erreurs causées par des radiations. Ces méthodes sont utilisées depuis des dizaines d'années, spécialement dans l'industrie spatiale. Ces méthodes fournissent des modules de détection-correction qui peuvent être implantés matériellement ou de manière logicielle.

Ces techniques sont d'une fiabilité relativement satisfaisante; elles détectent la quasi totalité des erreurs et de nombreux travaux de recherche contribuent à leur amélioration constante, spécialement en ce qui a trait à la réduction de la taille des modules à ajouter ainsi que la réduction des dégradations de performance dues au fonctionnement de ces modules. Malgré cela, un des problèmes majeurs reste l'évaluation des techniques de tolérance aux pannes. En effet, la quasi totalité des mécanismes de vérification consistent à exécuter des heures durant les programmes "durcis" par les méthodes en y injectant des erreurs, que ce soit par bombardement de radiations sur le processeur ou par des méthodes de simulation. Or, ces techniques de vérification sont très coûteuses et l'emploi d'une technique de vérification formelle à un niveau d'abstraction adéquat[1] peut réduire considérablement les temps de vérification, permettant ainsi d'améliorer grandement la productivité au niveau du développement de telles méthodes.

En résumé, les méthodologies de conception matériel et matériel-logiciel accusent un retard criant vis à vis de celles utilisées par la communauté logicielle. Le niveau de détail des spécifications reste très élevé, dégradant ainsi les performances des phases de vérification et entravant la productivité. Cette thèse

[1]par *adéquat* nous entendons que le niveau d'abstraction permet, malgré le fait qu'il soit plus élevé, de vérifier les mêmes propriétés qu'au niveau de détail employé par lors des simulations

démontre que l'apport de méthodes de vérification formelle de haut niveau permet d'améliorer considérablement le développement et la qualité des systèmes électroniques matériel-logiciel. Nous nous attardons sur chacun des axes énoncés précédemment, soit : a) la conception et, b) l'utilisation.

1.3 Axes cibles

Chacun des axes que nous ciblons est abordé du point de vue de la vérification. Pour chacun, nous proposons de greffer une méthodologie de vérification formelle à haut niveau d'abstraction au dessus du cycle de développement. Nous démontrons que les concepts théoriques que nous proposons, appuyés par des implémentations ainsi que des résultats concrets probants, apportent des gains de productivité et de qualité non négligeables.

1.3.1 Axe développement – Ingénierie des exigences

L'ingénierie des exigences concerne les aspects de capture, de spécification, et de vérification des exigences. Il s'agit d'un sujet traité par la communauté logicielle depuis déjà de nombreuses années, mais dont l'émergence dans la communauté matériel-logiciel est toutefois relativement récente. Nous inspirant des travaux de la communauté logicielle, nous proposons une méthodologie formelle d'ingénierie des exigences.

Notre méthodologie est destinée à être greffée au tout début du cycle de conception. Nous fournissons un mécanisme de représentation et documentation des fonctionnalités énoncées dans les exigences. Ce mécanisme se base sur la description d'une fonctionnalité suivant une structure formelle au sein de laquelle les éléments de la fonctionnalité sont exprimés de manière informelle en utilisant un anglais contrôlé. Les fonctionnalités ainsi modélisées sont traduites automatiquement en une représentation totalement formelle sur laquelle deux phases de vérification sont appliquées :

Cohérence – La première phase effectue une vérification de cohérence de manière

à s'assurer de l'absence de contradictions et d'éléments mal définis dans l'ensemble des fonctionnalités. Les résultats de cette vérification permettent aux concepteurs de raffiner les exigences en conséquence, résultant en une meilleure qualité des documents à partir desquels seront dérivées les diverses spécifications des phases subséquentes du cycle de développement.

Complétude – La seconde phase consiste en une extraction des fonctionnalités manquantes par un calcul de complétude. Cela consiste à s'assurer que toutes les fonctionnalités requises par rapport aux propriétés du système ont été définies. Les fonctionnalités manquantes identifiées peuvent être ajoutées par les concepteurs, permettant ainsi un raffinement des exigences.

L'ensemble des améliorations apportées par cette méthodologie englobe les divers problèmes liés à la conception que nous avons énoncés dans les premiers paragraphes de ce chapitre :

Exigences – Au sujet des exigences, nous fournissons une méthodologie complète de spécification et vérification des exigences.

Vérification – En ce qui a trait à la vérification, notre méthode améliore la qualité des exigences ayant un impact sur les spécifications qui en découlent, permettant ainsi une réduction des coûts de vérification.

Productivité – Enfin, sur le plan de la productivité, le traitement des exigences a des répercussions positives sur différents aspects du cycle de conception :

- Amélioration de la documentation et donc des échanges entre concepteurs ainsi que de la réutilisation,
- Amélioration de la qualité des spécifications dérivées des exigences,
- Réduction des coûts de vérification des spécifications,

Les résultats expérimentaux obtenus confirment que l'emploi de cette méthodologie permet d'effectuer en très peu de temps une vérification automatique des exigences dont les résultats sont fiables.

1.3.2 Techniques de tolérance aux pannes – Vérification formelle

La vérification des méthodes de tolérance aux pannes emploie des mécanismes de vérification par injection de pannes reposant sur des exécutions matérielles ou des simulations logicielles. Les quelques travaux portant sur la vérification formelle font état de méthodologies se heurtant à des problèmes d'explosion combinatoire en raison de la taille des systèmes à vérifier (voir chapitre 2, section 2.4). La solution que nous proposons est d'effectuer la vérification à un niveau d'abstraction beaucoup plus élevé avec la même précision et la même qualité des résultats.

Tout comme notre méthodologie d'ingénierie des exigences, celle que nous proposons pour la vérification des techniques de tolérance aux pannes est destinée à être greffée au tout début du cycle de développement. Nous fournissons un ensemble de programmes génériques de taille réduite dont chacun regroupe un ensemble de caractéristiques propres à une classe de programmes réels. En outre, nous fournissons un langage pseudo-assembleur utilisé pour implémenter les techniques de tolérance aux pannes dans les programmes génériques ainsi qu'un cadre de vérification formelle totalement automatisé. Notre approche se compose de deux étapes principales :

Implémentation – La phase d'implémentation consiste pour les concepteurs à implémenter la technique de tolérance aux pannes destinée à être évaluée au sein des programmes génériques donnés. Une telle tâche nécessite de maîtriser la technique à implémenter ainsi que le format des instructions du langage pseudo-assembleur utilisé.

Vérification – La seconde phase consiste à traduire les programmes pseudo-assembleur en des modèles formels et à vérifier par *model-checking* que ces derniers respectent un ensemble de propriétés formelles relatives à la fiabilité de la technique à évaluer.

La vérification terminée, les concepteurs peuvent consulter les résultats relatifs à la fiabilité de leur technique et extraire des scénarii graphiques illustrant les

comportements erronés. Cela leur permet de raffiner la technique, puis de la vérifier à nouveau formellement ou bien de se diriger vers des implémentations et des méthodes de tests plus traditionnelles.

Notre approche permet d'effectuer une vérification rapide et fiable d'une technique de tolérance aux pannes. Les phases d'implémentation et de vérification sont d'une extrême rapidité, améliorant de ce fait considérablement la productivité. En outre, les résultats expérimentaux obtenus à partir de l'implémentation et de la vérification formelle de trois techniques de tolérance aux pannes nous ont permis d'obtenir en quelques minutes des résultats qui confirment, avec un niveau de précision semblable, ceux obtenus en plusieurs heures de simulation.

1.4 Contributions

Le travail de recherche que nous avons accompli au cours de nos trois années de thèse a comme point de départ nos travaux antérieurs portant sur l'utilisation de méthodes formelles de haut niveau en téléphonie [8] [9]. À partir de ce point de départ, nous avons exploré d'autres formes de vérification de haut niveau pouvant être employées pour l'amélioration du cycle de développement matériel-logiciel contemporain. Cette thèse apporte à la communauté matériel-logiciel un certain nombre de contributions qui permettent une amélioration certaine du cycle de développement.

Dans un premier temps, sur le plan de l'ingénierie des exigences, nos travaux ont donné lieu à deux publications dans des conférences internationales [10] [11] ainsi qu'à un article récemment soumis à une revue [12]. La liste des contributions est la suivante :

- Un cadre de formalisation des exigences,
- Un ensemble de règles de vérification de cohérence,
- Un mécanisme d'extraction automatique des exigences manquantes,

- L'implémentation de ces concepts au sein d'un outil, fournissant ainsi une méthodologie d'ingénierie des exigences automatique complète permettant :
  - Une amélioration de l'expression et de la documentation des exigences,
  - Une notation commune permettant un meilleur partage entre concepteurs ainsi qu'une meilleure réutilisation,
  - Une amélioration de la qualité des exigences, permettant une réduction des coûts de vérification des phases subséquentes et donc une augmentation de la productivité,
- Finalement, un ensemble d'expérimentations démontrant la pertinence de nos travaux.

Deuxièmement, l'ensemble de nos travaux sur la vérification des techniques de tolérance aux pannes a donné lieu à une publication dans une conférence internationale [13] suivie d'une publication dans une revue [14]. Un troisième article de revue est en préparation [15]. L'ensemble des contributions que nous apportons dans le cadre de la conception des techniques de tolérance aux pannes est le suivant :

- Un ensemble de programmes génériques de taille réduite caractérisant des classes de programmes,
- Un langage pseudo-assembleur permettant l'implémentation des techniques à vérifier au sein des programmes,
- Des mécanismes de traduction permettant de générer un modèle formel à partir d'un programme pseudo-assembleur,
- L'implémentation de ces concepts sous la forme d'un outil, fournissant ainsi une méthodologie complète et automatique de vérification des techniques de tolérance aux pannes permettant:
  - Un prototypage rapide,

  - Une vérification de fiabilité formelle complète,
  - L'extraction de scénarii contre-exemples illustrant les défauts de la technique en évaluation,
  - Un gain de productivité considérable dû à la rapidité d'implémentation et, spécialement, de vérification des techniques,

- Finalement, un ensemble d'expérimentations démontrant la pertinence de nos travaux.

L'ensemble de ces contributions est repris de manière beaucoup plus détaillée à la fin des différents chapitres et plus particulièrement dans le dernier chapitre (conclusion, chapitre 7) de la thèse.

1.5 Organisation de la thèse

Cette thèse se compose d'une introduction et d'une conclusion entre lesquelles s'insère une revue de littérature talonnée par une alternance de chapitres comme suit :

Le chapitre 2 présente un état de l'art sur les systèmes matériel-logiciel. Nous présentons en premier lieu le cycle de développement ainsi que ses lacunes, puis nous introduisons les solutions que nous proposons et présentons l'état de l'art relatif à chacune de ces solutions.

Les chapitres 3 et 4 portent sur l'ingénierie des exigences. Le chapitre 3 présente les aspects théoriques de notre méthodologie d'ingénierie des exigences. Il est suivi par le chapitre 4 qui présente et discute l'implémentation des concepts théoriques énoncés au chapitre 3 ainsi que les études de cas et résultats expérimentaux.

Les chapitres 5 et 6 portent sur la vérification des techniques de tolérance aux pannes. Le chapitre 5 présente les aspects théoriques de notre méthodologie de vérification formelle des techniques de tolérance aux pannes. Le chapitre 6 présente et discute l'implémentation des concepts théoriques énoncés au chapitre 5 ainsi que les études de cas et résultats expérimentaux.

Enfin, le chapitre 7 résume les travaux effectués au cours de la thèse, rappelle les contributions et propose de nouveaux axes de recherche dans la lignée de ce qui a été présenté.

CHAPITRE 2

ÉTAT DE L'ART

Le développement de systèmes électroniques matériel-logiciel robustes repose sur un cycle bien établi au cours duquel les concepteurs ont à leur disposition de nombreux environnements de développement [16] [17] [18] [19], et de vérification [4] [20] qui ont été intensément élaborées par l'industrie au fil des ans. Malgré l'existence de tels environnements, la conception matériel-logiciel est actuellement à la frontière d'une crise importante [1] : la taille et la complexité des systèmes affecte profondément les phases de vérification ; les techniques formelles se heurtent à la barrière des explosions combinatoires tandis que les procédures de simulation nécessitent des temps de calcul devenant trop grands [4].

Ce problème affecte particulièrement les phases de vérification du cycle de développement, qu'il s'agisse de la vérification des modèles ou de la vérification des techniques de tolérance aux pannes. La communauté scientifique matériel-logiciel s'accorde sur le fait que la solution réside en partie dans l'élévation du niveau d'abstraction des modèles, ainsi que l'utilisation de nouvelles méthodes de modélisation et vérification agissant à des niveaux d'abstraction plus élevés [1].

Nous pensons que la solution aux problèmes de vérification réside effectivement dans une élévation du niveau d'abstraction, mais que cette élévation doit s'accompagner de méthodes de vérification formelles adaptées au niveau d'abstraction utilisé. Ainsi, nous proposons dans cette thèse une solution au problème de vérification des modèles ainsi qu'au problème de vérification des méthodes de tolérance aux pannes. Ce chapitre présente la problématique, l'ébauche des solutions que nous proposons, ainsi que l'état des travaux de recherche actuels. Les chapitres suivants présentent en détail les solutions que nous avons élaborées.

Ce chapitre s'articule autour de quatre sections. Dans une première section, nous présentons le cycle de développement matériel-logiciel. Elle est suivie d'une seconde section dans laquelle nous identifions les manques de ce cycle et présentons

les solutions que nous proposons pour pallier ces manques ; a) sur le plan de l'ingénierie des exigences, ainsi que b) sur le plan de la vérification des méthodes de tolérance aux pannes. Les deux dernières sections portent sur l'état des travaux de la communauté scientifique concernant : a) les méthodes d'ingénierie des exigences, et b) les méthodes de vérification des techniques de tolérance aux pannes.

2.1 Cycle de développement matériel-logiciel

Cette section présente le cycle de développement des systèmes matériel-logiciel robustes ainsi que les environnements de développement et techniques de vérification les plus couramment utilisés par la communauté actuelle.

2.1.1 Aperçu

La conception matériel-logiciel cible la production d'applications logicielles et de systèmes qui sont mis en place sur des plateformes matérielles précises en fonction de besoins spécifiques. Le cycle de développement est une approche descendante qui consiste, tel qu'illustré par la figure 2.1, en un amalgame des méthodes de conception matérielles et logicielles.

Le cycle commence avec les idées préliminaires relatives au système à développer, à partir desquelles sont écrits les documents énonçant la description du système et les exigences requises par rapport à ses fonctionnalités. De tels documents, aussi appelés "documents informels" consistent en la description, en langue naturelle, du système et de ses fonctionnalités. Ces descriptions sont la plupart du temps accompagnées de figures, tableaux, automates, algorithmes représentatifs, et parfois même, de détails particuliers concernant l'implémentation. Ce sont ces documents qui servent de point de départ à la phase de modélisation transactionnelle et fonctionnelle [21] durant laquelle est dérivé un premier modèle haut-niveau du système.

Les divers composants du modèle sont ensuite affectés à une implémentation matérielle ou logicielle en fonction des contraintes de modularité ou d'ordre temporel. Le reste du cycle consiste en deux branches parallèles au cours desquelles les

implémentations matérielles et logicielles sont raffinées et vérifiées par co-simulation ainsi que par simulation ou par des techniques formelles [22].

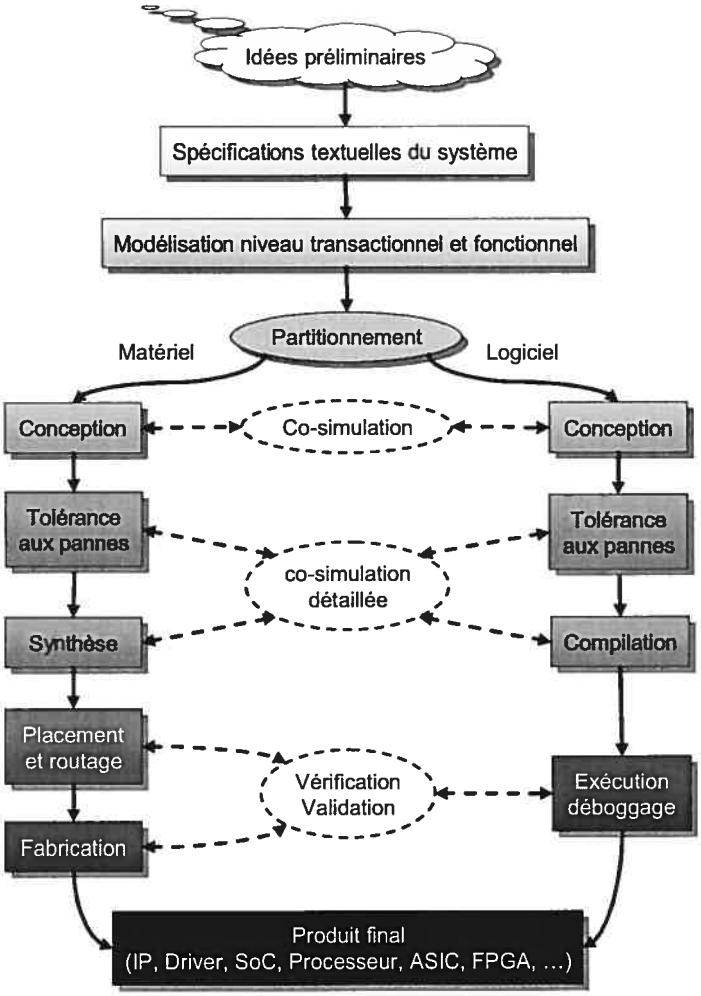

Figure 2.1: Cycle de développement matériel-logiciel

Les méthodes de tolérance aux pannes, pouvant être implantées tant au niveau matériel que logiciel, sont destinées à rendre le système robuste aux pannes provoquées par l'environnement (radiations, magnétisme, défauts électriques). Ces méthodes font partie intégrante de l'implémentation et leur fonctionnement est parfois testé par l'utilisation de techniques formelles, mais ce sont les méthodes d'injection de pannes qui sont majoritairement utilisées.

Dans une classe de techniques de tolérance logicielle populaire, ces techniques

sont ajoutées au niveau assembleur. Elles consistent en un "durcissement" de l'application par ajout d'instructions de vérification de signature aux différents blocs du code, permettant ainsi de détecter les flux de contrôle erronés. Lors d'une telle détection, suivant les méthodes, l'exécution du programme est alors arrêtée ou dirigée vers une routine de correction qui lui permettra de reprendre le flux d'exécution escompté.

Le produit final peut consister en un composant matériel ou bien en une plate-forme sur laquelle les parties logicielles et matérielles sont exécutées conjointement de manière coopérative. Il peut s'agir [22] de : blocs de propriété intellectuelle, contrôleurs de périphériques, systèmes sur puce, processeurs, circuits dédiés à des applications spécifiques, systèmes reconfigurables, ou toute autre combinaison adéquate.

2.1.2 Environnements de développement

De nombreux environnements et outils de développement sont accessibles aux concepteurs au cours du cycle. Les processus de conception et raffinement sont en partie, voire totalement automatisés, permettant ainsi un gain de productivité considérable. Les langages matériel et matériel-logiciel disponibles permettent de décrire à un niveau d'abstraction somme toute assez élevé, des composants dont les détails s'expriment parfois à très bas niveau. Cette section donne un aperçu des langages les plus couramment utilisés. Ces derniers sont synthétisables ou peuvent être compilés vers des langages qui le sont, exception faite des techniques de description formelles. De plus, tous fournissent des possibilités de vérification, formelle ou par simulation. Ces dernières sont présentées dans la section qui suit.

2.1.2.1 VHDL et Verilog

VHDL [19] et Verilog [18] sont deux langages de description fortement utilisés par les concepteurs autant dans le domaine industriel qu'académique.

VHDL (*VHSIC Hardware Description Language*) est un langage de descrip-

tion qui inclut une méthodologie complète visant l'utilisation d'un haut niveau d'abstraction, la portabilité du code, et la conception assistée par ordinateur. Sur le plan de la spécification, VHDL implémente de nombreux principes tels que : la conception modulaire hiérarchique, ainsi que la séparation des interfaces, l'utilisation unique ou partagée d'interfaces par différents modules. De plus, VHDL permet l'utilisation de descriptions paramétrables comportementales, structurelles, et mixtes.

VHDL est un langage de description standard pour les systèmes électroniques et est utilisé comme point de départ de nombreux outils de simulation, synthèse, et placement routage. Il fournit une notation unifiée pour la description de systèmes matériel-logiciel à travers de nombreux niveaux d'abstraction, étant ainsi utilisable tout au long du processus de développement. Sur le plan de la vérification, de nombreux outils [23] [24] [25] [26] permettent de simuler les modèles VHDL à tout niveau d'abstraction. La simulation peut ainsi être effectuée au niveau fonctionnel comme au niveau des portes logiques.

Les concepteurs sont généralement plus familiers avec le C qu'avec ADA. Pour cette raison, ils préfèrent souvent Verilog, dont la syntaxe est similaire au C, à VHDL dont la syntaxe est similaire à celle d'ADA. Verilog permet la description des modèles à un haut niveau d'abstraction tel qu'architectural ou comportemental, comme à un très bas niveau tel que les portes logiques. Tout comme VHDL, il implémente diverses fonctionnalités de spécification, simulation, vérification et synthèse. Il manque néanmoins à Verilog un certain nombre de primitives présentes dans VHDL. Ces dernières ont été ajoutées à SystemVerilog.

2.1.2.2 SystemC

SystemC [17] propose une nouvelle tendance dans les langages de description matérielle. Il consiste en une bibliothèque de modélisation matérielle développée en C++. Cette bibliothèque fournit des classes dédiées à la modélisation matérielle ainsi qu'un noyau de simulation. Elle permet aux concepteurs de créer et simuler des modèles matériels-logiciels du niveau système au niveau RTL (*Register Transfer*

Level).

Le but de SystemC est de fournir une alternative de conception aux langages de description matérielle existant en intégrant la programmation procédurale ordinaire (déjà présente dans Verilog et VHDL) et les paradigmes orientés objets de C++, à savoir : les classes, l'héritage, la sur-charge des fonctions, les modules (déjà présents dans VHDL et Verilog).

Les modèles peuvent être créés en combinant la bibliothèque SystemC avec les fonctionnalités C++ ANSI standard. D'autres bibliothèques dédiées à certains modèles spécifiques peuvent être utilisées conjointement à SystemC, permettant ainsi aux concepteurs de bénéficier de l'ensemble.

Les avantages des fonctionnalités multi-paradigmes que fournit SystemC ont été démontrés dans de nombreux travaux [27] [28] [29] [30]. Néanmoins, une telle liberté de modélisation s'accompagne de certaines limites. SystemC n'intègre pas toutes les fonctionnalités présentes en VHDL et Verilog et n'est pas, tout comme VHDL et Verilog, totalement synthétisable . Bien que ce langage ne soit pas encore aussi utilisé que VHDL et Verilog, il présente un potentiel non négligeable, spécialement en ce qui concerne la modélisation à haut-niveau.

2.1.2.3 Confluence

Confluence [31] découle d'une autre tendance actuelle dans les langages de description matérielle. Il s'agit d'un langage de programmation fonctionnel déclaratif qui a pour but de fournir une solution meilleure et plus simple pour la génération de code RTL. Confluence est désigné comme étant un "langage générateur" qui peut être compilé sous forme de structures RTL, permettant ainsi la description de systèmes complexes en un nombre réduit de lignes de code.

Confluence supporte le parallélisme, les hiérarchies, et les flux de données. Il fournit aussi un certain nombre de structures telles que la récurrence, des types de données haut-niveau, et des mécanismes de transparence référentielle[1]. La plupart

[1] La transparence référentielle est la propriété d'une expression dont la valeur est indépendante de son contexte d'évaluation.

des structures fournies par Confluence ne sont pas disponibles en VHDL ou Verilog.

Confluence semble être un langage très prometteur. Ses créateurs annoncent qu'en comparaison à Verilog, VHDL, et C, les systèmes conçus avec Confluence présentent une réduction du nombre de lignes de code allant de 50% à 80%, permettant une meilleure gestion et réutilisation du code. De plus, Confluence repose sur un compilateur dit *correct par construction* réduisant les erreurs de conception et permettant ainsi de réduire l'ampleur du processus de vérification du modèle.

2.1.2.4 SystemVerilog

Tout comme SystemC et Confluence, SystemVerilog [16] découle d'une autre tendance récente sur le marché des langages de description matérielle. Il s'agit d'une extension majeure du langage Verilog développée dans le but d'accroître la productivité et d'améliorer la vérification de systèmes complexes. SystemVerilog cible particulièrement les implémentation sur puce et les flux de vérification.

SystemVerilog supporte la modélisation et la vérification de systèmes à un niveau transactionnel et intègre le paradigme orienté-objet. De plus, il permet d'intégrer du code C, C++, et SystemC aux modèles écrits en SystemVerilog par l'appel de fonctions spécifiques. Permettant de co-simuler efficacement des blocs SystemC, ceci facilite la réutilisation de portions de code déjà écrites en SystemC. Sur le plan de la vérification, SystemVerilog intègre de nouveaux mécanismes d'expression de ce qui doit être vérifié basés sur les assertions, améliorant grandement la vérification des modèles.

En résumé, SystemVerilog est un langage très complet et puissant permettant le développement et la vérification d'applications complexes.

2.1.2.5 ESys.Net

Esys.Net [32] [33] [34] est un nouveau cadre de modélisation, simulation, et vérification de haut niveau. Il est développé au sein du laboratoire LASSO de l'Université de Montréal. Il s'agit d'un environnement reposant sur C# et l'environnement

Microsoft .Net. ESys.Net permet l'utilisation simultanée des concepts intrinsèques à C# et .Net. En somme, ce cadre de développement puise sa force dans les concepts suivants :

- Il permet le développement d'applications multi-langage,
- Il allège la gestion de la mémoire et offre la possibilité d'intégration de composants logiciels avec des systèmes d'exploitation,
- Il fournit des mécanismes d'annotation et de transformation des modèles en un format intermédiaire.

En prime, ESys.Net offre une couche de vérification formelle permettant aux concepteurs de représenter formellement les propriétés devant être vérifiées durant la simulation [34].

2.1.2.6 Techniques de description formelles

La modélisation à haut-niveau (transactionnel/système) peut être effectuée en utilisant des langages de représentation formels [35] [36] [37], déjà utilisés de manière intensive par la communauté du génie logiciel et des télécommunications. L'avantage principal de tels langages est que leur niveau d'abstraction permet un prototypage rapide. La capture des systèmes à un niveau transactionnel est donc possible. Néanmoins, la plupart des langages ne fournissent pas de représentation discrète du temps, ne permettant donc pas de spécifier des systèmes utilisant une horloge, ce qui est nécessaire pour la plupart des composants matériels.

À titre d'exemple, LOTOS [38] [39] (*Language of Temporal Ordering Specifications*) est une technique de description formelle [37] qui intègre les concepts d'algèbre des processus de CCS [40] et CSP [41]. LOTOS représente le comportement d'un système par un ensemble d'expressions dites "comportementales" qui décrivent un ordonnancement d'actions. Les actions représentent les comportements de base du système modélisé. Des expressions complexes peuvent

être obtenues par la combinaison d'actions et d'expressions comportementales par l'utilisation d'opérateurs spécifiques.

Il est possible de représenter un système de manière claire et précise en utilisant LOTOS. Malheureusement ce langage n'est pas supporté par un grand ensemble d'outils de simulation et de vérification. De plus, l'intégration d'une représentation discrète du temps est très récente [42] et n'est supportée par aucun outil pour le moment. Excepté quelques exemples [43] [44], LOTOS n'a pas été utilisé de manière intensive dans la spécification et la vérification de systèmes matériel-logiciel.

En résumé, les techniques de description formelles, telles qu'employées la plupart du temps, semblent se situer à niveau trop élevé pour être utilisées dans la spécification et la vérification matériel-logiciel. D'autre part, elles nécessitent des temps de calcul trop élevés lorsqu'il s'agit de simuler et/ou vérifier des modèles complexes intégrant des horloges et il semble que des langages comme ceux présentés plus tôt dans cette section soient bien plus appropriés dans la mesure où ce sont des langages qui ont été conçus dans le but d'être utilisés pour la modélisation matérielle.

2.1.3 Vérification des systèmes matériel-logiciel

Le but de la vérification d'un modèle est de s'assurer que ce dernier respecte les exigences, en termes de fonctionnalités et de comportements[2], décrites par les clients. Un grand ensemble de méthodes est dédié à la vérification des systèmes matériels et logiciels. Ces dernières sont appuyées par des outils fiables qui exploitent des techniques allant de la simulation aux techniques purement formelles. Trois tendances majeures co-existent : vérification par simulation, vérification par assertions, et vérification formelle. Cette section présente une revue de ces trois tendances.

[2]Par comportement, nous entendons une (ou plusieurs) action(s) effectuée(s) par un module du système en fonction de son état et des stimuli reçus.

2.1.3.1 Vérification par simulation et tests

La vérification par simulation est effectuée durant la simulation par l'utilisation d'un ensemble de stimuli regroupés dans ce qui est appelé un banc de test (*test-bench*) [45]. Un banc de test est une unité à part qui communique avec le modèle par des ports d'entrée et de sortie. Le banc de test envoie des stimuli au modèle et collecte les réponses, vérifiant alors si ces dernières sont appropriées ou non suivant les stimuli envoyés et les fonctionnalités testées. Un grand nombre de bancs de test différents peuvent être dédiés à la vérification d'un modèle, chacun vérifiant un ensemble de fonctionnalités spécifiques.

Il existe de nombreuses techniques de vérification par simulation. À part les techniques de description formelles, les environnements de développement présentés dans la section 2.1.2 intègrent des mécanismes de vérification par banc de test [46] [47]. D'autres proposent des approches légèrement différentes [48] [49] de manière à permettre de détecter les erreurs plus tôt dans le cycle de développement ou bien pour permettre la vérification de modèles transactionnels.

La simulation permet d'explorer le modèle à vérifier à différentes profondeurs. Les tests dits "boite-noire" (*blackbox testing*) ciblent la vérification des modèles en se basant seulement sur les entrées et sorties externes, faisant ainsi abstraction complète de ce qui se passe à l'intérieur du modèle et s'attardant uniquement aux résultats des fonctionnalités testées. Les tests dit "boite-blanche" (*whitebox testing*) ciblent la vérification des modèles en tenant compte des constructions internes de ces derniers, permettant ainsi d'effectuer des vérifications plus profondes en termes de détails et fonctionnalités.

La vérification par simulation affiche deux principales limites. Premièrement, la vérification cible les modèles complexes. S'assurer que l'ensemble des fonctionnalités est correctement implémenté implique de tester chaque couple stimulus-réponse possible. Le nombre de scénarii ainsi requis peut être immense. Même si ces scénarii peuvent être dérivés automatiquement, ils doivent être testés, ce qui nécessite habituellement des temps de calcul inacceptables, menant à des vérifica-

tions partielles. Deuxièmement, la vérification partielle peut être très sournoise. En effet, il est difficile de déterminer le taux de couverture des scénarii choisis ; il est difficile de rassembler l'information résultant de l'exécution de ces derniers et de déterminer si ceux-ci sont pertinents. C'est principalement pour cette raison que la vérification par assertions, a été développée.

2.1.3.2 Vérification par simulation et assertions

La vérification par assertions [50] [51] est une tendance relativement récente dans la communauté matérielle. Une assertion, parfois aussi appelée moniteur, est une description précise et concise d'un ou plusieurs comportements spécifiques supposés être présents dans le modèle en développement. Les assertions sont ainsi utilisées pour capturer et vérifier les fonctionnalités des modèles. Le niveau de description peut varier en fonction du niveau de vérification ciblé. Ainsi, une assertion peut spécifier un comportement en termes de signaux, transactions, ou à plus haut niveau encore.

Les assertions sont habituellement spécifiées durant la création du modèle et sont vérifiées durant les simulations, les stimuli de ces dernières pouvant reposer sur des bancs de test ou être aléatoires. Dans le premier cas, la couverture du modèle dépend uniquement des assertions. Dans le deuxième, assertions et banc de test fournissent une couverture complémentaire et/ou redondante. Les assertions améliorent la vérification en fournissant une meilleure observabilité du modèle et en permettant d'exprimer simplement des propriétés qui seraient complexes à exprimer dans un banc de test. Les assertions peuvent aussi faire partie des phases de modélisation où elles peuvent être utilisées pour spécifier des fonctionnalités pour lesquelles du code RTL peut ensuite être automatiquement dérivé à partir des assertions en question. Finalement, les assertions peuvent être utilisées à des fins de vérification formelle, améliorant ainsi les capacités de vérification des outils les intégrant.

Le langage OVA (*OpenVera Assertions*) [52] [53], un sous-ensemble du langage OpenVera [54], ainsi que les langages E [55] et PSL [56] sont des langages de de-

scription d'assertions. Ce sont des langages déclaratifs dont la sémantique est basée sur la théorie des expressions régulières et la logique linéaire temporelle [20]. La combinaison de ces deux théories résulte en un langage puissant pour l'expression des activités matériel-logiciel communes comme les invariants et les opérations des automates. De plus, les expressions LTL peuvent être traduites sous forme d'automates [57] [58] [59], qui peuvent alors être utilisés avec des outils de vérification formelle dans le but de vérifier formellement les propriétés exprimées sur un système de transitions.

Bien que les assertions permettent de décrire les propriétés et fonctionnalités d'une manière plus précise et plus efficace, leur vérification reste dépendante d'un banc de test ou d'une simulation aléatoire. Ainsi, le problème du temps de simulation nécessaire pour la vérification d'un ensemble d'assertions reste similaire à celui présent dans le cas de la vérification par simulation. Une alternative à ce problème peut être l'utilisation de méthodes de vérification formelle, lesquelles ne dépendent pas de la simulation.

2.1.3.3 Vérification formelle

Un certain nombre de techniques de vérification formelle applicables aux modèles matériel-logiciel existent [60] [4] [61] [62]. Ces techniques peuvent être utilisées à différents niveaux d'abstraction, néanmoins, leur utilisation dans le cadre des systèmes matériel-logiciel concerne la vérification de systèmes de "haut-niveau", c'est-à-dire dont les détails d'implémentation sont représentés à un niveau d'abstraction encore plus haut que celui des langages de description matérielle. Un tel niveau d'abstraction cible la représentation des fonctionnalités du système en faisant abstraction de sa structure et des détails d'implémentation comme les horloges. Nous distinguons différentes catégories d'approches respectivement basées sur : les fonctions booléennes, les automates, les logiques temporelles, et les logiques d'ordre supérieur (*high-order logics*).

Les techniques reposant sur les fonctions booléennes consistent à représenter le système et ses fonctionnalités en utilisant des tables de fonctions, des dia-

grammes de décision binaire, ainsi que la logique propositionnelle. Ces techniques permettent de représenter les caractéristiques fonctionnelles et relationnelles des modèles. La vérification repose sur une détection des inter-blocages ainsi que sur des vérifications d'équivalence et des preuves logiques sur les propositions logiques.

Les approches basées sur les automates [4] consistent à modéliser le système sous forme d'automates. Ces automates sont en général utilisés dans la représentation des comportements de circuits permettant un nombre fini ou infini de séquences d'entrées et de sorties. Quoi qu'il en soit, la théorie sur laquelle reposent ces approches permet seulement d'effectuer des vérifications d'équivalence entre automates. Ces approches peuvent être utiles quand il s'agit de comparer une implémentation à sa spécification (bisimulation), mais elles sont grandement limitées pour ce qui est de la vérification fonctionnelle par simulation, particulièrement en raison de la difficulté d'exprimer un ensemble de scénarios qui couvre l'ensemble des fonctionnalités.

Les approches reposant sur les logiques temporelles [63] [60] [4] [64] sont les plus utilisées. Les systèmes sont modélisés sous forme d'automates. Les fonctionnalités et comportements à vérifier sont décrits sous forme de propriétés temporelles à l'aide de langages formels tels que LTL [20] (*Linear Temporal Logic*) et CTL [4] (*Computation Tree Logic*). La vérification est effectuée à l'aide de *model-checkers* [65] [63] [66]. Ces derniers s'assurent de manière formelle, c'est-à-dire par des calculs mathématiques, que le modèle vérifié satisfait l'ensemble des propriétés données. La vérification formelle couvre tous les cas d'entrées possibles et permet donc, contrairement aux vérifications par simulation et assertion de preuves, de conclure que le modèle satisfait toutes les propriétés dans tous les cas. Néanmoins, l'espace mémoire requis pour ces calculs croît généralement de manière exponentielle avec la taille des modèles vérifiés [20].

Les logiques d'ordre supérieur [4] [67] ne sont pas très répandues dans le domaine de la vérification matériel-logiciel. Elles proposent des solutions basées sur la logique du premier ordre ainsi que sur des logiques d'ordre supérieur. La modélisation cible uniquement les comportements du modèle à vérifier. La vérifi-

cation, quant à elle, est effectuée avec des outils d'aide aux preuves (*theorem-provers*) [68] [69]. Leur faible taux d'utilisation est sans doute explicable par le fait que leur utilisation nécessite des connaissances mathématiques avancées.

2.1.4 Vérification des méthodes de tolérance aux pannes

La vérification des méthodes de tolérance aux pannes est majoritairement effectuée par des techniques d'injection de pannes. Tel qu'expliqué dans l'article [70], ces techniques se divisent en deux catégories distinctes relatives à leurs buts respectifs :

La première catégorie concerne les techniques ciblant l'élimination des pannes. L'objectif de ces techniques est de tester de manière systématique les propriétés de tolérance aux pannes d'un système. Des expérimentations d'injection de pannes sont utilisées pour déterminer si le système en question est capable de détecter et de gérer une panne (provenant de l'ensemble de pannes anticipées) de manière adéquate. Habituellement, un nombre relativement petit de pannes choisies avec minutie est injecté. Une des exigences principales qui s'applique aux outils d'expérimentation est la contrôlabilité, c'est-à-dire la possibilité d'injecter des pannes d'une manière bien contrôlée et reproductible à tout endroit du système évalué. D'autre part, toute déficience détectée lors de l'expérimentation doit s'accompagner d'un ensemble de données suffisant pour permettre d'effectuer un diagnostic et ainsi identifier les actions correctives à prendre. Cette condition requiert une bonne observabilité du système, c'est à dire la capacité d'observer toutes les cellules de stockage et les signaux pertinents.

La seconde catégorie englobe les techniques qui consistent à anticiper les conséquences des pannes sur le comportement d'un système. Ces techniques injectent des pannes pour accroître le nombre de pannes et accélérer leur apparition dans les expérimentations. Ces expérimentations sont utilisées dans le but d'établir des mesures statistiques quantitatives relatives à l'efficacité de détection (la couverture) des mécanismes et algorithmes de tolérance aux pannes évalués. Toute déviation des conditions d'expérimentation par rapport aux conditions initiales risque de mener à une estimation biaisée des propriétés de tolérance aux pannes. De ce

fait, il est essentiel de pouvoir créer un environnement expérimental représentatif de la situation d'utilisation du système anticipée (plateforme cible, charge de travail, ensemble de pannes). En particulier, les techniques utilisées pour l'injection de pannes ne doivent en aucun cas modifier le modèle évalué. De telles techniques sont qualifiées de non-intrusives. L'ensemble d'outils d'expérimentation doit être suffisamment flexible et puissant pour permettre l'émulation de la variété potentiellement grande de pannes anticipées. Un grand nombre de pannes doit être injecté afin d'atteindre un haut niveau de confiance dans les mesures statistiques établies. Cette exigence nécessite un ensemble d'outils automatisés capables d'injecter des séquences de pannes rapidement.

Indépendamment de leur but, les techniques d'injection de pannes se divisent en catégories distinctes en fonction de leur implémentation [70]. Nous distinguons ainsi : les techniques d'injection de pannes matérielles, logicielles, et par simulation. Une quatrième catégorie concerne les techniques dites hybrides, c'est à dire combinant deux ou trois des techniques susmentionnées.

2.1.4.1 Techniques matérielles

Les techniques dites matérielles [71] [72] exposent un prototype matériel du système testé à des pannes qui peuvent être artificiellement causées par un dispositif extérieur ou bien par des dispositifs matériels implémentés dans la plateforme sur laquelle le système en cours d'évaluation est exécuté.

La génération externe de pannes repose sur des perturbations telles qu'une exposition aux radiations, à un faisceau laser, ou à des irrégularités de l'alimentation électrique. Alternativement, des périphériques extérieurs peuvent être utilisés pour forcer la modification d'un signal du circuit.

Au niveau des dispositifs internes à la plateforme, une technique consiste à insérer des portes logiques additionnelles de manière à manipuler les valeurs des signaux. Néanmoins cette technique est intrusive et risque de ce fait de biaiser les résultats des expérimentations. Une autre approche, non intrusive, vise à geler l'exécution du système, à modifier les valeurs de registres et/ou de bascules, puis à

poursuivre l'exécution avec les valeurs modifiées. Cette dernière technique nécessite d'avoir un contrôle total sur la plateforme d'exécution.

D'un point de vue général, il est reconnu que l'utilisation d'un prototype physique pour l'évaluation d'une technique permet de travailler sur une représentation réelle (et exacte) du système testé. De ce fait, les résultats obtenus sont d'une très grande fiabilité, et donc très convaincants. Les pannes injectées, particulièrement celles dont la génération repose sur des perturbations provenant de dispositifs externes, sont reconnues comme étant des pannes dites "réelles" dans le sens où ces dernières pourraient tout à fait se produire dans des cas d'application réels. De plus, la majorité des méthodes étant non intrusive, les expérimentations peuvent être conduites en temps réel, facilitant ainsi l'obtention de résultats dans un laps de temps relativement raisonnable. L'utilisation de ces techniques est néanmoins peu répandue en raison des coûts élevés de fabrication des circuits ainsi que des dispositifs perturbateurs.

2.1.4.2 Techniques logicielles

Les techniques dites logicielles [73] [74] reposent sur des injections de pannes par des modules logiciels. Néanmoins, elles ne reposent pas sur un modèle logiciel du système en cours d'évaluation, mais bel et bien sur une plateforme matérielle au sein de laquelle sont implémentés le système en cours d'évaluation ainsi que des modules d'injection de pannes. Deux approches co-existent.

D'une part, les approches dites "hors-ligne" consistent en la modification des données (registres, variables en mémoire) avant l'exécution du système. Ce dernier opère ainsi sur un espace de variables et registres erronés comme si leurs valeurs avaient été modifiées par des perturbations extérieures. Généralement peu coûteux en termes d'espace requis sur la plateforme, ce type d'approche nécessite néanmoins le lancement d'une nouvelle exécution pour chaque groupe de pannes injecté, entraînant de ce fait des temps d'expérimentations élevés.

D'autre part, les approches dites "en-ligne" consistent en la modification des données durant l'exécution du système sur la plateforme. Ceci est pris en charge

par des séquences d'instructions spécifiques qui sont ajoutées au code du système en évaluation. Ces instructions peuvent être insérées directement (avant ou lors de la phase de compilation) ou durant l'exécution, par des mécanismes d'interruption. L'exécution d'une séquence a pour résultat la modification des valeurs de registres et/ou d'adresses mémoire, comme si leurs valeurs avaient été modifiées par des perturbations extérieures.

Contrairement aux techniques matérielles, les techniques logicielles ne requièrent aucun dispositif d'injection de pannes particulier, externe ou interne à la plateforme. Elles nécessitent simplement l'ajout de code à l'extérieur, ou à l'intérieur du système. Elles sont de ce fait beaucoup moins coûteuses que les techniques matérielles. D'autre part, elles ne modifient que très légèrement les temps d'expérimentation. D'un autre coté, ce sont des méthodes intrusives, basées sur l'insertion d'instructions supplémentaires dans le code du système. Elles ne garantissent donc pas la même fiabilité de résultats que les méthodes matérielles.

2.1.4.3 Techniques basées sur la simulation

L'objectif des techniques basées sur la simulation [75] [76] est de fournir un environnement de vérification très peu coûteux comparativement aux environnements matériels et logiciels qui nécessitent l'existence d'une plateforme d'exécution, voire même de dispositifs de perturbation externes dans le cas des techniques matérielles. Les techniques décrites dans cette sous-section ne nécessitent aucun dispositif de génération de pannes externe, ni même de plateforme d'exécution.

Les expérimentations sont conduites sur des simulateurs qui fournissent un environnement d'exécution en tout point similaire à celui que serait une plateforme donnée. Le système en cours d'évaluation est exécuté sur le simulateur qui prend en charge l'injection des pannes. Tout comme pour les techniques matérielles et logicielles, ces injections consistent en des modifications du contenu des registres et de la mémoire.

Malheureusement, bien qu'au départ elles soient beaucoup moins coûteuses que les techniques matérielles et logicielles, les techniques basées sur la simula-

tion nécessitent des temps de calcul excessivement longs, spécialement lorsqu'un niveau de détail important est requis par les expérimentations. Pour diminuer ces temps, les expérimentations sont couramment réduites à de très courtes périodes de temps réel, ne permettant pas d'observer les conséquences d'une panne à long terme.

D'autre part, bien que la simulation permette une observabilité et un contrôle parfaits sur les expérimentations, le comportement du système simulé n'est pas nécessairement identique à celui qu'aurait présenté le même système exécuté sur une plateforme matérielle. Plus encore, il n'est pas toujours possible de trouver une correspondance directe entre les pannes réelles et les pannes injectées par un environnement de simulation.

2.1.4.4 Techniques hybrides

La dernière catégorie concerne les techniques qui combinent deux ou trois techniques de catégories différentes. Le but de telles combinaisons est de profiter des avantages des différentes catégories, maximisant ainsi les bénéfices tout en réduisant les inconvénients. À titre d'exemple, la technique combinée proposée par les auteurs de l'article [77] permet de tirer profit des avantages des techniques logicielles et des techniques basées sur la simulation. Similairement, celle proposée dans l'article [78] intègre les avantages des techniques matérielles et logicielles.

2.2 Problèmes, lacunes, et solutions

Malgré les nombreux environnements de développement disponibles, le cycle de développement matériel-logiciel présente deux principaux problèmes :

- L'écart existant entre les exigences et le premier modèle exécutable est considérable, ce qui est propice à l'apparition de nombreuses erreurs lors du processus de modélisation. Ce problème est accentué par le fait que les exigences sont complètement informelles et ne sont validées que sporadiquement, et de manière informelle,

- Les méthodes de tolérance aux pannes, reposant pourtant sur un ensemble de principes suffisamment restreint pour être vérifié formellement, sont la plupart du temps vérifiées par injection de pannes, ce qui ne permet pas de s'assurer de leur complète fiabilité[3],

- Le développement et la vérification des systèmes matériel-logiciel actuels sont dégradés par leur complexité et leur taille. Les méthodes formelles comme les techniques de vérification par simulation sont de moins en moins adaptées à la quantité de mémoire et au temps de calcul requis.

L'impact de ces manques sur le temps de développement et la qualité des systèmes peut être réduit considérablement par l'élévation du niveau d'abstraction et l'emploi de techniques de vérification formelles adéquates, comme nous le démontrons dans cette thèse. Notre solution s'articule autour de deux volets :

1. Vérification formelle, à un haut niveau d'abstraction, des modèles en développement. Cette vérification doit avoir lieu le plus tôt possible dans le cycle de développement, avant même la phase de modélisation du système en développement, au moment de l'énumération des exigences. Nous qualifions cette phase de *méthodologie d'ingénierie des exigences.*

2. Vérification, à un haut niveau d'abstraction, des méthodes de tolérance aux pannes. Contrairement à celles les plus utilisées actuellement, cette technique doit être formelle. Nous qualifions cette dernière de méthodologie de vérification formelle des méthodes de tolérance aux pannes.

Les solutions que nous proposons pour chacun de ces volets sont présentées dans les sous-sections qui suivent.

[3]Par fiabilité, nous entendons que la technique dont il est question détectera toute altération du flux de contrôle.

2.2.1 Exigences

En ce qui concerne les exigences, nous proposons de définir une méthodologie agissant à un haut niveau d'abstraction tout en reposant sur un formalisme et des méthodes de vérification solides. Nous présentons ici un résumé de notre approche. Cette dernière est comparée aux autres approches actuelles dans la section 2.3. La théorie complète ainsi que les applications et résultats expérimentaux sont respectivement détaillés dans les chapitres 3 et 4.

La formalisation et la vérification d'un modèle suit un processus séquentiel itératif vertical descendant. Les explications qui suivent résument ce processus. Le processus prend comme entrée les exigences informelles qui sont alors pré-arrangées suivant une représentation en langue naturelle des propriétés et un découpage structurel des fonctionnalités. Le système est décomposé en modules dont les fonctionnalités sont représentées par des actions. Les états dans lesquels le système peut se trouver sont représentés par des propriétés.

La représentation complète du système est, de ce fait, une collection de propriétés, actions, contradictions possibles entre les propriétés, et contraintes sur les propriétés et actions. Les propriétés sont représentées par des phrases en anglais lexicalement et syntaxiquement contrôlé. Les exigences ainsi semi-formalisées sont transformées en exigences pleinement formelles par l'utilisation d'un processus de pré-traitement linguistique qui transforme les phrases décrivant les propriétés en forme prédicative.

Une fois les phases de formalisation terminées, un processus de vérification de cohérence est exécuté. Ce processus génère un rapport sur les incohérences présentes dans le système, permettant ainsi aux concepteurs de procéder aux corrections nécessaires. Cette phase est suivie par le processus d'extraction des fonctionnalités manquantes qui génère une liste d'états du système pour lesquels aucune fonctionnalité n'est prévue, permettant aux concepteurs de procéder, une fois de plus, aux corrections nécessaires.

Les processus de vérification et d'extraction des fonctionnalités manquantes sont

de nature itérative, permettant ainsi aux concepteurs de procéder par raffinements successifs tout en s'assurant des gains de qualité entre ces raffinements.

2.2.2 Techniques de tolérance aux pannes

Concernant la vérification de techniques de tolérance aux pannes, nous proposons un cadre de conception permettant l'implémentation d'une méthode au sein de programmes génériques de taille réduite. Ces programmes sont traduits automatiquement en des modèles formels en langage PROMELA [66] qui sont ensuite vérifiés formellement à l'aide du *model-checker* SPIN [66]. La taille minimale de ces modèles minimise considérablement l'espace mémoire et le temps nécessaires comme nous le présenterons au chapitre 6.

Nous présentons ici un résumé de notre approche. Cette dernière est comparée aux autres approches actuelles dans la section 2.4 et la théorie complète ainsi que les applications et résultats expérimentaux sont respectivement détaillés dans les chapitres 5 et 6. Le processus de vérification commence par l'implémentation de la méthode de tolérance aux pannes dans un des programme génériques donnés qui est écrit dans un pseudo langage d'assemblage. Une fois la méthode implémentée, le programme est transformé automatiquement en deux modèles PROMELA, le premier étant un modèle sujet aux pannes, l'autre étant un modèle de référence non sujet aux pannes. La phase de vérification consiste à s'assurer que, suite à une panne, toute différence de comportement entre les deux modèles sera détectée par la méthode de tolérance aux pannes implémentée. Cette différence de comportement, et sa détection, est surveillée par une routine d'arbitrage.

La vérification formelle du système constitué des deux modèles et de l'arbitre permet d'explorer toutes les conséquences de pannes possibles, et donc de s'assurer de la robustesse de la méthode de tolérance aux pannes. L'utilisation de SPIN permet d'obtenir des scénarii illustrant les cas où des pannes modifiant le comportement du modèle fautif ne sont pas détectées par la méthode. De ce fait, le concepteur peut, ayant connaissance des défauts, raffiner sa méthode et procéder à de nouvelles vérifications. Non seulement ce processus de vérification permet de détecter

les problèmes de fiabilité des méthodes de tolérance aux pannes détectant les corruptions de flux de contrôle, mais, en prime, il fournit une aide au développement de ces dernières.

2.3 Ingénierie des exigences

L'ingénierie des exigences, c'est-à-dire la capture et la vérification des exigences à un stade précoce dans le cycle de développement, permet une amélioration de leur qualité, résultant en une réduction remarquable du coût de détection d'erreurs ainsi que du nombre d'erreurs dans la suite du processus de développement. La communauté matériel-logiciel manque grandement de méthodes d'ingénierie des exigences [2] [3]. Parallèlement, de nombreuses approches ont été développées au sein de la communauté logicielle. L'ensemble de ces approches peut être partitionné en quatre classes distinctes :

- Les approches de type scénario représentent les exigences en termes de comportements modélisés sous forme de scénarii,

- Les approches basées sur UML utilisent cette notation pour exprimer la structure et les fonctionnalités du système en développement,

- Les approches dites linguistiques visent l'extraction directe des fonctionnalités et l'analyse des exigences à partir de documents en langues naturelles.

- Enfin, les approches dites "haut-niveau" consistent à exprimer les exigences à un haut niveau d'abstraction en utilisant divers formalismes mathématiques appropriés.

Les sous-sections suivantes présentent une revue des techniques d'ingénierie des exigences actuellement utilisées en génie logiciel. Chaque approche présentée est comparée à l'approche que nous proposons.

2.3.1 Scénarii

Dans les approches de type scénario [79], les fonctionnalités du système en développement sont vues comme des comportements possibles. Ces derniers sont exprimés à l'aide de scénarii. Les scénarii peuvent être utilisés autant pour produire une spécification que pour la vérification de la spécification du prochain niveau de raffinement. De multiples approches existent. Comme nous le verrons dans les paragraphes qui suivent, ces dernières ne se situent pas au même niveau d'abstraction que l'approche que nous proposons puisqu'en effet, elles utilisent des séquences d'actions et d'échanges de messages structurées et détaillées.

La différence avec notre approche est que cette dernière permet au concepteur de penser à autant d'actions que besoin est, sans pour autant avoir à les organiser sous forme de séquence. Cette représentation se situe à un niveau d'abstraction plus élevé et permet d'analyser des exigences plus précoces que celles exprimés par les approches de type scénario. Les approches que nous résumons ici ne sont pas en concurrence avec la nôtre et ne doivent pas non plus être vues comme complémentaires. En effet, un processus de génération de scénarii pourrait aisément être ajouté à notre méthodologie.

Les auteurs de l'article [80] proposent un ensemble d'outils graphiques qui permettent l'élaboration et la vérification des exigences par l'utilisation de spécifications animées. Les exigences sont spécifiées par l'utilisation d'un langage très formel basé sur Z [81]. Bien qu'étant formelle, cette représentation est complexe et son utilisation requiert une certaine aisance avec les formalismes mathématique. De plus, la vérification de scénarii est effectuée manuellement en les animant. Bien qu'étant intéressante et probablement utile pour certains concepteurs, cette approche nécessite l'utilisation d'un formalisme complexe et ne fournit pas pour autant une vérification automatique. En comparaison, notre approche agit à un niveau d'abstraction plus élevé et utilisant une notation moins complexe, tout en fournissant une vérification automatique.

L'article [82] présente une notation visant la description de scénarii en util-

isant des vues multiples. Les auteurs définissent une notion claire de ce que sont la cohérence et la complétude d'un ensemble de scénarii, ainsi que d'un modèle d'exigences. Cet article présente aussi un outil permettant la vérification de cohérence et complétude automatique d'un ensemble d'exigences. Ces derniers sont exprimés sous forme d'ensembles de scénarii. Cette technique est similaire à notre approche dans la mesure où elle pratique la vérification d'un ensemble de scénarii tandis que nous effectuons la vérification d'un ensemble d'actions. Néanmoins, nous agissons respectivement à des niveaux d'abstractions différents.

En dernier lieu, les auteurs de l'article [83] présentent une approche similaire à la précédente. Ces derniers développent une méthode fournissant une élaboration et vérification des exigences basées sur l'utilisation de scénarii. Tel que mentionné précédemment, la différence entre une telle approche et la nôtre est que la première agit au niveau des scénarii, nécessitant une connaissance un peu plus approfondie de l'organisation du système qui n'est pas nécessaire avec l'utilisation de notre approche. De plus, la différence avec notre approche est que, tandis que les approches présentées dans les articles [82] et [83] focalisent sur les scénarii, nous focalisons sur les fonctionnalités, lesquelles sont des parties des scénarii se situant à un plus haut niveau d'abstraction. La dérivation de scénarii à partir d'un ensemble d'actions est possible tandis qu'il n'est pas nécessaire de connaître l'ensemble d'un scénario pour exprimer une fonctionnalité isolée.

2.3.2 UML

Comme son nom l'indique, UML (*Unified Modeling Language*) est un langage de modélisation unifiant les notations proposées par Booch, Rumbaugh, et Jacobson, enrichi par un certain nombre de contributions additionnelles. UML a pour but de fournir une notation permettant d'exprimer et de documenter les systèmes en développement. La représentation d'un système est effectuée par l'utilisation de multiples diagrammes spécifiques permettant de représenter les caractéristiques des modèles telles que les exigences fonctionnelles, les relations entre diverses entités, et la visualisation du flux de contrôle, pour n'en nommer que quelques unes.

UML propose un standard d'échange des modèles ainsi qu'un "méta-modèle" définissant la syntaxe de la notation UML. Il ne s'agit pas d'une méthodologie ni d'un processus particulier pour le développement des systèmes mais plutôt d'une notation générale permettant la description des systèmes et de leurs fonctionnalités. Cette notation a été utilisée dans nombre de travaux récents portant sur la capture et la vérification des exigences. Son haut niveau d'abstraction et ses diverses fonctionnalités en font un bon candidat pour un champ de recherche tel que celui au sein duquel nous nous situons.

Des méthodologies ont été proposées pour la vérification de modèles UML. À titre d'exemple, l'article [84] présente VIATRA, un outil de vérification formelle des modèles UML basée sur un ensemble de transformations visuelles automatiques. Cette approche est basée sur les transformations de graphes. Elle permet d'effectuer une vérification de cohérence, complétude et dépendance des exigences dans un modèle UML. Une approche similaire, basée sur un ensemble d'automates abstraits, est présentée par les auteurs de l'article [85].

L'article [86] présente une manière d'exprimer les exigences qui est similaire à la nôtre. Les auteurs proposent d'utiliser des outils dédiés à la capture et l'analyse des exigences. Ces outils permettent la génération de documentation et de tests en fournissant une vérification de syntaxe et sémantique. Similairement aux approches de type scénarii, les fonctionnalités sont représentées en utilisant des *use cases* en faisant abstraction totale des pré- et post-conditions, contrairement à notre approche. La vérification de cohérence des exigences est effectuée de manière isolée uniquement. Contrairement à ce que notre approche permet, aucune vérification de cohérence d'un ensemble de fonctionnalités n'est possible ici.

D'autres approches comme celle présentée par les auteurs des articles [87] et [88] proposent un cycle complet de développement et évaluation matériel basé sur l'utilisation d'UML. Bien que très complets et utiles, les processus décrits dans ces articles négligent totalement les exigences précoces. Les exigences prises en considération présentent, comme toujours, un niveau de détail déjà très précis.

L'article [89] montre que l'utilisation d'UML pour la spécification des exigences

précoces présente encore un certain nombre de problèmes. De notre point de vue, nous considérons qu'UML n'est pas adapté à la modélisation d'exigences aussi précoces que celles que nous abordons dans notre méthodologie. Ceci est principalement dû au niveau de détail de la notation UML. En effet, bien qu'UML puisse être utilisé pour la représentation d'exigences très abstraites comme celles auxquelles nous faisons référence, les processus de vérification qui sont fournis ne permettent pas, du moins pour le moment, d'analyser ces dernières. C'est pourquoi nous pensons que notre méthodologie pourrait aisément être greffée au dessus d'UML, fournissant ainsi un processus d'ingénierie des exigences supplémentaire et plus précoce, à un niveau d'abstraction supérieur. Les principes d'une telle greffe appliquée à l'approche présentée dans l'article [88] sont présentés dans la section 2.2.1.

2.3.3 Linguistique

Comme mentionné dans les sections précédentes, les exigences précoces sont en général exprimées sous forme de documents écrits en langue naturelle. Bien que qualifiés d'*informels*, ces documents respectent néanmoins un certain nombre de règles syntaxiques et grammaticales sur lesquelles la langue écrite se base. Une variété de techniques linguistiques permet d'effectuer nombre de vérifications sur de tels documents. L'application de ces techniques à la vérification des exigences semble parfaitement naturelle et des travaux récents présentent des solutions et résultats intéressants.

Les auteurs de l'article [90] proposent de fournir une assistance linguistique au processus de développement des modèles. Cette assistance repose sur des algorithmes permettant l'identification automatique des concepts et éléments d'un modèle. Cette identification aide les concepteurs dans la création et la documentation de leurs modèles. Néanmoins, cette approche ne fournit ni vérification de cohérence, ni extraction de fonctionnalités manquantes. En comparaison, notre méthodologie permet de fournir une assistance linguistique limitée dans la mesure où le concepteur doit représenter les exigences sous forme de fonctionnalités structurées. Néanmoins, nous permettons vérification de cohérence et extraction de

fonctionnalités manquantes.

Les auteurs de l'article [91] présentent GOOAL (Graphic Object-Oriented Analysis Laboratory), un outil dédié à l'analyse graphique de modèles orienté-objet. Cet outil prend en entrée une description écrite en langue naturelle et la transforme en un modèle phrase après phrase. Le concepteur doit assister le processus de transformation et valider les décisions les unes après les autres. Similairement au travaux de l'article [90], GOOAL vise à fournir uniquement des mécanismes de transformation semi-automatique, aucune vérification quelconque n'est abordée.

L'article [92] par contre propose une application linguistique pour l'analyse des *use cases*. Comme nous l'avons vu pour les deux méthodes précédemment décrites, les techniques linguistiques sont utiles pour fournir une assistance à la capture des exigences, néanmoins, le manque de structure des documents ne permet pas d'aborder le problème de la vérification. Les auteurs de l'article [92] stipulent que les *use cases* permettent de capturer les exigences de manière structurée. Utilisant les *use cases* comme structure et la langue naturelle pour la description des fonctionnalités, cet article décrit les techniques qui peuvent être utilisées pour supporter une analyse sémantique des *use cases*. Cette approche est similaire à celle que nous adoptons, à ceci près qu'elle repose sur l'utilisation de *use cases*, c'est-à-dire de scénarii, et non de fonctionnalités. De plus, le processus de vérification n'est pas aussi complet que le nôtre et aucune extraction des fonctionnalités manquantes n'est fournie.

Les auteurs de l'article [93] présentent une autre méthode intéressante basée sur les *use cases* et l'analyse automatique de leur cohérence. Les exigences sont capturées séparément sous forme de *use cases* en langue naturelle. Ces derniers sont ensuite automatiquement transformés en diagrammes UML. Cette approche permet uniquement l'acquisition et la formalisation des exigences. La vérification est considérée en tant que future recherche. D'autre part, l'extraction de fonctionnalités manquantes n'est pas mentionnée dans cet article.

Une dernière approche intéressante est celle du projet PROSPER, présentée dans les articles [94] et [95]. PROSPER est un outil de vérification basé sur

SMV [63] (*Symbolic Model Verification*)[4]. Ce dernier fournit des mécanismes de vérification permettant de s'assurer qu'une spécification écrite sous la forme d'un automate dans le langage *inout* (langage dédié de SMV) satisfait un ensemble de propriétés écrites dans la logique temporelle CTL [4]. La spécification des exigences est écrite en CTL [4]. Les propriétés que le modèle doit respecter sont écrites dans un anglais lexicalement et syntaxiquement contrôlé. Les propriétés ainsi écrites sont par la suite traduites automatiquement en CTL de manière à être utilisées pour valider la spécification CTL du modèle. Le processus de formalisation que nous utilisons dans notre approche est similaire à celui utilisé par les auteurs de PROSPER. La différence entre nos deux approches réside dans l'application. En effet, les auteurs de PROSPER spécifient leur modèle formellement et utilisent l'anglais pour spécifier informellement les propriétés à vérifier tandis que nous utilisons l'anglais pour spécifier une partie de notre modèle.

2.3.4 Haut-niveau

Les approches dites de "haut-niveau" regroupent diverses techniques reposant sur des formalismes mathématiques. Un des formalismes les plus significatifs est Z [81]. Il s'agit d'un formalisme mathématique dédié à la représentation formelle des systèmes, permettant ainsi d'effectuer de nombreuses vérifications toutes aussi formelles. Z est similaire aux formalismes tels que LOTOS [38] [39] [42], CSP [41], CCS [40], et SDL (*Specification and Description Language*) [96]. Tous ces formalismes sont très puissants en termes de représentation et vérification de modèles. Néanmoins, ce sont aussi des langages qui sont complexes, peu intuitifs, et nécessitent souvent un apprentissage préalable de la part des concepteurs amenés à les utiliser, ce qui est exactement ce que nous tentons d'éviter avec notre méthodologie.

Les auteurs de l'article [97] présentent une méthodologie dans laquelle les exigences sont modélisées à l'aide d'un langage formel. Un outil basé sur le *model-checker* SMV [63] permet ensuite d'effectuer la vérification formelle du modèle.

[4]SMV est la version non commerciale de FormalCheck [65].

Cette approche est très similaire à celle présentée par les auteurs de l'article [98]. Ces approches utilisent un formalisme complexe dont l'utilisation nécessite un ensemble d'exigences déjà très détaillé. De par ce fait, elles ne sont pas adaptées à un ensemble d'exigences précoces qui nécessite des langages permettant une modélisation à un niveau d'abstraction beaucoup plus élevé.

Certains travaux visent à faciliter l'utilisation de tels formalismes, ainsi les auteurs de l'article [99] présentent une méthodologie dans laquelle les fonctionnalités d'un domaine sont représentées par des tâches. L'exécution d'une tâche est contrôlée par un ensemble de pré-conditions et d'évènements entrants. Une fois effectuée, une tâche produit des évènements sortants et satisfait un ensemble de post-conditions. Les processus de vérification disponibles consistent à s'assurer de la validité de propriétés de *vivacité* [20] et *sécurité* [20]. Cette approche est similaire à la nôtre dans la mesure où il est possible de comparer les tâches aux fonctionnalités. Néanmoins, les niveaux de vérification sont passablement différents, comme nous le verrons au chapitre 3.

L'article [100] propose une méthodologie basée sur la représentation des exigences à l'aide de *feuilles de style* XML [101]. Ces derniers sont utilisés pour générer des documents d'exigences. En prime, les auteurs introduisent des heuristiques de vérification et de production de métriques permettant l'évaluation de la qualité des exigences et de leur vérification. Cette approche semble intéressante et prometteuse, néanmoins, comme celles présentées dans cette section, elle présente quelques manques vis à vis de la vérification.

2.3.5 Lacunes

De multiples méthodologies de vérification des exigences existent. Elles utilisent différents formalismes et techniques de vérification. Chaque méthodologie présente des forces et des faiblesses et toutes résolvent des problèmes importants. Nous identifions néanmoins les problèmes majeurs suivants :

- D'une part, les méthodes fournissant des processus de vérification solides

reposent sur l'utilisation de formalismes complexes dont l'utilisation n'est pas adaptée à la capture d'exigences précoces.

- D'autre part, les méthodes adéquates pour la capture des exigences précoces reposent sur les langues naturelles ou la description abstraite de *use cases* qui ne permettent pas, ou très peu, de processus de vérification automatique.

- À notre connaissance, il n'existe pas de méthodologie permettant l'extraction des exigences manquantes.

Comme nous l'avons mentionné au long des sections précédentes, l'approche que nous proposons apporte des solutions aux problèmes énoncés. La représentation des exigences est basée sur un formalisme structurel léger et facile d'utilisation. Un processus de pré-traitement linguistique permet de transformer cette représen-tation en modèle formel sur lequel vérification de cohérence et extraction des exigences manquantes sont effectués. Cette approche est introduite dans la section suivante et est présentée en détail dans le chapitre 3.

2.4 Tolérance aux pannes : vérification formelle

Comme nous l'avons vu dans la section 2.1.4, Les méthodes traditionnellement utilisées pour la vérification des techniques de tolérance aux pannes sont basées sur les principes d'injection de pannes durant l'exécution. Trois catégories de techniques co-existent : matérielles, logicielles, et reposant sur la simulation. Les deux premières techniques permettent une vérification en temps réel mais leur utilisation entraîne des coûts non négligeables, particulièrement du fait que celles-ci nécessitent la présence d'une plateforme d'exécution. La dernière technique consiste en une exécution du système sur un simulateur, permettant ainsi d'effectuer des vérifications à moindre coût, mais nécessitant malheureusement des temps de calcul excessifs.

Une catégorie additionnelle regroupe les méthodes basées sur l'application des principes et outils formels. La littérature contient peu d'articles sur ce sujet, la

majeure partie des travaux de vérification portant sur les méthodes d'injection de pannes, mais certains sont néanmoins fort intéressants. Nous distinguons trois sous-catégories d'approches :

- Les approches de type *model-checking* reposent sur une modélisation formelle du système à tester et emploient des *model-checkers* pour vérifier les propriétés de détection des pannes.
- Les approches de type *theorem-proving* reposent sur une modélisation mathématique de la méthode de tolérance aux pannes qui doit être testée et vérifient cette dernière en procédant par preuves mathématiques par l'utilisation de *theorem provers*.
- Enfin, les autres approches consistent à modéliser les systèmes à évaluer en utilisant des langages formels et à les tester à l'aide de calculs d'équivalence, ou voire parfois, par simulation.

Quelle que soit la catégorie, toutes les méthodes procèdent par injection de pannes dans le modèle formel. Les sous-sections suivantes présentent une revue des travaux portant sur l'emploi de ces méthodes formelles. Chacune est résumée et comparée à l'approche que nous proposons.

2.4.1 Approches de type *model-checking*

Dans les approches de type *model-checking*, le système de tolérance aux pannes à évaluer est formalisé dans un langage de description d'automates et la vérification est effectuée par un *model-checker* qui s'assure que le modèle satisfait un ensemble de propriétés, la plupart du temps écrites en LTL [20] ou en CTL [4]. En règle générale, tout comme les approches non formelles, les approches de type *model-checking* diffèrent dans le modèle de panne, c'est-à-dire dans la manière dont les pannes sont injectées.

L'article [102] présente une autre approche qui consiste à définir le modèle de panne comme étant une interface de la technique de tolérance aux pannes en cours

d'évaluation. Cela permet de ne pas devoir recourir à un ensemble de pannes ou de scénarii d'erreurs générés manuellement. L'approche présentée dans cet article utilise le *model-checker* symbolique SAL ainsi qu'un certain nombre de concepts permettant d'abstraire le problème étudié au niveau de SAL.

Le système à évaluer consiste en un algorithme de démarrage d'un décodeur. Cet algorithme garantit un démarrage dénué d'erreurs et sans retard en présence d'un composant erroné. La vérification de ce système à l'aide du *model-checker* SAL *Symbolic Analysis Laboratory* permet d'examiner les quelques milliards d'états générés en quelques dizaines de minutes. Contrairement à la nôtre, cette approche n'utilise pas de programmes génériques réduits. La vérification du système produit un nombre d'états très important et nécessite de ce fait des temps de calcul considérablement plus longs que ceux que nous obtenons avec notre approche. D'autre part, l'approche présentée ici ne permet pas d'obtenir des scénarii illustrant les cas où des pannes ne sont pas détectées par la technique en évaluation.

Les auteurs de l'article [103] proposent une méthodologie de vérification automatique des méthodes de tolérance aux pannes qui est basée sur les techniques de *model-checking*. Le choix de telles techniques réside le fait que ces dernières permettent de procéder à la vérification d'automates de manière systématique et totalement automatisée. Ainsi, toute technique de tolérance aux pannes modélisée sous forme d'un automate peut être vérifiée automatiquement par *model-checking*. La méthodologie présentée dans cet article utilise le *model-checker* SMV [63]. Bien qu'ayant peu été utilisé par la communauté purement matérielle, SMV jouit d'une réputation fort bonne au sein des communautés logicielle et matériel-logiciel.

Les auteurs proposent de modéliser le système à évaluer sous forme d'un système de transitions étiquetées [104]. Ils fournissent un ensemble de principes de modélisation pour représenter un tel programme ainsi qu'un outil de traduction qui permet de passer d'une telle représentation au langage *inout* utilisé par SMV. La vérification du système repose sur un ensemble de formules CTL qui représentent les propriétés intrinsèques que doit respecter la technique de tolérance aux pannes en cours d'évaluation. De manière similaire à la nôtre, la méthodologie proposée

dans cet article se veut être générale, c'est-à-dire applicable à la vérification d'un grand nombre de techniques, contrairement à la majeure partie des approches formelles qui visent la vérification d'une unique technique de tolérance aux pannes. Néanmoins, contrairement à la nôtre, leur approche n'utilise pas de programmes génériques réduits et ne permet pas non plus d'obtenir des scénarii illustrant les cas où des pannes modifiant le comportement du modèle fautif ne sont pas détectées par la méthode.

L'article [105] présente une technique alliant *model-checking* et injection de pan-nes. Les auteurs introduisent un environnement de vérification expérimental dédié à l'évaluation des techniques de tolérance aux pannes logicielles complexes. Cet environnement prend en compte le fait que les systèmes et les techniques de tolérance aux pannes qui y sont implémentées doivent respecter des procédures de certification de plus en plus sévères. La stratégie proposée montre comment la spécification rigoureuse d'une technique de tolérance aux pannes basée sur une description architecturale adéquate permet une amélioration de la vérification de son implémenta-tion dans des conditions réelles. En ce sens, l'article [105] appuie les idées sur lesquelles notre approche repose. En effet, ces auteurs illustrent comment des contraintes telles que l'utilisation de procédures rigoureuses dans le développement des techniques de tolérance aux pannes peut mener, à long terme, à un bénéfice important du point de vue de leur vérification, permettant ainsi d'obtenir des techniques beaucoup plus fiables.

Les conclusions préliminaires montrent que l'application de cette approche à une partie restreinte d'un système commercial tolérant aux pannes permet de clarifier son usage ainsi que son utilité pour la vérification. L'approche combinée des méthodes formelles et des techniques d'injection de pannes est intéressante. En effet, d'une part l'emploi d'une méthode formelle permet de vérifier la fiabilité de la méthode, d'autre part, l'injection de pannes permet de d'identifier les scénarii d'erreur pour lesquels la technique de tolérance aux pannes laisse échapper les comportements erronés. Nous retrouvons ces deux possibilités avec notre approche qui permet, par l'utilisation de formules spécifiques d'employer SPIN pour faire

une vérification complète tout en générant les scénarii de pannes pour lesquels la méthode évaluée n'agit pas correctement. L'approche décrite dans cet article a néanmoins recours à des techniques d'injection de pannes par simulation relativement lentes, et n'utilise pas de modèle minimal pour la vérification.

Les auteurs de l'article [106] présentent DAWN ; un environnement de vérification reposant sur les réseaux de Petri (*Petri Nets*) [107] et la logique linéaire temporelle (LTL). Tout comme SPIN sur lequel notre approche repose, l'environnement décrit dans cet article supporte à la fois la représentation simple des actions atomiques d'un algorithme et l'expression intuitive de ses propriétés temporelles. La définition du modèle de pannes consiste à spécifier l'impact de la présence des pannes sur le système au sein duquel est implémentée la méthode de tolérance aux pannes. Il faut donc déterminer, pour chaque système modélisé, l'impact des pannes sur ce système, ce que nous n'avons pas à faire dans notre approche dans la mesure où nous utilisons les mêmes programmes génériques quelle que soit la technique d'injection de pannes concernée. D'autre part, cet article ne mentionne rien à propos de la génération des scénarii d'erreurs pour lesquels la technique de tolérance aux pannes n'est pas fiable.

2.4.2 Approches de type *theorem-proving*

Les approches de type *theorem-proving* reposent sur des techniques de raisonnement automatisées. Le système à évaluer est représenté de manière formelle sous forme d'équations et la vérification est effectuée à partir d'un ensemble de règles de manipulation permettant d'inférer des verdicts au sujet du système modélisé. Cette vérification est conduite de manière automatique par des *theorem-provers*.

L'article [108] illustre un système de vérification utilisant un *theorem-prover*. Les auteurs présentent une méthode de représentation et de vérification des techniques de tolérance aux pannes fournissant : a) une représentation permettant de modéliser les systèmes matériels à différents niveaux d'abstraction, et b) un système de génération de preuves haut-niveau au sujet d'un système matériel non trivial. Le système à évaluer est modélisé à l'aide de réseaux de Petri étendus, appelés

réseau de flux (*flow net*). Ce modèle s'accompagne d'un ensemble de propriétés relatives au comportement dudit système. La procédure de vérification consiste à prouver les propriétés du système en cours d'évaluation. Ceci est effectué de manière automatisée par le *theorem-prover* ITP qui exécute et manipule symboliquement le modèle donné. Une propriété est prouvée par l'utilisation d'hypothèses et d'axiomes explicitement définis.

Cette approche est très différente de celle que nous proposons, particulièrement en ce qui concerne l'aspect vérification. De notre point de vue l'avantage d'une approche basée sur le *model-checking* est que, contrairement au *theorem-proving*, les procédures de type *model-checking* conduisent à des vérifications dites décidables, c'est-à-dire à propos desquelles le système arrive toujours à un résultat. Les approches de type *theorem-proving* ne peuvent garantir la décidabilité des résultats et cette limite est malheureusement immuable.

Les auteurs de l'article [109] proposent une méthodologie de génération de pannes dédiée à la vérification par injection de pannes. Cette dernière emploie un formalisme de modélisation des méthodes de tolérance aux pannes qui permet la génération d'une représentation arborescente où chaque chemin, de la racine à une feuille est une formule logique. Cet ensemble de formules constitue un ensemble de séquences de tests qui peut être utilisé comme des séquences d'entrées à un système d'injection de pannes.

Il est possible d'obtenir le même résultat à l'aide de notre approche. En effet, à partir de l'implémentation d'une technique de tolérance aux pannes dans un de nos programmes génériques réduits, il est possible de générer, par l'utilisation de SPIN, non pas seulement les séquences de pannes pour lesquelles la technique de tolérance n'est pas fiable mais aussi l'ensemble complet des séquences de pannes qui peuvent se produire sur ce modèle. Néanmoins, dans la mesure où cet ensemble de pannes est fortement lié au programme générique, cet ensemble n'est probablement pas réutilisable sur d'autres programmes.

2.4.3 Autres approches

Un certain nombre d'approches autres que celles reposant sur le *model-checking* ou le *theorem-proving* existent. Ces dernières consistent la plupart du temps à modéliser les systèmes à évaluer en utilisant des langages formels et à les tester à l'aide de calculs d'équivalence, ou voire parfois, par simulation.

À titre d'exemple, l'article [110] présente une approche de vérification reposant sur des techniques de transformation. Le système en cours d'évaluation est représenté en utilisant une logique temporelle. Les pannes sont modélisées comme étant causées par un ensemble d'opérateurs de pannes. Ces opérateurs agissent sur la modélisation du système en illustrant les changements d'états dus aux pannes, permettant ainsi de détecter les défauts de fiabilité de la technique de tolérance aux pannes en cours d'évaluation.

Les auteurs de l'article [111], quant à eux, présentent tout un cycle de développement et vérification des méthodes de tolérance aux pannes. Cet environnement utilise SDL [96], un langage de spécification exécutable, pour la modélisation et la vérification des techniques de tolérance aux pannes. Le système et le modèle de pannes sont tous deux modélisés en SDL. La vérification consiste en une analyse d'atteignabilité (*reachability analysis*) ainsi qu'une vérification par exécution de scénarii.

Ces dernières approches sont très différentes de celle que nous proposons. Elles ne permettent en aucun cas l'extraction de l'ensemble des scénarii d'erreurs pour lesquels la technique de tolérance aux pannes en cours d'évaluation n'est pas fiable.

2.4.4 Lacunes

En dépit des travaux sur la vérification formelle des méthodes que nous avons présentés, très peu de techniques sont vérifiées formellement et la communauté privilégie les méthodes d'injection de pannes matérielles, logicielles, et plus particulièrement celles reposant sur la simulation, en raison de leur faible coût financier. Deux raisons principales expliquent la sous-utilisation des méthodes formelles :

Le problème d'explosion d'états – Les algorithmes employés par les *model-checkers* sont P-ESPACE [63] complets, c'est-à-dire que l'espace mémoire nécessaire à la vérification est une fonction exponentielle de la taille du modèle à vérifier.

Comme nous l'expliquons au cours du chapitre 5, notre approche n'est pas dérangée par une telle limite dans la mesure où les techniques de tolérance aux pannes sont implémentées dans des programmes génériques réduits dont la taille permet d'éviter ledit problème d'explosion d'états.

Les difficultés de modélisation – Tel que mentionné dans l'article [102], la plupart des langages utilisés par les *model-checkers* ont été développés pour les spécifications matérielles et logicielles. Ils ne sont pas adaptés à la description des techniques de tolérance aux pannes et des modèles de pannes à un niveau d'abstraction approprié.

De notre point de vue, il nous semble, au contraire, qu'il est tout à fait possible d'utiliser ces langages de manière tout à fait satisfaisante. Il est possible de modéliser une technique, quel que soit le langage dans lequel elle est originellement écrite et quel que soit son niveau d'abstraction, dans un langage dédié au *model-checking*. Nous l'illustrons d'ailleurs avec notre approche où nous transformons une technique implémentée dans un langage d'assemblage en un modèle PROMELA, vérifié ensuite à l'aide du *model-checker* SPIN.

Comme nous avons pu le constater, tous les travaux présentés dans les sous-sections précédentes (2.4.1, 2.4.2, et 2.4.3) se heurtent soit au problème d'explosion d'états, soit à des difficultés de modélisation, soit aux deux. Comme nous le précisons dans le chapitre 5, l'approche que nous proposons s'affranchit de tels problèmes et est en mesure de se substituer aux méthodes de vérification traditionnellement utilisées.

2.5 Résumé

Dans ce chapitre, nous avons présenté l'état de l'art concernant le développement des systèmes matériel-logiciel robustes. Nous avons commencé par une présentation du cycle de développement, des environnements de conception, et des techniques de vérification des systèmes et techniques de tolérance aux pannes. À la suite de cette présentation, nous avons identifié deux des lacunes et problèmes majeurs de ce cycle de développement et avons présenté l'ébauche de la solution que nous proposons.

Cette solution s'articulant autour de deux volets, nous avons, dans les deux sections suivantes présenté une analyse détaillée de l'état de l'art détaillé relatif à chacun des axes, à savoir, l'ingénierie des exigences et les méthodes de vérification des techniques de tolérance aux pannes. Au cours de ces analyses détaillées de l'état de l'art relatif à chacun des axes, nous avons comparé chaque volet de notre approche à celles que nous avons présenté. Les quatre chapitres qui suivent présentent en détail les aspects théoriques et expérimentaux des deux volets de la solution que nous proposons.

CHAPITRE 3

INGÉNIERIE DES EXIGENCES – THÉORIE

L'ingénierie des exigences que nous proposons vise leur formalisation, améliorant ainsi la documentation, et la détection rapide et précoce des problèmes potentiels, entraînant ainsi une amélioration de la qualité dans la suite du cycle de développement, et enfin une réduction des coûts de production.

La méthodologie que nous proposons est dédiée à être greffée au dessus du cycle de développement matériel-logiciel. Elle consiste en la transformation d'un ensemble d'exigences donné en une représentation formelle à très haut niveau d'abstraction. Le système est représenté par l'ensemble des fonctionnalités qu'il fournit.

Le formalisme utilisé est suffisamment léger (dénué de notations mathématiques complexes) pour être facilement mis en place et en même temps suffisamment formel pour permettre une automatisation du traitement (vérification, extraction des exigences manquantes, etc.).

Dans ce chapitre, nous présentons d'abord les principes de notre méthodologie pour ensuite détailler les fondements théoriques sur lesquels elle repose. Le chapitre suivant présentera son implémentation et les études de cas sur lesquelles reposent nos résultats d'évaluation. Les résultats expérimentaux obtenus, présentés dans le chapitre 5, ainsi que nos publications [10], [11] et [12]. Les résultats expérimentaux obtenus confirment la pertinence de notre approche.

3.1 Présentation

La méthodologie que nous proposons découle de l'intersection des quatre approches orthogonales discutées au chapitre 2 ; scénarii, UML, linguistique, et méthodes formelles. Il vise la formalisation ainsi que la vérification de cohérence et de complétude booléen des fonctionnalités du système décrit par les exigences.

Au niveau d'abstraction que nous proposons, nous concevons un système comme étant une collection de modules coopérants et/ou indépendants qui définissent son architecture et auxquels sont attachées des fonctionnalités. Les fonctionnalités d'un module sont des actions que celui-ci peut effectuer. La description d'une action fait abstraction complète de la manière dont celle-ci est exécutée. Nous nous attardons simplement aux déclencheurs et aux répercussions de cette dernière, c'est-à-dire l'état dans lequel le système se trouve avant et après qu'elle ait été exécutée. Une action possède donc :

- Un ensemble de pré-conditions représentant l'état dans lequel le système doit se trouver pour que l'action soit déclenchée,
- Un ensemble de post-conditions représentant l'état dans lequel le système se trouvera à la fin de l'action.

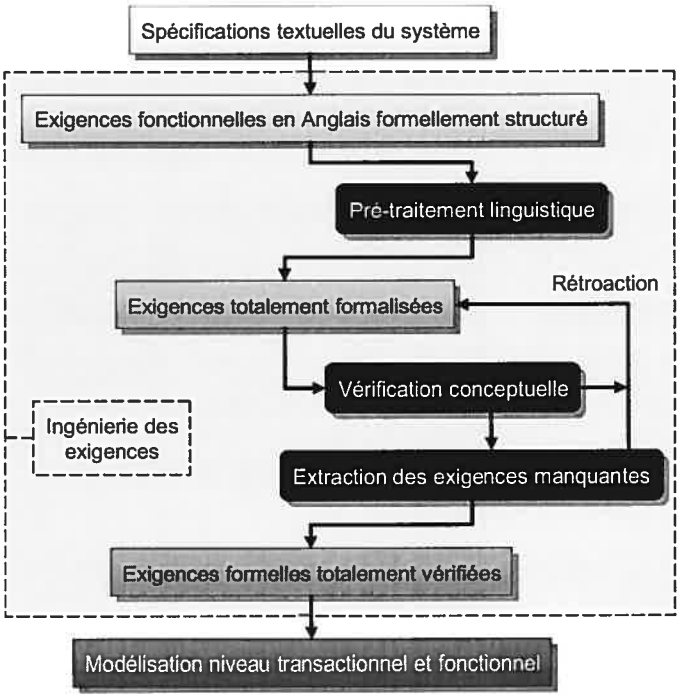

Figure 3.1: Méthodologie

Cette représentation est détaillée de manière précise dans la section 3.2. Bien qu'il soit question d'actions et d'états du système, les phases de vérification abordent

le système d'un point de vue uniquement statique. Aucune simulation n'est effectuée et il n'est pas tenu compte de l'évolution du système dans le temps. La raison de ce choix est qu'au niveau d'abstraction auquel nous nous situons, les fonctionnalités décrites par les exigences sont souvent présentées de manière très abstraites, risquant dans ces cas là de compromettre toute analyse supposant un quelconque aspect dynamique. La capture et la vérification des fonctionnalités suit un processus itératif dont nous illustrons le fonctionnement décrit au chapitre 2, section 2.2.1 à l'aide de la figure 3.1.

Collecte des exigences

Celles-ci se composent de documents décrivant l'architecture et les fonctionnalités du système en langage naturel. Ces descriptions sont parfois accompagnées de figures illustratrices.

Extraction et semi-formalisation des fonctionnalités

Les fonctionnalités sont manuellement extraites des documents et sont exprimées sous une forme semi-formelle ; les propriétés des pré- et post-conditions sont exprimées en utilisant un anglais standard lexicalement et syntaxiquement restreint. Le concepteur n'a donc pas besoin d'élaborer un ensemble complexe de noms, abréviations, ou prédicats pour représenter les connaissances. Chaque propriété est décrite à l'aide d'une phrase.

Formalisation complète des fonctionnalités

L'ensemble de fonctionnalités semi formel est automatiquement formalisé. Au cours de cette phase, les pré- et post-conditions sont traduites sous forme prédicative. Bien qu'il ne s'agisse pas de vérification à proprement parler, cette transformation permet d'ores et déjà de détecter certaines ambiguïtés dans l'expression des conditions. Le pré-traitement linguistique est détaillé à la section 3.3.

Vérification de la cohérence des exigences

Cette vérification se base sur un ensemble de règles préalablement établies.

Les résultats de cette analyse permettent au concepteur de prendre connaissance des incohérences, conflits, etc. qui peuvent exister entre les fonctionnalités capturées. À ce stade, le concepteur peut choisir de modifier les fonctionnalités, puis de relancer une phase de vérification. Le détail des règles de cohérence est donné dans la section 3.4.

Extraction des fonctionnalités manquantes

Les fonctionnalités manquantes sont extraites via un calcul de complétude. À ce stade, le concepteur peut ajouter les fonctionnalités manquantes et relancer une vérification de cohérence suivie d'une nouvelle extraction des fonctionnalités manquantes. Il est à noter que ce calcul permet uniquement d'extraire les fonctionnalités manquantes par rapport à un ensemble donné. Des explications détaillées sont données dans la section 3.5.

Fin du processus de formalisation/vérification

L'ensemble de fonctionnalités capturées à été formalisé et vérifié, les concepteurs peuvent aborder la génération d'un modèle de plus bas niveau dans un langage tel que SystemC ou SystemVerilog.

Les quatre sections qui suivent traitent respectivement du formalisme utilisé, du processus de pré-traitement linguistique, des règles de vérification de cohérence, et finalement du calcul de complétude pour l'extraction des fonctionnalités manquantes.

3.2 Formalisme

Tel que mentionné au début de la section 3.1, le système étudié est une collection de modules coopérants et/ou indépendants qui définissent son architecture et auxquels sont attachées des fonctionnalités. La représentation formelle d'un tel système repose sur les ensembles suivants :

Propriétés – Les différents états possibles de chaque module sont définis par des propriétés qui sont liées au module en question. L'ensemble de toutes les

propriétés du système est dénoté Π. Les propriétés sont traitées dans la section 3.2.1.

Actions – Les fonctionnalités d'un module sont définies par des actions qui sont liées au module en question. L'ensemble de toutes les actions du système est dénoté A. Les actions sont traitées dans la section 3.2.2.

Contradictions – Les propriétés contradictoires sont énumérées par paires. Cet ensemble d'énumérations est dénoté X. Les contradictions sont traitées dans la section 3.2.3.

Deux ensembles distincts contenant des propriétés contradictoires sont automatiquement contradictoires l'un vis à vis de l'autre.

Contraintes – Un ensemble T de contraintes temporelles sur les propriétés, et un ensemble Δ de contraintes déontiques sur les compositions d'actions permettent l'énonciation de certaines restrictions particulières au système. Les contraintes sont traitées dans la section 3.2.4

Formellement, le nom d'un module est désigné par le symbole μ tandis que le nom d'une propriété ou d'une action est désigné par le symbole ν. Les quatre sous-sections qui suivent traitent respectivement des propriétés, actions, contradictions, et contraintes.

3.2.1 Propriétés

Une propriété symbolise une partie (ou la totalité) d'un état du module auquel elle est associée. Cette dernière est dénotée $\pi \in \Pi$ et se définit comme étant une composition d'éléments formellement exprimée $\pi = [\ \nu \cdot \mu \cdot \lambda^{\pi} \cdot \theta\ ]$, où :

- ν est le nom de la propriété,
- μ est le nom du module auquel la propriété est associée,
- $\lambda^{\pi} \subseteq \Pi$ est un ensemble de sous-propriétés composant π, cette dernière étant de ce fait une conjonction de propriétés,

- θ est une chaîne de caractères documentant la propriété.

À toute propriété π correspond sa négation notée $\neg\pi$, implicitement définie et appartient à l'ensemble des propriétés dites exploitables, c'est-à-dire étant disponibles à des fins de modélisation du système : $\forall\pi \in \Pi \ : \exists\neg\pi \in \Pi$.

Selon que π est atomique ou composée, $\lambda^{\pi} = \emptyset$ ou bien $\lambda^{\pi} = \{\pi_0, ..., \pi_i\}$ avec $\forall i \in \mathbf{N} \ : \pi_i \in \Pi$. L'utilisation de sous-propriétés permet l'encapsulation d'un ensemble de propriétés en une seule, permettant ainsi de définir divers niveaux de granularité dans la description des fonctionnalités. Bien que cela puisse limiter les performances au niveau d'une implémentation, la profondeur des niveaux d'encapsulation est, sur le plan théorique, illimitée.

Considérons un système producteur-consommateur composé de deux entités ; un producteur et un consommateur. Les états des diverses entités du système producteur-consommateur sont formalisés comme suit:

Producteur – Les propriétés atomiques $\pi_{11}, \pi_{12}, \pi_{13}$ sont liées au producteur et symbolisent respectivement les états d'inactivité, de préparation de données, et d'envoi de ces dernières :

- $\pi_{11} = [\ inactif \cdot prod \cdot \emptyset \cdot$ "producteur inactif" $]$
- $\pi_{12} = [\ preparation \cdot prod \cdot \emptyset \cdot$ "producteur en préparation de données" $]$
- $\pi_{13} = [\ envoi \cdot prod \cdot \emptyset \cdot$ "producteur en envoi de données" $]$

Consommateur – Les propriétés $\pi_{21}, \pi_{22}, \pi_{23}$ sont liées au consommateur et symbolisent respectivement les états d'inactivité, de réception de données, et de traitement de ces dernières :

- $\pi_{21} = [\ inactif \cdot cons \cdot \emptyset \cdot$ "consommateur inactif" $]$
- $\pi_{22} = [\ reception \cdot cons \cdot \emptyset \cdot$ "consommateur en réception de données" $]$
- $\pi_{23} = [\ traitement \cdot cons \cdot \emptyset \cdot$ "consommateur traitement de données" $]$

Système – Finalement, la propriété π_0 qui symbolise le fait que le système complet est inactif est une propriété composée des sous propriétés π_{11} et π_{21}.

Autrement dit, le système est considéré comme inactif si le producteur et le consommateur sont tous deux inactifs :

- $\pi_0 = [\ inactif \cdot sys \cdot \{\pi_{11}, \pi_{21}\} \cdot$ "système inactif" $]$

Les propriétés définies ici illustrent la formalisation abstraite des états d'un système communicant simple. Cette formalisation est donnée à titre d'exemple. Au cours du raffinement des exigences, d'autres propriétés décrivant divers états du système et de ses modules devraient être ajoutées suivant le même principe si cela s'avérait nécessaire.

3.2.2 Actions

Une action symbolise une fonctionnalité du module auquel elle est liée. Elle est dénotée $\alpha \in A$ et est formellement définie comme $\alpha = [\ \nu \cdot \mu \cdot \phi \cdot \psi \cdot \lambda^{\alpha} \cdot \theta\ ]$, où :

- ν est le nom de l'action,
- μ est le nom du module susceptible d'effectuer l'action α,
- $\phi \subseteq \Pi$ dénote l'ensemble de pré-conditions nécessaires au déclenchement de α,
- $\psi \subseteq \Pi$ dénote l'ensemble de post-conditions accompagnant la fin de l'exécution de α,
- $\lambda^{\alpha} \subseteq A$ est un ensemble ordonné (une liste) de sous-actions composant α, cette dernière étant de ce fait un regroupement d'actions,
- θ est une chaîne de caractères documentant l'action.

Une action fait passer le module auquel elle est liée (et donc le système) de l'état courant à un nouvel état. Les ensembles de pré- et post-conditions sont des conjonctions de propriétés faisant respectivement référence à l'état du système avant le

déclenchement et après l'exécution de l'action dont elles font partie, tel qu'illustré par la fiture 3.2. Ces conditions sont respectivement définies comme $\phi = \{\pi_0, ..., \pi_i\}$ et $\psi = \{\pi'_0, ..., \pi'_j\}$ avec $\forall i, j \in \mathbf{N} \ : \pi_i, \pi'_j \in \Pi$.

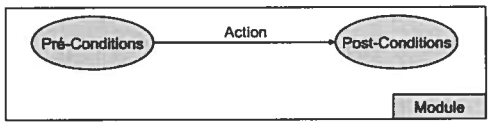

Figure 3.2: Action

L'ensemble de sous-actions λ^α peut être vide ($\lambda^\alpha = \emptyset$) ou bien contenir un certain nombre d'actions devant être effectuées de manière séquentielle ou parallèle (voir figure 3.3. Dans ces deux derniers cas, λ^α est respectivement défini comme $\lambda^{\alpha-seq} = \{\alpha_0, ..., \alpha_k\}$ ou bien $\lambda^{\alpha-par} = \{\alpha_0, ..., \alpha_k\}$ avec $\forall k \in \mathbf{N} \ : \alpha_k \in A$.

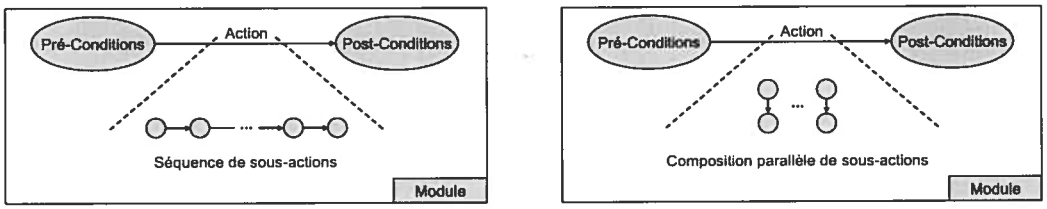

Figure 3.3: Sous-actions séquentielles et parallèles

Tout comme pour les propriétés, l'utilisation de sous-actions permet l'encapsulation d'un ensemble d'actions en une seule, permettant ainsi de définir divers niveaux de granularité dans la description des fonctionnalités. Ici encore, bien que cela puisse limiter les performances au niveau d'une implémentation, la profondeur des niveaux d'encapsulation est, sur le plan théorique, illimitée. À titre d'exemple, reprenons le système producteur-consommateur dont les états sont décrits dans la section 3.2.1 et formalisons les fonctionnalités qui lui sont propres.

Producteur – Les actions atomiques $\alpha_{11}, \alpha_{12}, \alpha_{13}$ sont liées au producteur et symbolisent respectivement les passages de l'état inactif à la préparation des données, de la préparation à l'envoi, et de l'envoi à l'état inactif.

- $\alpha_{11} = [\ prepare \cdot prod \cdot \{\pi_{11}\} \cdot \{\pi_{12}\} \cdot \emptyset \cdot$ "inactif à préparation" $]$

- $\alpha_{12} = [\ envoi \cdot prod \cdot \{\pi_{12}, \pi_{21}\} \cdot \{\pi_{13}\} \cdot \emptyset \cdot$ "préparation à envoi" $]$

- $\alpha_{13} = [\ inactif \cdot prod \cdot \{\pi_{13}\} \cdot \{\pi_{11}\} \cdot \emptyset \cdot$ "envoi à inactif" $]$

On notera que les pré-conditions de l'action α_{12} stipulent que l'envoi de données ne peut commencer que si le consommateur est inactif, c'est-à-dire prêt à recevoir lesdites données.

Consommateur – Les actions atomiques $\alpha_{21}, \alpha_{22}, \alpha_{23}$ sont liées au consommateur et symbolisent respectivement les passages de l'état inactif à la réception des données, de la réception au traitement, et du traitement à l'état inactif :

- $\alpha_{21} = [\ recoit \cdot cons \cdot \{\pi_{21}, \pi_{13}\} \cdot \{\pi_{22}\} \cdot \emptyset \cdot$ "inactif à réception" $]$

- $\alpha_{22} = [\ traite \cdot cons \cdot \{\pi_{22}\} \cdot \{\pi_{23}\} \cdot \emptyset \cdot$ "réception à traitement" $]$

- $\alpha_{23} = [\ inactif \cdot cons \cdot \{\pi_{23}\} \cdot \{\pi_{21}\} \cdot \emptyset \cdot$ "traitement à inactif" $]$

On notera que les pré-conditions de l'action α_{21} stipulent que la réception de données débute lorsque le consommateur est inactif et que le producteur est dans l'état d'envoi de données.

Système – Un cycle complet d'échange de données débutant lorsque le système au complet est inactif et revenant à ce même état après l'échange des données est formalisé par une action composée contenant la séquence appropriée d'actions du producteur et du consommateur :

- $\alpha_0 = [\ ech \cdot sys \cdot \{\pi_0\} \cdot \{\pi_0\} \cdot \{\alpha_{11}, \alpha_{12}, \alpha_{21}, \alpha_{13}, \alpha_{22}, \alpha_{23}\}^{\alpha-seq} \cdot$ "cycle d'échange" $]$

Tout comme les propriétés définies dans la section 3.2.1, les actions définies ici formalisent, à titre d'exemple, les fonctionnalités abstraites de notre système communicant simple. Encore une fois, au cours du raffinement des exigences, d'autres actions décrivant diverses fonctionnalités du système et de ses modules devraient être ajoutées suivant le même principe si cela s'avérait nécessaire.

3.2.3 Contradictions

Une grande partie de l'analyse de cohérence repose sur la recherche de contradictions entre les pré- et post-conditions des différentes fonctionnalités formalisées. Le fait que deux propriétés $\pi_x, \pi_y \in \Pi$ sont contradictoires est exprimé par un couple dénoté $\chi(\pi_x, \pi_y) \in X$. Par définition, toute propriété et sa négation sont contradictoires ; $\forall \pi \in \Pi \ : \chi(\pi, \neg\pi) \in X$. Tel que mentionné plus tôt, deux ensembles distincts contenant des propriétés contradictoires sont contradictoires l'un vis à vis de l'autre: $\forall E_1, E_2 : \exists \pi_1 \in E_1, \pi_2 \in E_2 \rightarrow \chi(E_1, E_2)$.

À part la contradiction entre une propriété et sa négation et quelques rares exceptions qui sont de nature syntaxique, la majeure partie des contradictions est de nature sémantique, ne permettant donc pas de les inférer automatiquement. À titre d'exemple, les propriétés du producteur π_{11} et π_{12} (voir section 3.2.1) qui symbolisent respectivement l'état inactif et l'état de préparation des données sont contradictoires. En effet, le producteur ne peut être inactif et en préparation de données en même temps mais cela ne peut être détecté de manière certaine et automatique. Il appartient donc au concepteur de formaliser cette contradiction comme suit : $\chi(\pi_{11}, \pi_{12})$. Les autres contradictions seront formalisées suivant le même principe.

Bien entendu, l'énumération de la totalité des contradictions peut être particulièrement longue et fastidieuse. L'utilisation de variables dans la définition des contradictions pallie ce problème en permettant de formaliser la caractéristique d'un ensemble de contradictions au lieu d'énumérer tout l'ensemble. Ainsi, le fait qu'une propriété et sa négation sont contradictoires est simplement formalisé comme $\chi(\pi, \neg\pi)$ où $\pi \in \Pi$ est une variable représentant toutes les propriétés de l'ensemble Π.

3.2.4 Contraintes

Deux types de contraintes sont disponibles : les contraintes temporelles relatives aux propriétés, et les contraintes déontiques relatives aux compositions d'actions.

3.2.4.1 Contraintes temporelles

Les contraintes temporelles dénotent des restrictions sur les propriétés. Les divers opérateurs disponibles sont empruntés à la logique temporelle [20] et sont *toujours* et *fatalement*. Leur utilisation est définie comme suit :

- Une propriété devant toujours être satisfaite est dénotée $\tau_a(\pi) \in T$, où le $_a$ associé à τ fait référence à *always* en anglais.
- Similairement, une propriété devant fatalement être satisfaite est dénotée $\tau_e(\pi) \in T$.

Par exemple, nous pouvons spécifier que le producteur doit finir par envoyer une donnée. Ceci revient à spécifier que le producteur doit fatalement passer dans l'état d'envoi, c'est-à-dire que la propriété π_{13} doit fatalement être satisfaite :

- $\tau_e(\pi_{13})$

L'ensemble d'opérateurs temporels disponibles peut sembler restreint et le lecteur pourrait penser que, par exemple, *Until* et *Next* devraient être inclus. Néanmoins, les deux opérateurs utilisés sont suffisants et leur choix est intrinsèque à la nature statique de l'approche que nous proposons. En effet, au niveau d'abstraction auquel nous nous situons, bien que le modèle formel décrive le système sous un aspect dynamique (les pré-conditions sont les déclencheurs des actions), les diverses vérifications effectuées négligent totalement tout aspect dynamique, tel qu'expliqué dans la section 3.1.

3.2.4.2 Contraintes déontiques

Lors de son apparition, la logique déontique [112] était dédiée à la formalisation du discours en droit dans le but de permettre l'inférence automatique de verdicts pour des problèmes donnés. Plus tard, l'emploi de cette logique s'est diversifié à un grand nombre d'applications, spécialement les systèmes agents où elle est employée pour résoudre automatiquement les conflits entre agents [113] [114] [115].

Les contraintes déontiques que nous définissons ici dénotent des restrictions sur les compositions d'actions. Nous aurions pu utiliser les opérateurs de la logique linéaire temporelle, cependant, dans un souci de partitionnement, nous avons opté pour un ensemble d'opérateurs différents. Les opérateurs utilisés sont *obligation*, *permission*, et *interdiction*. L'utilisation des opérateurs est définie comme suit :

- Une composition d'actions obligatoires est dénotée $\delta_o(\lambda^\alpha) \in \Delta$, où la lettre $_o$ associée à δ fait référence à l'obligation.
- Similairement, une composition d'actions permises est dénotée $\delta_p(\lambda^\alpha) \in \Delta$.
- Finalement, une composition d'actions interdites est dénotée $\delta_i(\lambda^\alpha) \in \Delta$.

On notera que les compositions symbolisées par λ^α peuvent tout aussi bien être des compositions parallèles que séquentielles. D'autre part, seulement deux opérateurs sont en réalité suffisants ; *obligation* et *permission*, l'opérateur *interdiction* étant en fait la négation de l'opérateur *permission*. Néanmoins, l'utilisation de l'opérateur *interdiction* nous semble plus intuitive et c'est pourquoi nous avons finalement gardé les trois opérateurs.

À titre d'exemple, toujours dans notre système producteur-consommateur, le producteur ne peut pas passer directement de l'état inactif à l'envoi de données sans les avoir préalablement préparées. Ceci est exprimé formellement comme suit :

- $\delta_i(\{\alpha_{13}, \alpha_{12}\}^{\alpha-seq})$

Un autre exemple est le fait que le consommateur ne peut passer dans l'état de réception en même temps que le producteur passe dans l'état d'envoi, en effet, ces deux actions doivent être effectuées de manière séquentielle. Formellement :

- $\delta_i(\{\alpha_{12}, \alpha_{21}\}^{\alpha-par})$

En général, sera spécifié uniquement ce qui est interdit ainsi que ce qui est obligatoire, ce qui est permis étant, par définition, l'ensemble des combinaisons au sujet desquelles aucune contrainte n'est spécifiée, ainsi que ce qui est spécifié comme étant obligatoire (puisque tout ce qui est obligatoire est forcément permis).

3.2.5 Modèle de calcul

Le modèle de calcul de notre formalisme se rapporte à la logique propositionnelle pour les raisons suivantes :

- Les propriétés peuvent être comparées à des propositions atomiques,
- Les actions sont similaires à des programmes pouvant être composés séquentiellement ou parallèlement,
- Les actions sont, comme les programmes en logique propositionnelle, gardées par des pré- et post-conditions qui sont des conjonctions de propriétés,
- Enfin, l'intervalle de valeurs des variables est fini, permettant ainsi les calculs décidables.

De modestes différences avec la logique propositionnelle sont toutefois présentes :

- Les pré- et post-conditions sont uniquement des conjonctions de propriétés atomiques dans la mesure où nous restreignons le formalisme aux opérateurs de conjonction et de négation pour des raisons de simplicité. L'opérateur de disjonction reste cependant possible,
- L'addition de contradictions ainsi que de contraintes déontiques et temporelles permet l'expression de caractéristiques spécifiques qui ne seraient pas exprimables en logique propositionnelle.

Mis à part ces différences, les règles de dérivation appliquées aux propriétés et actions restent celles de la logique propositionnelle. La nature statique des contradictions et contraintes élimine la nécessité de règles de dérivations pour ces dernières. Un modèle de calcul pourrait être celui des réseaux de Petri [107] dans lesquels les contraintes et l'absence de contradictions pourraient être vérifiées sur un graphe de marquage.

3.3 Pré-Traitement linguistique

Le but du pré-traitement linguistique est la traduction de phrases exprimant des propriétés dans un anglais lexicalement et syntaxiquement restreint, auquel nous ferons référence en utilisant les termes "anglais restreint", en une forme prédicative. Ce traitement est uniquement une transformation syntaxique. Il ne s'agit en aucun cas d'une analyse sémantique des phrases données. Il est de ce fait sensiblement différent de celui que l'on retrouve dans les travaux sur l'outil PROSPER [95]. Notre traitement linguistique se compose des trois étapes suivantes :

- Analyse lexicale des phrases via l'utilisation d'une grammaire à clauses définies (DCG – Definite Clause Grammars),
- Représentation des dépendances sous formes de relations de dépendances,
- Traduction en logique des prédicats.

Les propriétés sont extraites à partir des phrases par un simple processus linguistique utilisant une DCG. Les DCG sont des mécanismes basés sur les grammaires hors contexte et font partie intégrante du langage Prolog [116] [117]. Une DCG permet d'exprimer une sous-partie lexicale et syntaxique bien définie de l'anglais à l'aide d'un ensemble restreint de règles de grammaire. Cet ensemble de règle est utilisé pour traduire les phrases écrites en anglais restreint en une représentation syntaxique arborescente [118].

Une phrase est définie de manière minimale comme étant la conjonction d'une expression nominale et d'une expression verbale. Chaque composant d'une phrase est ainsi défini tour à tour suivant les restrictions appliquées au sous-langage. À titre d'exemple, deux règles de grammaire seront nécessaires pour exprimer une expression de type verbe intransitif, avec ou sans complément circonstanciel.

Chaque règle décompose un sous-niveau syntaxique de la phrase, parcourant ainsi, de la racine jusqu'aux feuilles, la structure syntaxique arborescente de la phrase. À partir d'une phrase complète, les règles d'analyse syntaxique extraient

les composants jusqu'à l'unité linguistique indépendante la plus profonde sur le plan du graphe : le mot. Les mots sont encodés comme des entrées d'un lexique anglais prédéfini.

De manière à réduire la taille du lexique, un analyseur morphologique simple, inspiré de l'article [119] a été ajouté à l'analyseur lexical. Un tel analyseur permet la reconnaissance d'une variation verbale infléchie en se basant sur le tronc du verbe ainsi que sur les inflexions verbales anglaises possibles pour le verbe en question. Cet analyseur permet aussi la détection des dérivations nominales.

L'utilisation d'un tel mécanisme permet donc une réduction substantielle de la taille du lexique. Une fois identifiée, chaque unité terminale permettant la satisfaction d'une règle est rétro-propagée vers la règle précédente, jusqu'à la règle racine de la phrase. Une fois que la phrase complète est analysée, celle-ci est réduite à une simple fonction syntaxique. Ainsi, à titre d'exemple, la phrase "some data arrives to consumer", sera représentée comme suit :

```
phrase( expr_nominale( some, nom(data) ),
        expr_verbale( verbe(arrive),
                      expr_prepositionnelle( to, nom(consumer) )
                    )
      ).
```

Cette représentation ne peut pas être utilisée telle qu'elle est pour dériver une clause prédicative. Néanmoins, sa structure syntaxique nous donne suffisamment de renseignements pour permettre d'inférer les relations d'inter-dépendance entre les différentes unités.

La représentation basée sur les inter-dépendances consiste en la distinction entre les unités dites de tête et les unités dites dépendantes. Les relations entre une unité de tête et ses unités dépendantes sont décrites en termes de relations de dépendances sémantiquement justifiées. Huit relations syntaxiques universelles ont été définies en accord avec trois principaux types de dépendance linguistiques profondes ; complément, modification, et relations coordinatrices [120]. La relation de complément correspond aux actants à partir desquels un prédicat est caractérisé. Par exemple, "to arrive" est un verbe connu pour avoir deux actants :

1. l'objet ou la personne qui arrive,

2. le lieu : X arrive à/sur Y.

Ces deux actants correspondent aux arguments du verbe dans la forme prédicative. La relation de modification est apposée aux unités qui modifient la signification de leur tête, généralement en la restreignant. Par exemple, les adjectifs modifient les noms, les adverbes modifient les verbes, les compléments nominaux modifient leur noms tête, etc. Finalement, la relation coordinatrice est apposée aux constructions coordinatrices tel que "some data arrives on consumer a AND consumer b".

Les pré- et post-conditions représentent des évènements ou bien des états. Les variables et autres objets dépendent de ces prédicats, et finalement, les modificateurs raffinent ces arguments. Au stade actuel de notre étude, seules les relations de complément et de modification sont utilisées. Une grammaire complète permettrait l'utilisation de structures coordinatrices, mais pourrait cependant introduire des ambiguïtés à travers les pré- et post-conditions.

De manière à éviter toute ambiguïté possible, les structures syntaxiques complexes ne sont pas permises. Une fois qu'une relation de dépendance est inférée à partir de l'arbre d'analyse syntaxique, une assertion est faite. Cette assertion, appelée *arc* possède trois arguments:

1. l'unité tête,

2. le type de relation,

3. le dépendant.

L'unité tête et son dépendant sont identifiés par des numéros de référence ($\#Ref1$ et $\#Ref2$) qui sont récupérés à partir du lexique contrôlé durant l'analyse syntaxique.

```
arc(HeadUnit#Ref1, Re1, DependentUnit#Ref2).
```

Par exemple, lors de l'analyse syntaxique de la phrase "some date arrives on consumer", deux relations de dépendance sont enregistrées :

```
arc(arrive#1, I, data#2).
arc(arrive#1, II, consumer#3).
```

Les analyses distinctes sont possibles sur des phrases uniques. Si la grammaire n'est pas ambiguë, seulement une analyse syntaxique peut être suffisante. La règle d'analyse syntaxique qui suit spécifie qu'une phrase consiste en une expression nominale suivie par une expression verbale. La première unité nominale analysée sera présumée être le premier argument du verbe tête de la phrase :

```
phrase( phrase(NP, VP)) -->
  expr_nominale( NP, N, CarN),
  expr_verbale( VP, N, CarN, V#A, CarV),
  {
    ( arc( V#1, 'I', N)
       ;
      asserta( arc(V#1, 'I', N))
    )
  }.
```

La règle d'analyse syntaxique qui suit exprime le fait que si une expression verbale complexe composée d'un verbe ("arrives") et d'une expression prépositionnelle ("on consumer") est analysée, alors le nom tête de l'expression prépositionnelle est censé être le deuxième argument du verbe. De ce fait, un arc sera inséré.

```
expr_verbale( verbe(V#A, S), SujetNP, V#A) -->
  verbe(V#A), expr_prepositionnelle(NP, N),
  {
    (
      arc(V#1, 'II', N)
       ;
      assertz( arc(V#1, 'II', N)
    )
  }.
```

En utilisant le numéro de référence trouvé dans l'assertion, la représentation syntaxique arborescente de la phrase "some data arrives on consumer" peut être imprimée :

```
arrive [classe:verbe, temps:present]
I   data [classe:nom_commun, nombre:sg]
II   consumer [classe:nom_commun, nombre:sg]
```

Cette relation de dépendance permet de traduire la phrase originale en une représentation prédicative telle que :

```
arrive(data, consumer).
```

Les phrases sont ainsi remplacées par les propriétés correspondantes extraites. Cette traduction permet ainsi la formalisation complète et automatique des fonctionnalités décrites semi-formellement. Un tel traitement permet aux concepteurs d'ex-primer les fonctionnalités d'une manière semi-formelle, donc plus facile à utiliser, et permet un gain de temps considérable au niveau de la formalisation finale. Une fois la phase de traduction terminée, les fonctionnalités ainsi formalisée peuvent être utilisées à des fins de vérification.

3.4 Vérification de cohérence statique

La vérification de cohérence est basée sur un ensemble de règles de cohérence. Ces règles caractérisent les notions de cohérence basiques qui doivent être respectées par tout modèle. Le processus de vérification consiste à s'assurer que le modèle ne transgresse aucune règle. Chaque exemple de transgression d'une règle dénonce un problème dans le modèle. Cette de vérification de cohérence est statique et consiste à s'assurer que toutes les règles de l'ensemble de règles de cohérence, que nous présentons dans cette section, sont satisfaites.

Bien entendu, l'ensemble de règles présentées ici n'est pas fixé et de nouvelles pourraient être ajoutées pour répondre aussi bien à un besoin spécifique que général. L'ensemble des règles est partitionné en différents groupes suivant le problème concerné. Les sous-sections suivantes présentent en détail chaque groupe et son ensemble de règles. Chaque règle est énoncée formellement, conformément au formalisme présenté dans la section 3.2, et est accompagnée d'une définition informelle.

3.4.1 Définitions

Tel que mentionné dans la section 3.2, les composants sur lesquels repose la description du modèle sont les actions et les propriétés qui composent leurs pré- et post-conditions. Pour assurer une description cohérente, chaque action et propriété utilisée doit être correctement définie. Ainsi, l'ensemble de règles qui suit vise la vérification de l'existence d'une définition correcte pour chaque action et propriété utilisée dans la description du système.

Règle 1. $\forall \pi_i \in \Pi \ : \forall \pi_x \in \lambda_i^{\pi} \ : \pi_x \in \Pi$

Cette règle stipule que pour chaque propriété définie, toute propriété mentionnée dans son ensemble de sous-propriétés doit être définie. Par exemple, la propriété π_0 définie dans la section 3.2.1 et rappelée ici :

$$\pi_0 = [\ inactif \cdot sys \cdot \{\pi_{11}, \pi_{21}\} \cdot \text{“système inactif”}\ ]$$

ne transgresse pas la règle puisque ses deux sous-propriétés π_{11} ou π_{21} sont définies dans la section 3.2.1 elles aussi. Si ce n'était pas le cas, la règle ne serait pas satisfaite, dénotant un manque de clarté, ou un oubli, dans les exigences.

Règle 2. $\forall \alpha_i \in A \ : \phi_i \neq \emptyset \wedge \psi_i \neq \emptyset$

Toujours concernant les définitions d'actions, chaque action définie doit avoir un ensemble de pré- et post-conditions non vides. Ainsi, l'action fictive α_{f1} suivante :

$$\alpha_{f1} = [\ fictif \cdot prod \cdot \emptyset \cdot \{\pi_{f1}\} \cdot \{\alpha_{f2}\}^{\alpha-seq} \cdot \text{“action fictive”}\ ]$$

qui ne spécifie pas quel doit être l'état du système pour qu'elle soit déclenchée (ensemble de pré-conditions vide), transgresse la règle 2.

Règle 3. $\forall \alpha_i \in A \ : \forall \pi_x \in \phi_i, \pi_y \in \psi_i \ : \pi_x, \pi_y \in \Pi$

Au sujet des pré- et post-conditions attachées aux actions, toute propriété mentionnée dans un ensemble de pré- ou post-conditions doit être définie.

Ainsi, notre action fictive α_{f1} citée en exemple dans la règle 2 transgresse aussi cette règle dans la mesure où la propriété π_{f1}, à laquelle fait référence son ensemble de post-conditions, n'est définie nulle part dans les exigences.

Règle 4. $\forall \alpha_i \in A \ : \forall \alpha_x \in \lambda_i^{\alpha - \{par|seq\}} \ : \alpha_x \in A$

Finalement, similairement aux pré- et post-conditions, pour toute action, les actions faisant partie de son ensemble de sous-actions doivent être définies. L'action α_{f1} donnée en exemple pour la règle 2 transgresse donc aussi la règle puisque son ensemble de sous-actions fait référence à l'action α_{f2} qui n'est pas définie dans les exigences.

Règle 5. $\forall \alpha_x, \alpha_y \in A, x \neq y \ : \nexists \alpha_x, \alpha_y | \phi_x \subseteq \phi_y \wedge \not\chi(\phi_x, \phi_y) \wedge \psi_x \neq \psi_y.$

Il n'existe pas deux actions pouvant être déclenchées en même temps et dont les post-conditions sont différentes. Autrement dit, tout non déterminisme sera détecté par cette règle.

3.4.2 Enumérations

L'ensemble de règles concernant les énumérations a pour but de vérifier la cohérence des ensembles de propriétés, c'est-à-dire, les sous-propriétés et les pré- et post-conditions.

Règle 6. $\forall \pi_i \in \Pi \ : \forall \pi_x, \pi_y \in \lambda_i^{\pi}, x \neq y \ : \neg\chi(\pi_x, \pi_y)$

Aucune propriété ne possède de sous-propriétés contradictoires. Ainsi, la propriété :

$$\pi_{f2} = [\ fictive \cdot sys \cdot \{\pi_{11}, \pi_{12}\} \cdot \text{“propriété fictive”}\ ]$$

illustre un exemple transgressant la règle 6 puisque sa liste de sous-propriétés se compose de π_{11} et de π_{12} qui sont définies comme étant deux propriétés contradictoires dans la section 3.2.3. En effet, ces dernières dénotent différents états du même module et il est incohérent de spécifier qu'un composant doit se trouver simultanément dans différents états.

Règle 7. $\forall \alpha_i \in A \ : \forall \pi_x, \pi_y \in \phi_i, x \neq y \ : \neg\chi(\pi_x, \pi_y)$

Les pré-conditions d'une action ne contiennent pas de propriétés contradictoires. Par exemple, l'action suivante :

$$\alpha_{f3} = [\ fictif \cdot prod \cdot \{\pi_{11}, \pi_{12}\} \cdot \{\pi_{f1}\} \cdot \emptyset \cdot \text{“action fictive”}\ ]$$

illustre une incohérence transgressant la règle 7 puisque ses pré-conditions sont contradictoires.

Règle 8. $\forall \alpha_i \in A \ : \forall \pi_x, \pi_y \in \psi_i, x \neq y \ : \neg\chi(\pi_x, \pi_y)$

Similairement pour les post-conditions.

3.4.3 Redondance

Les règles de redondance servent à garantir qu'il n'existe pas de duplicata parmi les propriétés et les actions.

Règle 9. $\forall \pi_i, \pi_j \in \Pi, i \neq j \ : \nu_i \neq \nu_j \vee \mu_i \neq \mu_j$

Cette règle concerne les propriétés et stipule qu'il n'existe pas deux propriétés ayant le même nom se rattachant au même module. En d'autres termes, chaque propriété d'un module doit être unique. Les deux propriétés suivantes illustrent un exemple de transgression de cette règle :

$$\pi_{f3} = [\ fictive \cdot sys \cdot \{\pi_{11}, \pi_{12}\} \cdot \text{“propriété fictive”}\ ]$$
$$\pi_{f4} = [\ fictive \cdot sys \cdot \{\pi_{21}\} \cdot \text{“autre propriété fictive”}\ ]$$

puisqu'elles sont attachées au même module. Par contre, les propriétés suivantes :

$$\pi_{11} = [\ inactif \cdot prod \cdot \emptyset \cdot \text{“producteur inactif”}\ ]$$
$$\pi_{21} = [\ inactif \cdot cons \cdot \emptyset \cdot \text{“consommateur inactif”}\ ]$$

définies dans la section 3.2.1 ne transgressent pas la règle 9 puisqu'elles font référence à différents modules ; producteur et consommateur.

Règle 10. $\forall \alpha_i, \alpha_j \in A, i \neq j \ : \nu_i \neq \nu_j \vee \mu_i \neq \mu_j$

Tout comme pour les propriétés, cette règle stipule qu'il ne peut exister deux actions de même nom se rattachant au même module.

Règle 11. $\forall \alpha_i, \alpha_j \in A, i \neq j \ : \mu_i \neq \mu_j \vee \phi_i \neq \phi_j \vee \psi_i \neq \psi_j \vee \lambda_i^{\alpha-\{par|seq\}} \neq \lambda_j^{\alpha-\{par|seq\}}$

Cette règle stipule qu'il ne peut exister deux actions se rattachant au même module et ayant exactement les mêmes caractéristiques. Deux actions doivent se rattacher à différents modules ou bien se distinguer l'une de l'autre par leurs pré-, post-conditions, ou ensembles de sous-actions. Par exemple, les actions suivantes :

$\alpha_{f3} = [\ fictif_1 \cdot prod \cdot \{\pi_{11}, \pi_{12}\} \cdot \{\pi_{f1}\} \cdot \emptyset \cdot$ "action fictive" $]$
$\alpha_{f4} = [\ fictif_2 \cdot prod \cdot \{\pi_{11}, \pi_{12}\} \cdot \{\pi_{f1}\} \cdot \emptyset \cdot$ "action fictive" $]$

transgressent cette règle puisqu'elles sont en tout point identiques, excepté en ce qui concerne leurs noms.

3.4.4 Inter-énumérations

L'ensemble de règles dites d'*inter-énumération* vise la vérification de cohérence entre diverses énumérations de propriétés qui peuvent exister dans les définitions d'actions.

Règle 12. $\forall \alpha_i \in A, \lambda_i^{\alpha-seq} = \{\alpha_z, ...\} : \nexists \pi_x \in \phi_i, \pi_y \in \phi_z \mid \chi(\pi_x, \pi_y)$

Cette règle vérifie la cohérence des pré-conditions d'une action avec celles des sous-actions séquentielles dont elle est composée. Pour toute action possédant un ensemble de sous-actions devant être exécutées de manière séquentielle, il ne doit pas exister de contradiction entre les pré-conditions de l'action et celles de la première action de l'ensemble de sous-actions. À titre d'exemple, les deux actions suivantes :

$\alpha_{f5} = [\ fictif \cdot prod \cdot \{\pi_{11}\} \cdot \{\pi_{f1}\} \cdot \{\alpha_{f6}, \alpha_{f7}, \alpha_{f8}\}^{\alpha-seq} \cdot$ "action fictive" $]$

$\alpha_{f6} = [\ fictif \cdot prod \cdot \{\pi_{12}\} \cdot \{\pi_{13}\} \cdot \emptyset \cdot$ " autre action fictive" $]$

transgressent la règle. En effet, les pré-conditions de l'action α_{f5} contiennent la propriété π_{11} et les pré-conditions de sa première sous-action, α_{f6}, contiennent π_{12} tandis que π_{11} et π_{12} sont définies comme contradictoires dans la section 3.2.3.

Règle 13. $\forall \alpha_i \in A, \lambda_i^{\alpha-seq} = \{..., \alpha_z\} : \nexists \pi_x \in \psi, \pi_y \in \psi_z \mid \chi(\pi_x, \pi_y)$

Similairement à la règle 12, cette règle vérifie que pour toute action possédant un ensemble de sous-actions devant être exécutées de manière séquentielle, il n'existe pas de contradiction entre les post-conditions de l'action et celles de la dernière action de l'ensemble de sous-actions.

Règle 14. $\forall \alpha_i \in A, \alpha_z \in \lambda_i^{\alpha-par} : \nexists \pi_x \in \phi_i, \pi_y \in \phi_z \mid \chi(\pi_x, \pi_y)$

Cette règle procède à la vérification de la cohérence des pré-conditions d'une action avec celles des sous-actions parallèles dont elle est composée. Elle vérifie que pour toute action possédant un ensemble de sous-actions devant être exécutées en parallèle, il n'existe pas de contradiction entre les pré-conditions de l'action et celles de chaque action de l'ensemble de sous-actions. Ainsi une action telle que la suivante :

$\alpha_{f9} = [\ fictif \cdot prod \cdot \{\pi_{11}\} \cdot \{\pi_{f1}\} \cdot \{\alpha_{f6}, \alpha_{f7}, \alpha_{f8}\}^{\alpha-par} \cdot$ "action fictive"]

transgresse la règle du fait des pré-conditions de l'action α_{f6} (voir règle 12) sont en contradiction avec celles de l'action α_{f9}.

Règle 15. $\forall \alpha_i \in A, \alpha_z \in \lambda_i^{\alpha-par} : \nexists \pi_x \in \psi_i, \pi_y \in \psi_z \mid \chi(\pi_x, \pi_y)$

Similairement à la règle 14, cette règle vérifie que pour toute action possédant un ensemble de sous-actions devant être exécutées en parallèle, il n'existe pas de contradiction entre les post-conditions de l'action et celles de chaque action de l'ensemble de sous-actions.

Règle 16. $\forall \alpha_i \in A, \forall \alpha_x, \alpha_y \in \lambda_i^{\alpha-par} : \nexists \pi_a \in \phi_x, \pi_a \in \phi_y \mid \chi(\pi_a, \pi_b)$

Cette règle vérifie que les pré-conditions de deux actions faisant partie d'un ensemble de sous-actions exécutées en parallèle n'entrent pas en contradiction.

Règle 17. $\forall \alpha_i \in A, \forall \alpha_x, \alpha_y \in \lambda_i^{\alpha-par} : \nexists \pi_a \in \psi_x, \pi_b \in \psi_y \mid \chi(\pi_a, \pi_b)$

Similairement pour les post-conditions.

Règle 18. $\forall \alpha_i \in A, \forall \alpha_x, \alpha_{x+1} \in \lambda_i^{\alpha-seq} : \nexists \pi_a \in \psi_x, \pi_b \in \phi_{x+1} \mid \chi(\pi_a, \pi_b)$

Cette règle s'attarde sur les ensembles de sous-actions exécutées séquentiellement. Elle vérifie que dans un tel ensemble, dénoté $\lambda^{\alpha-seq}$, il n'existe pas deux actions qui se suivent et pour lesquelles les post-conditions de la première entrent en contradiction avec les pré-conditions de la suivante. Ainsi, considérons l'action suivante :

$$\alpha_{f7} = [\ fictif \cdot prod \cdot \{\pi_{13}\} \cdot \{\pi_{f1}\} \cdot \{\alpha_{f8}, \alpha_{f6}\}^{\alpha-seq} \cdot \text{“action fictive”}\ ]$$
$$\alpha_{f8} = [\ fictif \cdot prod \cdot \{\pi_{13}\} \cdot \{\pi_{11}\} \cdot \emptyset \cdot \text{“action fictive”}\ ]$$

dont l'ensemble des sous-actions contient la séquence α_{f8}, α_{f6}. Nous constatons que les pré-conditions de α_{f6}, contenant π_{12}, entrent en contradiction avec les post-conditions de l'action qui la précède puisque ces dernières contiennent π_{11}.

3.4.5 Contraintes temporelles

L'ensemble de règles dédié aux contraintes temporelles vérifie que ces dernières sont statiquement respectées dans le modèle.

Règle 19. $\forall \tau_e(\pi) : \exists \alpha \in A \mid \pi \in \psi$

Les propriétés qui doivent être fatalement satisfaites sont présentes dans au moins un ensemble de post-conditions. À titre d'exemple, la contrainte temporelle :

$$\tau_e(\pi_0)$$

n'est pas satisfaite si nous nous référons aux actions du système producteur-consommateur présenté dans la section 3.2, puisqu'aucune des actions définies ne contient π_0 dans ses post-conditions.

Règle 20. $\forall \tau_e(\pi) \ : \nexists \alpha \in A \mid \neg\pi \in \psi$

Les propriété devant toujours être satisfaites ne sont jamais niées dans quelque ensemble de post-conditions que ce soit. Ainsi, toujours en nous référant au système producteur-consommateur défini dans la section 3.2, la contrainte :

$\tau_a(\neg\pi_{21})$

n'est pas satisfaite puisque π_{21} apparaît dans les post-conditions de l'action α_{23}.

Règle 21. $\forall \tau_a(\pi) \ : \nexists \tau_a(\neg\pi) \wedge \nexists \tau_e(\neg\pi)$

Il n'existe pas de contradictions dans les contraintes temporelles : Si une contrainte spécifie qu'une propriété doit toujours être satisfaite, il n'existe pas de contrainte contredisant cette dernière. Par exemple :

$\tau_a(\neg\pi_{21})$ et $\tau_e(\pi_{21})$

sont des contraintes contradictoires puisque la première spécifie que π_{21} doit toujours être niée tandis que la seconde spécifie que celle-ci doit fatalement être satisfaite.

Règle 22. $\forall \tau_e(\pi) \ : \nexists \tau_a(\neg\pi)$

Similairement, si une contrainte spécifie qu'une propriété doit fatalement être satisfaite, il n'y a pas de contrainte spécifiant que sa négation doit toujours être satisfaite.

3.4.6 Contraintes déontique

L'ensemble de règles dédié aux contraintes déontiques vérifie que ces dernières sont statiquement respectées dans le modèle.

Règle 23. $\forall \alpha_i \in A \ : \nexists \lambda_x^{\alpha-par} \subseteq \lambda_i^{\alpha-par} \mid \delta_i(\lambda_x^{\alpha-par})$

Aucune composition interdite n'est présente dans les ensembles de sous-actions des actions définies. Par exemple, la contrainte :

$\delta_i(\{\alpha_{f7}, \alpha_{f8}\})$

est transgressée par l'action :

$\alpha_{f10} = [\ fictif \cdot prod \cdot \{\pi_{11}\} \cdot \{\pi_{f1}\} \cdot \{\alpha_{f6}, \alpha_{f7}, \alpha_{f8}\}^{\alpha-seq} \cdot$ "action fictive"]

qui contient la sous-séquence interdite dans son ensemble de sous-actions.

Règle 24. $\forall \alpha_i \in A \ : \exists \lambda_x^{\alpha-\{par|seq\}} \subseteq \lambda_i^{\alpha-\{par|seq\}} \mid \delta_o(\lambda_x^{\alpha-\{par|seq\}})$

Similairement pour les compositions obligatoires. Il existe des ensembles de sous-actions dans lesquels ces dernières sont présentes.

Règle 25. $\forall \delta_o(\lambda_i^{\alpha-\{par|seq\}}) \ : \nexists \delta_i(\lambda_j^{\alpha-\{par|seq\}})$

Une composition d'actions obligatoires n'est pas définie comme étant interdite ailleurs dans l'ensemble des exigences. Par exemple, la contrainte :

$\delta_o(\{\alpha_{f7}, \alpha_{f8}\})$

contredit celle qui est définie dans l'exemple de la règle 23.

Règle 26. $\forall \delta_i(\lambda_i^{\alpha-\{par|seq\}}) \ : \nexists \delta_o(\lambda_x^{\alpha-\{par|seq\}}) \wedge \nexists \delta_p(\lambda_y^{\alpha-\{par|seq\}})$

Similairement pour les compositions interdites.

3.5 Extraction des fonctionnalités manquantes

L'extraction des fonctionnalités manquantes consiste à vérifier la complétude de l'ensemble de fonctionnalités donné et à produire, le cas échéant, une liste de celles qui ne sont pas définies. Ce processus se base sur la logique Booléenne, plus précisément sur le calcul du complément d'une fonction.

Tel que défini dans la section 3.2, les fonctionnalités sont représentées par des actions gardées par des pré- et post-conditions. L'ensemble de pré-conditions

représente l'état déclencheur, c'est-à-dire la configuration dans laquelle le système doit être pour que l'action soit effectuée. L'ensemble de post-conditions représente la nouvelle configuration résultant de l'action effectuée.

Un ensemble de pré-conditions est une conjonction de propriétés dénotant un état du système dans lequel une action doit être exécutée. De ce fait, la collection d'actions définies peut être transposée en une fonction booléenne $\mathcal{F}$ exprimée sous forme de somme de produits. En effet, les produits correspondent aux actions tandis que les variables de ces derniers correspondent aux propriétés. À titre d'exemple, reprenons notre exemple de système producteur consommateur. Nous avons défini 6 propriétés ainsi que 6 actions atomiques que nous recopions ici pour plus de clarté :

Propriétés :

$\pi_{11} = [\ inactif \cdot prod \cdot \emptyset \cdot$ "producteur inactif" $]$

$\pi_{12} = [\ preparation \cdot prod \cdot \emptyset \cdot$ "producteur en préparation de données" $]$

$\pi_{13} = [\ envoi \cdot prod \cdot \emptyset \cdot$ "producteur en envoi de données" $]$

$\pi_{21} = [\ inactif \cdot cons \cdot \emptyset \cdot$ "consommateur inactif" $]$

$\pi_{22} = [\ reception \cdot cons \cdot \emptyset \cdot$ "consommateur en réception de données" $]$

$\pi_{23} = [\ traitement \cdot cons \cdot \emptyset \cdot$ "consommateur traitement de données" $]$

Actions :

$\alpha_{11} = [\ prepare \cdot prod \cdot \{\pi_{11}\} \cdot \{\pi_{12}\} \cdot \emptyset \cdot$ "inactif à préparation" $]$

$\alpha_{12} = [\ envoi \cdot prod \cdot \{\pi_{12}, \pi_{21}\} \cdot \{\pi_{13}\} \cdot \emptyset \cdot$ "préparation à envoi" $]$

$\alpha_{13} = [\ inactif \cdot prod \cdot \{\pi_{13}\} \cdot \{\pi_{11}\} \cdot \emptyset \cdot$ "envoi à inactif" $]$

$\alpha_{21} = [\ recoit \cdot cons \cdot \{\pi_{21}, \pi_{13}\} \cdot \{\pi_{22}\} \cdot \emptyset \cdot$ "inactif à réception" $]$

$\alpha_{22} = [\ traite \cdot cons \cdot \{\pi_{22}\} \cdot \{\pi_{23}\} \cdot \emptyset \cdot$ "réception à traitement" $]$

$\alpha_{23} = [\ inactif \cdot cons \cdot \{\pi_{23}\} \cdot \{\pi_{21}\} \cdot \emptyset \cdot$ "traitement à inactif" $]$

Procédons maintenant à l'isolation des ensembles de pré-conditions, nous obtenons :

$\phi_{\alpha_{11}} = \{\pi_{11}\}$ $\qquad$ $\phi_{\alpha_{21}} = \{\pi_{21}, \pi_{13}\}$

$\phi_{\alpha_{12}} = \{\pi_{12}, \pi_{21}\}$ $\qquad$ $\phi_{\alpha_{22}} = \{\pi_{22}\}$

$\phi_{\alpha_{13}} = \{\pi_{13}\}$ $\qquad$ $\phi_{\alpha_{23}} = \{\pi_{23}\}$

Et nous déduisons la fonction booléenne $\mathcal{F}$ représentant toutes les configurations du système pour lesquelles une action à effectuer est définie :

$$\mathcal{F} = \{\pi_{11}\} + \{\pi_{12}, \pi_{21}\} + \{\pi_{13}\} + \{\pi_{21}, \pi_{13}\} + \{\pi_{22}\} + \{\pi_{23}\}$$

L'extraction des fonctionnalités manquantes repose sur le calcul du complément booléen de la fonction $\mathcal{F}$, dénotée $\overline{\mathcal{F}}$. Cette dernière représente les états du système pour lesquels aucune action n'est prévue.

Le calcul du complément d'une fonction booléenne repose sur sa représentation cubique. Une telle représentation est un tableau dans lequel chaque ligne, appelée cube, représente un produit de la somme. La totalité du tableau représente la somme de produits. Nous disposons les propriétés en colonnes tandis que les cellules du tableau contiennent un bit initialisé en fonction du fait que la propriété soit : présente (1), présente et niée (0), ou bien absente (-) (*don't care*) dans le produit correspondant à la ligne.

	π_{11}	π_{12}	π_{13}	π_{21}	π_{22}	π_{23}
α_{11}	1	-	-	-	-	-
α_{12}	-	1	-	1	-	-
α_{13}	-	-	1	-	-	-
α_{21}	-	-	1	1	-	-
α_{22}	-	-	-	-	1	-
α_{23}	-	-	-	-	-	1

Table 3.1: Représentation cubique

L'ensemble d'actions est donc traduit en une représentation cubique dans laquelle les lignes représentent les actions et les colonnes représentent l'ensemble de toutes les propriétés du système. Les cellules sont initialisées avec la valeur appropriée, i.e. 1, 0 ou -, suivant le fait que la propriété soit présente, présente et niée, ou bien absente dans les pré-conditions de l'action correspondante. Le tableau 3.1 illustre la représentation cubique des actions de notre système producteur-consommateur.

Le résultat brut du calcul du complément donne le résultat présenté par le tableau 3.2. Chaque *don't care* représente les deux combinaisons 0 ou 1 possi-

bles. Les lignes qui en contiennent doivent dont être remplacées par les diverses combinaisons possibles.

	π_{11}	π_{12}	π_{13}	π_{21}	π_{22}	π_{23}
β_{01}	0	0	0	-	0	0
β_{02}	0	-	0	0	0	0

Table 3.2: Complément brut

Le complément raffiné est présenté par le tableau 3.3. À partir de celui-ci, nous pouvons extraire les cas pour lesquels aucune action n'est définie.

	π_{11}	π_{12}	π_{13}	π_{21}	π_{22}	π_{23}
$\beta_{01a-02a}$	0	0	0	0	0	0
β_{01b}	0	0	0	1	0	0
β_{02b}	0	1	0	0	0	0

Table 3.3: Complément raffiné

$\beta_{01a-02a} = \{\neg\pi_{11}, \neg\pi_{12}, \neg\pi_{13}, \neg\pi_{21}, \neg\pi_{22}, \neg\pi_{23}\}$

Le système est dans un état dans lequel aucun des modules n'est dans un état valide. Dans ce cas, le système est dans un état d'erreur et une action devrait effectivement être définie.

$\beta_{01b} = \{\neg\pi_{11}, \neg\pi_{12}, \neg\pi_{13}, \pi_{21}, \neg\pi_{22}, \neg\pi_{23}\}$

Le consommateur est inactif et le producteur n'est pas dans un état valide. D'après la définition du système, le consommateur a besoin que le producteur soit dans l'état d'envoi pour pouvoir transiter vers son état de réception des données. Si le producteur n'est pas dans un état valide, le consommateur est bloqué. Il s'agit là encore d'un état d'erreur pour lequel une action devrait être définie.

$\beta_{02b} = \{\neg\pi_{11}, \pi_{12}, \neg\pi_{13}, \neg\pi_{21}, \neg\pi_{22}, \neg\pi_{23}\}$

Le producteur est en train de préparer les données mais le consommateur n'est pas dans un état valide. Similairement au cas précédent, le producteur

ne pourra jamais passer dans l'état d'envoi des données puisqu'une des conditions est que le consommateur soit dans un état inactif, c'est-à-dire, prêt à recevoir. Comme précédemment, il s'agit de définir une action pour cet état d'erreur.

L'extraction des exigences manquantes est un processus important. La solution que nous proposons permet de calculer les fonctionnalités manquantes sur un très grand nombre d'exigences. Néanmoins, cette approche est limitée dans la mesure où l'ensemble de fonctionnalités manquantes extraites est uniquement le complément de l'ensemble de fonctionnalités définies. Dans certains cas de figures, cet ensemble pourrait être complet tandis qu'une partie des fonctionnalités système seraient manquante. En d'autres mots, l'extraction ne peut être faite à partir de rien.

En dépit de ces considérations, il reste que ce processus est utile dans la mesure où il permet d'apporter une certain réflexion sur un ensemble de fonctionnalités déjà donné, comme l'a montré l'exemple développé au cours de ce chapitre.

3.6 Complexité algorithmique

La complexité algorithmique des traitements que nous proposons dans ce chapitre diffère d'un module à l'autre. La complexité du traitement linguistique et celle du calcul du complément sont connues pour être respectivement linéaire et NP complète. La complexité de la vérification de cohérence nécessite une analyse. Cette vérification repose sur un ensemble de règles dont la complexité peut varier légèrement de l'une à l'autre. Néanmoins, nous pouvons la borner de la manière qui suit.

Borne inférieure – Considérons notre règle la plus simple: la règle 1. La vérification de cette règle repose sur l'algorithme suivant:

- Pour toute propriété $\pi_i \in \Pi$,
  - Pour toute sous-propriété π_j de π,

* Vérifier que π_j soit définie.

Ainsi, si dans le pire des cas chaque propriété possède un sous-ensemble de propriétés égal à Π, nous nous trouvons face à une complexité polynomiale de l'ordre de $|\Pi|^2$.

Borne supérieure – Considérons à présent notre règle la plus complexe: la règle 15. La vérification de cette dernière repose sur l'algorithme suivant:

- Pour toute action $\alpha_i \in A$,
  - Pour toute sous-action α_j de α_i,
    * Pour toute post-condition de α_j; $\pi_j \in \psi_j$,
      · Pour toute propriété de α_i; $\pi_i \in \psi_i$,
        * Pour tous les couples de contradictions,
          - π_i et π_j ne satisfont pas ce couple.

Ainsi, si dans le pire des cas:

1. chaque action possède un sous-ensemble d'actions contenant toutes les actions possibles
2. les post-conditions des actions contiennent toutes les propriétés possibles,
3. L'ensemble des contradictions couvre toutes les propriétés,

nous avons alors une complexité polynomiale de l'ordre de: $|A|^2 * |\Pi|^4$

La complexité de l'ensemble de règles que nous avons défini est donc bornée par $\Omega(|\Pi|^2)$ et $O(|A|^2 * |\Pi|^4)$. Cette complexité limite la taille des modèles à vérifier. Néanmoins, dû au niveau d'abstraction auquel nous nous situons, il est possible, comme nous le verrons dans le prochain chapitre, de modéliser des systèmes assez importants tout en gardant un temps d'analyse raisonnable.

Cependant, l'ensemble de règles que nous avons défini au cours de nos travaux de recherche n'est pas définitif; il est possible de définir de nouvelles règles d'une

complexité beaucoup élevée. Les sous-actions d'une action peuvent à leur tour contenir des sous-actions et notre analyse de cohérence ne considèrera que le premier niveau. Il est évident que l'ajout de règles permettant de considérer tous les niveaux d'une hiérarchie d'actions et de propriétés amènerait notre borne supérieure à une complexité exponentielle.

3.7 Résumé

Dans ce chapitre, nous avons présenté notre méthodologie d'ingénierie des exigences destinée à être greffée entre les phases de collection des exigences et de modélisation du système.

Cette méthodologie permet la représentation des fonctionnalités sous une forme semi-formelle alliant structure formelle et langage naturel. Cette représentation est ensuite traduite en un modèle purement formel via un ensemble de techniques linguistiques. À partir de cette formalisation, deux formes de vérification sont effectuées tour à tour. La première vise à vérifier la cohérence des fonctionnalités à partir d'un ensemble de règles pré-déterminées. La seconde vise l'extraction des fonctionnalités manquantes.

Étant donné le niveau d'abstraction, les erreurs de cohérence et fonctionnalités manquantes détectées ne seront bien entendu pas du même type que celles pouvant être détectées à plus bas niveau. En effet, les erreurs détectées par notre méthodologie s'apparentent plus à des erreurs dues au processus de formalisation qu'à de véritables erreurs dans les exigences. Néanmoins, il est important et bénéfique de détecter les erreurs dues au processus de formalisation le plus tôt possible.

D'autre part, cette méthodologie s'adresse principalement à une classe de modèles comportementaux, c'est à dire à des spécifications décrivant les comportements d'un système. Elle ne peut donc pas être utilisée pour traiter la totalité de la spécification d'un système dans la mesure où elle ne serait pas adaptée pour décrire certaines exigences comme les exigences architecturales par exemple.

Les seules limites de cette approche sont :

- Sa complexité,
- Le fait que l'ensemble du processus doit être appliqué de nouveau pour tout nouvel ensemble d'actions ou de propriétés ajouté à la suite d'une itération.

Néanmoins, étant donné le niveau de détail auquel nous nous situons, l'utilisateur ne sera pas affecté par ces limites, comme les montrent les études de cas présentées dans le chapitre suivant.

En conclusion, l'utilisation d'une telle méthodologie pour la formalisation et la vérification des exigences permet une amélioration de la documentation, et la détection rapide et précoce des problèmes potentiels, entraînant ainsi une hausse de la qualité dans la suite du cycle de développement, et enfin une réduction des coûts de production.

CHAPITRE 4

INGÉNIERIE DES EXIGENCES – OUTILS ET APPLICATIONS

Ce chapitre porte sur l'implantation de notre méthodologie d'ingénierie des exigences ainsi que son évaluation. Dans un premier temps, nous présentons l'implémentation des concepts théoriques énoncés au chapitre 3 sous forme d'un outil prototype. Ensuite, nous illustrons l'utilisation de ce prototype sur trois études de cas. Une première très simple qui nous a permis de tester le bon fonctionnement de notre outil. Deux autres de plus grande envergure, portant respectivement sur l'architecture RapidIO et un ensemble d'exigences fourni par PMC-Sierra, nous ont permis d'obtenir des résultats d'envergure industrielle.

L'existence d'une implémentation permet tout d'abord d'affirmer les concepts sur lesquels notre méthodologie repose tout en rendant possible son évaluation au delà d'un cadre purement théorique. Les résultats de cette évaluation, basée sur des études de cas pertinentes, nous permettent de confirmer l'efficacité et la pertinence d'une telle méthodologie.

4.1 Implémentation

L'implémentation présentée dans ce chapitre intègre la totalité des trois processus de traitement énoncés au chapitre 3. La majeure partie de cette implémentation est en Prolog. Ce choix est justifié par le fait que ce langage permet d'exprimer les règles de fonctionnement d'un processus sous une forme très naturelle. D'autre part, l'existence de mécanismes tels que le *backtracking*, intrinsèques au langage, permet un prototypage rapide en focalisant uniquement sur les concepts à implémenter et non sur la manière dont cette implémentation est effectuée.

L'implémentation complète de l'outil ainsi que les études de cas et un manuel d'utilisation sont disponibles sur le web à l'adresse donnée en référence [121] sous forme d'une archive en format Zip. Notre prototype se nomme RAVE (Require-

ments Analysis and Verification Engine) et se compose d'un noyau faisant appel à trois modules indépendants :

Noyau – Implémenté en Prolog, le noyau propose divers mode de fonctionnement gérés par des options sur la ligne de commande. Il appelle les modules nécessaires en fonction de ce qui est demandé par l'usager.

Pré-traitement linguistique – Ce module est implémenté en Prolog. Il procède à la formalisation des propriétés des pré- et post-conditions de la base de données fournie. Il produit une base de données formelle complète contenant propriétés, actons et organisation du système en modules. Son implémentation est détaillée dans la section 4.1.2.

Vérification de cohérence – Aussi implémenté en Prolog, ce module contient les règles de vérification de cohérence. Il procède à la lecture et l'analyse de la base de données formelle, et enregistre les résultats dans un fichier texte. Son implémentation est détaillée dans la section 4.1.3.

Extraction des fonctionnalités manquantes – Implémenté en C, ce module transforme l'ensemble des actions en notation cubique, appelle le programme de calcul du complément (implémenté en C) et retranscrit les lignes du complément obtenu sous forme d'ensembles de propriétés, améliorant ainsi sa lisibilité. Son implémentation est détaillée dans la section 4.1.4.

Les exigences sont stockées dans un format relatif à leur type. Ainsi, deux formats de données sont utilisés par le prototype : semi-formel et formel. Ces derniers sont présentés dans la section 4.1.1.

4.1.1 Formats de données

Les exigences semi-formelles consistent en un ensemble d'actions dans lequel les propriétés des pré- et post-conditions sont des phrases en anglais syntaxiquement et lexicalement restreint. Aucune définition formelle n'existe pour les propriétés et

ces exigences ne sont pas exploitables par les processus d'analyse de cohérence et d'extraction des fonctionnalités manquantes.

Les actions sont regroupées dans un fichier texte dans lequel elles sont enregistrées sous forme de prédicats Prolog. Chaque prédicat contient 5 champs qui sont respectivement : le nom de l'action, la liste de ses pré-conditions, la liste de ses post-conditions, l'ensemble des sous-actions, une chaîne de caractère de documentation. Ainsi, à titre d'exemple, l'action suivante :

$$\alpha_x = \left| \begin{array}{lcl} \nu & = & \text{open} \\ \mu & = & \text{door} \\ \phi & = & \{ \text{“someone is at door”, “door is not opened”} \} \\ \psi & = & \{ \text{“door is opened”, “timer is not expired”} \} \\ \lambda^{\alpha} & = & \emptyset \\ \theta & = & \text{“opening of the door”} \end{array} \right.$$

Sera représentée comme suit :

```
action(Name, PreCds, PostCds, SubActions, Documentation):-
  Name           = open,
  PreCds         = [ 'someone is at door',
                     'door is not opened' ],
  PostCds        = [ 'door is opened'
                     'timer is not expired' ],
  SubActions     = [],
  Documentation  = 'opening of the door'.
```

Les exigences formelles, elles, consistent en un ensemble d'actions formelles dont les pré- et post-conditions sont des ensembles de propriétés exprimées sous forme prédicative. Cet ensemble s'accompagne d'un ensemble de définitions de propriétés ainsi que d'un ensemble de descriptions de modules. Propriétés, actions et modules sont enregistrés dans des fichiers différents respectivement nommés `properties.pl`, `actions.pl`, et `modules.pl`. L'exemple d'action semi-formelle précédent sera représenté formellement comme suit :

```
action(Name, PreCds, PostCds, SubActions, Documentation):-
```

```
  Name           = open,
  PreCds         = [ is(at(door), someone),
                     not(is(opened, door) ],
  PostCds        = [ is(opened, door)
                     not(is(expired, timer)) ],
  SubActions     = [],
  Documentation  = 'opening of the door'.
```

Et sera enregistrée dans le fichier `actions.pl`. Chaque propriété du système, comme `is(at(door), someone)`, sera enregistrée dans le fichier `properties.pl` et sera représentée comme suit :

```
property(Name, SubProperties, Documentation):-
  Name           = is(at(door), someone),
  SubProperties  = [],
  Documentation  = 'Someone is at door'.
```

Finalement, le regroupement de l'ensemble des propriétés et actions sera enregistré dans le fichier `modules.pl` dans le format suivant :

```
module(Name, InitialConditions, Properties, Actions,
       LTLConstraints, OPIConstraints, Documentation):-
  Name              = door,
  InitialConditions = [],
  Properties        = [ is(at(door), someone), ... ],
  Action            = [ open, ... ],
  LTLConstraints    = [],
  OPIConstraints    = [],
  Documentation     = 'Incomplete (fictive example)'.
```

Les points de suspension représentent le reste des propriétés et actions qui seraient ajoutées dans le cadre d'une spécification complète.

4.1.2 Pré-traitement linguistique

Le pré-traitement linguistique est implémenté en Prolog. Il prend en entrée une base de données semi-formelle et génère une base de données formelle sous forme des trois fichiers `properties.pl`, `actions.pl`, et `modules.pl`.

Lors de cette génération, chaque action formalisée est ajoutée à l'ensemble des actions définies. Chaque propriété non niée qui ne fait pas déjà partie de l'ensemble de propriétés définies est ajoutée à cette liste. Toute propriété niée est transformée en son équivalent non niée qui est ensuite ajouté à la liste de propriétés définies si nécessaire. Les hiérarchies d'actions et de propriétés ne posent pas de problème de traitement, quelle que soit leur profondeur.

Le noyau du module de pré-traitement linguistique lit le fichier contenant la base de données semi-formelle. Suite à cette lecture, il appelle les différents composants d'analyse pour construire la représentation formelle interne des propriétés. Une fois cette phase terminée, il procède à la génération des prédicats Prolog pour les actions, propriétés et découpage en modules et enregistre le tout dans les fichiers appropriés. L'analyse et la conversion des phrases trouvées dans les pré- et post-conditions repose sur les outils suivants :

Ensemble d'utilitaires – Utilisés pour le traitement de chaînes de caractère en Prolog,

Analyseur syntaxique – Utilisé pour découper les phrases et construire la représentation syntaxique arborescente,

Dictionnaire de verbes – Contenant les verbes reconnus par le module de traitement, leur radical, fonction, groupe, et types de nominalisation permis,

Analyseur morphologique – Implémentant les méthodes de lemmatisation et de déclinaison des verbes à partir du radical *défaut* de chaque verbe dans lesquels le temps du verbe(présent, imparfait ou futur de l'indicatif) est encodé,

Traducteur – Transformant les structures syntaxiques arborescentes internes en formes prédicatives.

Les limites de ce processus reposent sur l'ensemble de règles de l'analyseur morphologique ainsi que sur la taille du dictionnaire utilisé. L'état actuel du prototype ne permet son utilisation que sur les études de cas I et III (sections 4.2 et 4.4).

Néanmoins, un enrichissement du dictionnaire et de l'analyseur morphologique permettra son utilisation dans un cadre beaucoup moins restreint.

4.1.3 Vérification de cohérence

Le module de vérification, implémenté en Prolog, contient l'implémentation Prolog de chacune des règles décrites dans la section 3.4 du chapitre 3. Le module de vérification de cohérence charge la base d'exigences en mémoire, s'assure que chaque règle est satisfaite et génère un rapport de cohérence. Pour une règle donnée, chaque configuration violant cette dernière (contre-exemple) est signalée dans le rapport de cohérence, permettant ainsi d'avoir une liste complète des problèmes du système.

L'implémentation Prolog des règles est très directe. Chaque règle est implémentée sous forme d'une version modifiée de manière à chercher les contre-exemples. Cette implémentation exploite le mécanisme de *backtracking* au maximum. Considérons, à titre d'exemple, la règle 17 :

Règle 17. $\forall \alpha_x, \alpha_{x+1} \in \lambda^{\alpha-seq} : \nexists \pi_x \in \psi_x, \pi_{x+1} \in \phi_{x+1} \mid \chi(\pi_x, \pi_{x+1})$

Cette règle s'attarde sur les ensembles de sous-actions exécutées séquentiellement. Elle vérifie que dans un tel ensemble, dénoté $\lambda^{\alpha-seq}$, il n'existe pas deux actions qui se suivent et pour lesquelles les post-conditions de la première entrent en contradiction avec les pré-conditions de la suivante. Considérons l'action suivante :

Le code Prolog de cette règle, dans lequel des numéros de lignes ont été ajoutés pour plus de lisibilité, est donné ci-après (note : une connaissance de base de Prolog est nécessaire à la compréhension du code donné en exemple).

```
l1:  rule17:-
l2:      action(A, PA, _, seq([E|_]), _),
l3:      action(E, PE, _, _, _),
l4:      contradiction(PA, PE, [CA, CE]),
l5:      \+(found_rule17_contradiction(A, E, [CA, CE])), !,
l6:      write(found_rule17_contradiction(A, E, [CA, CE])),
l7:      asserta(found_rule17_contradiction(A, E, [CA, CE])),
l8:      rule17, !.
```

La première instance du prédicat `rule17` consiste à sélectionner les diverses actions existantes dans la base de données qui est en mémoire (lignes l2, l3) de sorte que les post-conditions de la première (A, PA) entrent en contradiction avec celles de la deuxième (E, PE) comme le stipule la ligne l4. Si une configuration satisfait la ligne l4 et qu'elle n'a pas encore été enregistrée (ligne l5), un rapport sera généré (l6), la contradiction sera enregistrée (l7) et la règle sera récursivement rappelée.

Dans le cas où aucune contradiction n'est trouvée après avoir exploré toutes les configurations possibles, l'appel du prédicat se terminera sans avoir écrit quelque ligne que ce soit dans le rapport. Dans le cas où des contradictions sont trouvées, Prolog continuera l'exploration de tous les cas de figure jusqu'à ce qu'il ne puisse plus en trouver qui permettent de satisfaire la ligne l5 (c'est à dire, qui n'aient pas déjà été trouvés).

Dans la mesure où il s'agit d'un prototype, nous n'avons pas poussé son implémentation au delà de certaines limites, le but étant de prouver que l'on pouvait implémenter les concepts théoriques, pas de construire un outil commercial. Ainsi, tout d'abord, l'analyse de cohérence est effectuée d'une manière peu optimale dû au *backtracking* et aux appels récursifs. Ensuite, l'implémentation actuelle ne permet pas le traitement d'une hiérarchie de sous-actions et sous-propriétés d'une profondeur supérieure à un. Ces limites peuvent aisément être éliminées et dépendent uniquement du statut expérimental de notre prototype.

4.1.4 Extraction des fonctionnalités manquantes

Le module d'extraction des fonctionnalités manquantes est implémenté en deux parties. Une première, en Prolog, effectue les tâches suivantes :
Pour chaque module :

- Traduction de la liste d'actions en forme cubique et enregistrement de cette forme dans un fichier,
- Exécution (via appel système) de l'outil *espresso* [122] pour effectuer une pré-réduction de la forme cubique,

- Exécution (via appel système) du programme de calcul du complément,
- Nouvelle exécution (via appel système) de l'outil *espresso* pour effectuer une pré-réduction de la forme cubique,
- Traduction des lignes du complément obtenu sous forme cubique en listes de pré-conditions et écriture de ces dernières dans un fichier rapport.

L'outil de calcul du complément est implémenté en C. Ce choix est en partie fonction du fait que les structures de données à manipuler sont des tableaux de bits, structures pour lesquelles le C est un langage bien plus approprié que Prolog. L'algorithme de calcul du complément est le suivant :

1. Pour chaque variable du premier cube :
   (a) Créer un cube-complément avec le complément de la variable dans la colonne correspondante et des *don't care* dans les autres colonnes,
2. pour chaque autre cube :
   (a) pour chaque variable du cube :
      i. Créer un cube-complément avec le complément de la variable dans la colonne correspondante et des *don't care* dans les autres colonnes,
   (b) Effectuer l'intersection du dernier ensemble cube-complément calculé avec le résultat de l'intersection précédente (ou bien celui de l'ensemble cube-complément du premier cube si aucune intersection n'a été calculée précédemment),
   (c) Effectuer une réduction à la volée de l'intersection calculée au point précédent (élimination des cubes couverts par d'autres cubes).

Similairement à la vérification de cohérence, l'état actuel du prototype ne permet pas de traiter les hiérarchies de sous-actions et sous-propriétés, quel que soit leur degré de profondeur.

4.2 Étude de cas I – Porte automatique

Notre première étude de cas porte sur un système fictif de porte automatique. Son fonctionnement consiste à s'ouvrir lorsqu'elle est fermée et que quelqu'un se trouve devant. Une fois ouverte, elle doit obligatoirement se refermer après un certain temps. Les actions d'ouverture et de fermeture sont mutuellement exclusives, c'est-à-dire que la porte ne peut s'ouvrir et se fermer en même temps.

4.2.1 Modélisation

Nous identifions deux actions : ouverture et fermeture. Leur modélisation consiste à les décrire en utilisant la représentation structurée semi-formelle tel qu'illustré par le tableau 4.1. Ce modèle est ensuite automatiquement modifié par le processus de pré-traitement linguistique. Ce dernier transforme en prédicats les phrases énonçant les pré- et post-conditions. Le résultat est illustré par le tableau 4.2.

Ouverture	
ν	open
μ	door
ϕ	{ "someone is at door", "door is not opened" }
ψ	{ "door is opened", "timer is not expired" }
λ^{α}	$\emptyset$
θ	"opening of the door"

Fermeture	
ν	close
μ	door
ϕ	{"someone is not at door", "door is opened", "timer is expired" }
ψ	{ "door is not opened" }
λ^{α}	$\emptyset$
θ	"closing of the door"

Table 4.1: Représentation semi-formelle des fonctionnalités

Comme expliqué dans la section 4.1.2, lors de cette transformation, chaque action formalisée est ajoutée à la liste des actions définies. Chaque propriété non niée qui ne fait pas déjà partie de la liste de propriétés définies est ajoutée à cette

liste. Toute propriété niée est transformée en son équivalent non niée qui est ensuite ajouté à la liste de propriétés définies si nécessaire. Le système formalisé se compose des éléments suivants :

Trois propriétés :

- $\pi_0 = [\ is(at(door), someone) \cdot door \cdot \emptyset \cdot$ "someone is at door" $]$
- $\pi_1 = [\ is(opened, door) \cdot door \cdot \emptyset \cdot$ "door is opened" $]$
- $\pi_2 = [\ is(expired, timer) \cdot door \cdot \emptyset \cdot$ "timer expires" $]$

Deux actions :

- $\alpha_0 = [\ open \cdot door \cdot \{\pi_0, \neg\pi_1\} \cdot \{\pi_1, \neg\pi_2\} \cdot \emptyset \cdot$ "opening of the door" $]$
- $\alpha_1 = [\ close \cdot door \cdot \{\neg\pi_0, \pi_1, \pi_2\} \cdot \{\neg\pi_1\} \cdot \emptyset \cdot$ "closing of the door" $]$

Propriétés	**Prédicats**	**Phrases sources**
π_0	is(at(door), someone)	"someone is at door"
π_1	is(opened, door)	"door is opened"
π_2	is(expired, timer)	"timer is expired"
$\neg\pi_0$	not(is(at(door), someone))	"someone is not at door"
$\neg\pi_1$	not(is(opened, door))	"door is not opened"
$\neg\pi_2$	not(is(expired, timer))	"timer is not expired"

Table 4.2: Représentation formelle des propriétés

Il est ensuite nécessaire d'ajouter manuellement les contraintes temporelles et déontiques, lesquelles ne peuvent être inférées automatiquement. Conformément à la description textuelle du fonctionnement de la porte, nous ajoutons :

- Une contrainte temporelle stipulant que la porte doit toujours finir par être fermée : $\tau_f(\neg\pi_1)$
- Et une contrainte déontique stipulant qu'elle ne peut s'ouvrir et se fermer en même temps : $\delta_i(\{\alpha_0, \alpha_1\}^{par})$

Ce système est ensuite analysé, d'abord par le processus de vérification de cohérence puis par celui d'extraction des fonctionnalités manquantes. Les résultats sont présentés et discutés dans la section qui suit.

4.2.2 Résultats

Étant donné la taille du système, la vérification de cohérence est exécutée de manière quasi instantanée ; moins d'une seconde sur un processeur Centrino cadencé à 1.5 GHz. Les résultats obtenus sont les suivants :

- Définitions : Pas d'erreur à signaler. Les propriétés et actions sont correctement définies,
- Énumérations : Pas d'erreur non plus,
- Redondance : Idem,
- Inter-énumérations : Pas d'erreur (le système ne comporte pas d'inter-énumérations),
- Contraintes temporelles : Satisfaite par l'action α_1,
- Contraintes déontiques : Pas de problème de la contrainte déontique.

Du point de vue de la vérification de cohérence, notre système de porte automatique ne comporte donc pas d'erreur. L'extraction des fonctionnalités manquantes quant à elle nous permet d'identifier certains manques. Le tableau 4.3 montre la représentation tabulaire des actions le complément de cette dernière.

	π_0	π_1	π_2
α_0	1	0	-
α_1	0	1	1

	π_0	π_1	π_2
β_0	-	1	0
β_1	0	0	-
β_2	1	1	-

Table 4.3: Actions et leur complément

Retraduites sous forme de pré-conditions en langue naturelle via la correspondance entre les prédicats et leurs phrases sources, les actions complémentaires énoncent que rien n'est prévu pour les cas suivants :

β_0 : "door is opened", "timer is not expired"

Que doit faire le système lorsque la porte est ouverte et que le compteur n'a pas expiré, qu'il y ait quelqu'un à la porte ou non ? Cette action manquante n'est pas vraiment un problème puisqu'il s'agit simplement de ne rien faire. Nous pouvons néanmoins enrichir le système avec l'action suivante :

- $\alpha_2 = [\ w\_close \cdot door \cdot \{\pi_1, \neg\pi_2\} \cdot \{\pi_1, \neg\pi_2\} \cdot \emptyset \cdot$ "wait to close door" $]$

β_1 : "someone is not at door", "door is not opened"

Que doit faire le système lorsque personne n'est à la porte et que celle-ci est fermée, quelle que soit l'état du compteur ? Cette action manquante n'est pas non plus vraiment un problème puisqu'il s'agit, comme pour le cas précédent, de ne rien faire. Nous pouvons ajouter l'action suivante :

- $\alpha_3 = [\ w\_open \cdot door \cdot \{\neg\pi_0, \neg\pi_1\} \cdot \{\neg\pi_0, \neg\pi_1\} \cdot \emptyset \cdot$ "wait to open door" $]$

β_2 : "someone is at door", "door is opened"

Que doit faire le système lorsque quelqu'un est à la porte, que cette dernière est ouverte, quelle que soit l'état du compteur ? Si le compteur n'a pas expiré, l'action manquante n'est pas un problème puisque la porte doit rester ouverte. Ce cas est déjà couvert par l'action $alpha_2$ ajoutée précédemment.

Par contre, si le compteur a expiré et que quelqu'un est devant la porte, cela risque effectivement de poser un problème car la porte devrait rester ouverte. En effet, l'action α_1 précise que la porte est fermée si personne ne se trouve devant, qu'elle est ouverte, et que le compteur expire. Nous raffinons donc le système en ajoutant l'action α_3 qui stipule que ce dernier doit rester dans le même état si quelqu'un est devant la porte. Une fois que la personne ne sera plus devant la porte, ce sera l'action α_1 qui sera exécutée.

- $\alpha_3 = [\ w\_close \cdot door \cdot \{\pi_0, \pi_1, \pi_2\} \cdot \{\pi_0, \pi_1, \pi_2\} \cdot \emptyset \cdot$ "wait to close door" $]$

Une nouvelle itération des processus de vérification et d'extraction des fonctionnalités manquantes ne donne aucune erreur. Ceci signifie que notre système est

à présent complet et que nous n'avons pas introduit d'erreur en ajoutant de nouvelles actions. Dans le cas contraire, nous aurions il aurait fallu le raffiner encore et passer à travers une nouvelle itération des processus de vérification et d'extraction des fonctionnalités manquantes tant et aussi longtemps que nous n'aurions pas été satisfaits des résultats.

4.3 Étude de cas II – RapidIO

Cette deuxième étude de cas porte sur l'architecture RapidIO [123] dont la spécification est actuellement développée par le *RapidIO Trade Association Technical Working Group*. Il s'agit d'une architecture développée dans le but d'apporter une solution aux besoins de haute performance associés aux systèmes de commutation de paquets reposant sur des interconnexions utilisant un faible nombre de broches.

Elle se veut, en premier lieu, être une interface intra-système permettant des communications puce-à-puce et carte-à-carte avec un débit de l'ordre de 1 Giga-octet par seconde et est destinée à être utilisée comme une norme dans de nombreuses applications à haute performance de type réseau, télécom et embarquées. Sa spécification se compose de trois couches principales qui sont :

Logique – Définissant le protocole global, les formats de paquets, ainsi que les types de transactions et l'adressage des données,

Transport – Fournissant les informations de routage des paquets d'un point à un autre,

Physique – Contenant les interfaces telles que les mécanismes de transport, le contrôle de flux, les caractéristiques électriques et la gestion des erreurs de bas niveau.

La couche physique se sub-divise en trois sous-couches qui sont :

PMA – Gérant les émissions (sérialisation) et réception (dé-sérialisation),

PCS – Effectuant l'encodage des caractères,

Comportementale – Dictant les comportements tels que le contrôle de flux et la détection d'erreurs.

Notre étude de cas porte sur les mécanismes d'initialisation des ports de communication (initialisation, synchronisation et alignement des lignes). L'architecture RapidIO prévoit 4 lignes de communication indépendantes à travers lesquelles des données dé-sérialisées peuvent être envoyées parallèlement. Ainsi, la séquence de bits 011110100110 sera découpée en trois groupes distincts 0111, 1010, et 0110, lesquels seront envoyés parallèlement les uns après les autres tel qu'illustré par la figure 4.1.

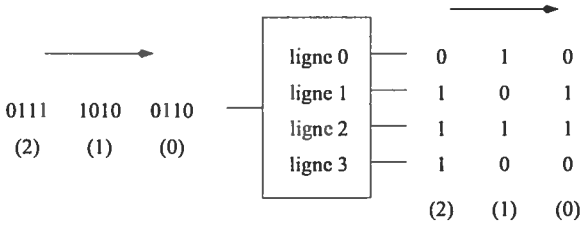

Figure 4.1: RapidIO : Dé-sérialisation

La mise en place et la maintenance d'un échange de données à travers ces lignes repose sur trois opérations qui font partie de la sous-couche comportementale et qui sont :

Initialisation – Détection d'un partenaire sur la ligne,

Synchronisation – Synchronisation au niveau des bits transmis et prise en charge des re-synchronisations lorsque nécessaire,

Alignement – Gestion de l'alignement des lignes pour la transmission de données en mode dé-sérialisé.

L'ensemble de ces opérations est regroupé au sein d'un document PDF dans lequel chaque processus est décrit de manière détaillée en anglais. Chaque état du système ainsi que les conditions de transition vers un autre état sont énumérés. La figure 4.2 illustre un exemple de paragraphe.

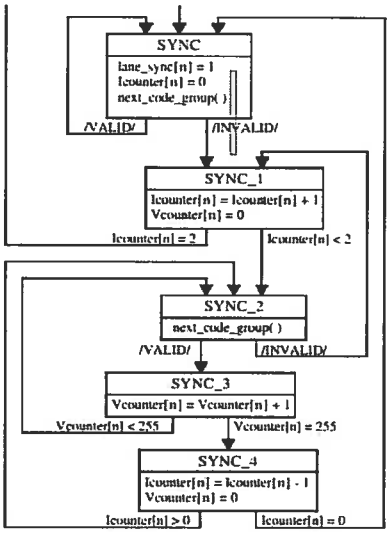

Figure 4.3: Automate

4.6.3.3 Lane Synchronization State Machine

The Lane_Synchronization state machine monitors the bit synchronization and code-group boundary alignment for a lane receiver. A port that supports only 1x mode has one Lane_Synchronization state machine. A port that supports both 1x and 4x modes has four Lane_Synchronization state machines, one for each lane (Lane_Synchronization[0] through Lane_Synchronization[3]).

The state machine determines the bit synchronization and code-group boundary alignment state of a lane receiver by monitoring the received code-groups and looking for /K28.5/s, other valid code-groups and invalid code-groups. The code-group /K28.5/ contains the "comma" bit sequence that is used to establish code-group boundary alignment. When a lane is error free, the "comma" pattern occurs only in the /K28.5/ code-group. Several counters are used to provide hysteresis so that occasional bit errors do not cause spurious lane_sync state changes.

Figure 4.2: Description textuelle (fragment)

Ces descriptions textuelles sont accompagnées d'une représentation plus formelle donnée sous forme d'un automate correspondant à ladite description. Un exemple d'un tel automate est donnée à la figure 4.3. Notre travail a consisté à modéliser l'ensemble des mécanismes d'initialisation, synchronisation, et alignement de chacune des quatre lignes.

4.3.1 Modélisation

Nous n'avions pas encore élaboré le processus de pré-traitement linguistique au moment de cette étude, aussi, nous avons directement formalisé les exigences données. Ainsi, à titre d'exemple, le code Prolog qui suit illustre la formalisation d'une fonctionnalité propre au processus d'alignement des lignes.

```
action(Name, PreConditions, PostConditions, SubActions,
       Documentation):-
  Name           = lane_alignment(transition_02),
  PreConditions  = [ not(bs_03), not(bs_04), not(bs_05),
                     not(internal('||A||')) ],
  PostConditions = [ not(bs_03), not(bs_04), not(bs_05),
                     not(lanes_aligned),
                     internal(equals(acounter, 0)),
                     called(next_column) ],
```

```
SubActions      = [],
Documentation   = 'Lane alignement - Transition 02'.
```

Nous avons étiqueté les transitions avec des numéros suivant l'ordre dans lequel nous les avons modélisées. Les états, eux, sont représentés suivant l'encodage de Huffman. Au total, ce sont 128 propriétés et 168 fonctionnalités qui ont ainsi été formalisées. Le tableau 4.4 résume les types et nombres de fonctionnalités modélisées. Le système ne contenait pas de contrainte temporelle ou déontique.

Processus	Module	Actions	Propriétés
Initialisation	Toutes lignes	17	21
Synchronisation	Ligne 0	20	20
	Ligne 1	20	20
	Ligne 2	20	20
	Ligne 3	20	20
Alignement	Toutes lignes	19	16
Activité	Ligne 0	13	5
	Ligne 1	13	5
	Ligne 2	13	5
	Ligne 3	13	5
Total		**168**	**128**

Table 4.4: Fonctionnalités modélisée

4.3.2 Résultats

La vérification de cohérence, appliquée au système complet (168 actions, 128 propriétés) prend 43 secondes sur un processeur Centrino cadencé à 1.5 GHz. Sur le même processeur, l'extraction des fonctionnalités manquantes, appliquée au système complet, ainsi qu'aux différents modules, prend moins d'une seconde pour effectuer la totalité des extractions. La vérification de cohérence nous a permis de détecter quelques cas que nous avions introduits par erreur lors de la formalisation des actions. Hormis cela, le protocole ne comportait pas d'erreur.

L'extraction des fonctionnalités manquantes a pointé quelques états du système dans lesquels aucune action n'est prévue si aucune donnée n'arrive sur les lignes.

Ces manques ne sont pas critiques dans la mesure où dans les états identifiés, le système n'est pas censé faire quoi que ce soit et doit rester dans le même état en attendant que des données arrivent. Ajouter des actions pour combler ces manques reviendrait simplement à décrire une boucle sur le même état, ce qui n'est pas nécessaire dans ce cas dans la mesure où toutes les transitions sont conditionnelles à des changements sur les lignes de communication.

Les temps d'analyse obtenus confirment quant à eux l'implémentation non-optimale du prototype au niveau de la vérification de cohérence. Pour ce qui est de l'extraction des exigences manquantes, le nombre de propriétés par module (21 dans le cas du processus d'initialisation) est encore suffisamment petit pour permettre un temps de calcul raisonnable.

4.4 Étude de cas III – Dispositif *cross-connect*

Notre dernière étude de cas porte sur un ensemble d'exigences fourni par PMC-Sierra. Cet ensemble est une sous-partie des exigences d'un dispositif de connexion croisée (*cross-connect*) [124]. Un tel dispositif est un système de commutation de paquets permettant, à partir de diverses sources données, de ré-acheminer les données d'une (ou plusieurs) source(s) idéale(s).

Notre étude porte spécifiquement sur les fonctionnalités de routage du dispositif *cross-connect*. Ces dernières sont regroupées dans un document *Word*. Elles sont présentées sous forme d'une liste dans laquelle chaque item est décrit en anglais et est appelé *requirement*.

Requirement #	**Description**
r3.22	A synchronization request with no synchronization timer running shall force the start of the synchronization timer and immediate sending of an acknowledgement.
r3.23	When the TGV timer expires and no input is configured, the system should switch to the last known source.
...	...

Table 4.5: Liste des fonctionnalités données

Le tableau 4.5 montre un exemple représentatif dans lequel, pour des raisons de confidentialité, les noms et descriptions sont fictifs. Au total, 27 fonctionnalités sont ainsi décrites. En supplément, cet ensemble s'accompagne d'un tableau récapitulatif dans lequel les colonnes représentent les conditions de déclenchement de la fonctionnalité (les pré-conditions) tandis que les lignes représentent les fonctionnalités et le résultat de l'action (les post-conditions). Somme toute, cette représentation correspond exactement à une somme de produits représentée sous forme cubique.

4.4.1 Modélisation

La modélisation de ces fonctionnalités en Prolog est suit un processus similaire à la modélisation de celles de l'étude de cas précédente. Ainsi, la fonctionnalité r3.22 donnée en exemple dans la section 4.3 est représentée comme suit :

```
action(Name, PreCds, PostCds, SubActions, Documentation):-
   Name          = r3-22,
   PreCds        = [ 'there is a synchronization request'
                     'synchronization timer is not running' ],
   PostCds       = [ 'synchronization timer is running',
                      'acknowledgement is sent' ],
   SubActions    = [],
   Documentation = 'A synchronization request with no
                    synchronization timer running shall
                    force the start of the synchronization
                    timer and immediate sending of an
                    acknowledgement.'
```

Ce qui donne, après le passage du processus de traitement linguistique :

```
action(Name, PreCds, PostCds, SubActions, Documentation):-
   Name          = r3-22,
   PreCds        = [ not(request(synchronization)),
                     not(is(running, timer(synchronization))) ],
   PostCds       = [ is(running, timer(synchronization))),
                      is(sent, acknowledgement) ],
   SubActions    = [],
   Documentation = 'A synchronization request with no
```

```
                synchronization timer running shall
                force the start of the synchronization
                timer and immediate sending of an
                acknowledgement.'
```

L'ensemble des fonctionnalités appartient au même module et le modèle ne contient aucune contrainte temporelle ou déontique. Toujours pour des raisons de confidentialité, l'ensemble des fonctionnalités ainsi décrites ne peut malheureusement pas être donné en exemple mais leur regroupement au sein du même module suit le modèle donné dans la section 4.1 :

```
module(Name, InitialConditions, Properties, Actions,
       LTLConstraints, OPIConstraints, Documentation):-
  Name              = cross-connect,
  InitialConditions = [],
  Properties        = [ pi0, pi1, pi2, pi3, pi4, pi5, pi6, pi7,
                        pi8, pi9, pi10, pi11, pi12, pi13, pi14,
                        pi15 ],
  Actions           = [ r3-18, r3-19, r3-20, r3-21, r3-22,
                        r3-23, r3-24, r3-25, r3-26, r3-27,
                        r3-28, r3-29, r3-30, r3-31, r3-32,
                        r3-33, r3-34, r3-35, r3-36 ],
  LTLConstraints    = [],
  OPIConstraints    = [],
  Documentation     = 'Cross-Connect PMC-Sierra'.
```

4.4.2 Résultats

La vérification de cohérence nécessite 5.4 secondes sur un processeur Centrino cadencé à 1.5 GHz. Les résultats obtenus sont les suivants :

- Définitions : Pas d'erreur à signaler. Les propriétés et actions sont correctement définies,
- Énumérations : Pas d'erreur non plus,
- Redondance : Idem,

- Inter-énumérations : Pas d'erreur (le système ne comporte pas d'inter-énumérations),
- Contraintes temporelles : Pas de contraintes temporelles,
- Contraintes déontiques : Pas de contraintes déontiques.

La description des fonctionnalités ne comporte pas de problèmes de cohérence. Ceci peut être expliqué par le fait que les exigences données étaient celles d'un système déjà implanté par PMC-Sierra. Ces dernières avaient donc été analysées et raffinées un certain nombre de fois avant que nous ne les obtenions.

L'extraction des fonctionnalités manquantes nous a permis d'identifier un certain nombre de manques. Tout d'abord, le tableau 4.6 illustre la représentation cubique de l'ensemble des fonctionnalités. Toujours pour des raisons de confidentialité, les noms de fonctionnalités et propriétés sont respectivement remplacés par α et π. Le calcul du complément de ce tableau nécessite 2.5 secondes sur un processeur Centrino cadencé à 1.5 GHz. Le résultat, après post-réduction via *espresso* est illustré par le tableau 4.7.

Nous avons rapporté ces résultats à notre contact à PMC-Sierra et les avons analysés avec lui. Celui-ci nous a confirmé que ces derniers étaient pertinents d'autant plus que l'ensemble d'exigences qu'il nous avait donné était incomplet.

L'ensemble de fonctionnalités manquantes que nous avions extrait correspondait à l'ensemble de fonctionnalités que PMC-Sierra ne nous avait pas donné, présenté par le tableau 4.8. De ce fait, toutes les exigences manquantes identifiées ont pu être expliquées et, par conséquent, aucun problème réel n'a été identifié dans le dispositif implémentant ces exigences.

Une fois en possession de l'ensemble de fonctionnalités qui ne nous avaient pas été données, nous avons identifié les lignes du complément qui couvraient ces dernières. Un certain nombre de fonctionnalités sont couvertes uniquement par le complément tandis que d'autres sont couvertes par l'amalgame du complément et d'une ou plusieurs fonctionnalités déjà existantes.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_1	-	-	-	-	-	-	1	1	-	0	0	0	1	-	0	-
α_2	-	-	-	-	-	-	1	1	-	0	0	0	1	-	1	0
α_3	-	-	-	-	-	-	1	1	-	0	0	1	-	-	0	-
α_4	-	-	-	-	-	-	1	1	-	0	0	1	-	-	1	0
α_5	-	-	-	-	-	-	1	1	0	0	0	-	0	-	1	-
α_6	-	-	-	-	-	-	1	1	1	0	0	-	0	-	1	-
α_7	-	-	-	-	-	-	1	1	1	0	0	0	1	-	1	-
α_8	-	-	-	0	-	-	-	-	-	-	1	-	-	-	-	-
α_9	-	-	-	0	-	-	-	-	-	1	-	-	-	-	-	-
α_{10}	-	-	-	1	-	-	-	-	-	-	1	-	-	-	-	-
α_{11}	-	-	-	1	-	-	-	-	-	1	-	-	-	-	-	-
α_{12}	0	-	-	-	-	-	1	0	-	0	0	-	0	-	-	-
α_{13}	0	-	-	-	-	-	1	0	-	0	0	0	1	1	-	-
α_{14}	0	-	-	-	-	-	1	1	-	0	0	-	0	-	0	-
α_{15}	0	-	-	0	-	-	0	-	-	0	0	0	-	-	-	-
α_{16}	0	-	-	1	0	-	0	-	-	0	0	0	-	-	-	-
α_{17}	0	-	-	1	1	1	0	-	-	0	0	0	-	-	-	-
α_{18}	1	0	-	-	-	-	1	0	-	0	0	-	0	-	-	-
α_{19}	1	0	-	-	-	-	1	0	-	0	0	0	1	1	-	-
α_{20}	1	0	-	-	-	-	1	1	-	0	0	-	0	-	0	-
α_{21}	1	0	-	1	0	-	0	-	-	0	0	0	-	-	-	-
α_{22}	1	0	-	1	1	1	0	-	-	0	0	0	-	-	-	-
α_{23}	1	1	1	-	-	-	1	0	-	0	0	-	0	-	-	-
α_{24}	1	1	1	-	-	-	1	0	-	0	0	0	1	-	-	-
α_{25}	1	1	1	-	-	-	1	1	-	0	0	-	0	-	0	-
α_{26}	1	1	1	0	-	-	0	-	-	0	0	0	-	-	-	-
α_{27}	1	1	1	1	-	-	0	-	-	0	0	0	-	-	-	-

Table 4.6: Liste des fonctionnalités données

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
β_1	-	-	-	-	-	-	-	-	-	0	0	1	1	-	1	1
β_2	-	-	-	-	-	-	-	0	-	0	0	1	1	-	-	-
β_3	-	-	-	-	-	-	0	-	-	0	0	1	-	-	-	-
β_4	-	-	-	-	-	-	1	1	0	0	0	-	1	-	1	1
β_5	-	0	-	-	-	-	1	0	-	0	0	-	1	0	-	-
β_6	-	0	-	1	1	0	0	-	-	0	0	-	-	-	-	-
β_7	0	-	-	-	-	-	1	0	-	0	0	-	1	0	-	-
β_8	0	-	-	1	1	0	0	-	-	0	0	-	-	-	-	-
β_9	1	0	-	0	-	-	0	-	-	0	0	-	-	-	-	-
β_{10}	1	1	0	-	-	-	-	-	-	0	0	0	0	-	0	-
β_{11}	1	1	0	-	-	-	-	0	-	0	0	-	-	-	-	-
β_{12}	1	1	0	-	-	-	0	-	-	0	0	-	-	-	-	-

Table 4.7: Complément de la liste de fonctionnalités données

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{28}	-	-	-	-	-	-	0	-	-	0	0	1	-	-	-	-
α_{29}	-	-	-	-	-	-	1	0	-	0	0	1	-	-	-	-
α_{30}	-	-	-	-	-	-	1	1	-	0	0	1	-	-	1	1
α_{31}	-	-	-	-	-	-	1	1	0	0	0	0	1	-	1	1
α_{32}	-	-	-	-	-	-	1	1	1	0	0	0	1	-	0	-
α_{33}	0	-	-	-	-	-	1	0	-	0	0	0	1	0	-	-
α_{34}	0	-	-	1	1	0	0	-	-	0	0	0	-	-	-	-
α_{35}	1	0	-	-	-	-	1	0	-	0	0	0	1	0	-	-
α_{36}	1	0	-	0	-	-	0	-	-	0	0	0	-	-	-	-
α_{37}	1	0	-	1	1	0	0	-	-	0	0	0	-	-	-	-
α_{38}	1	1	0	-	-	-	1	0	-	0	0	-	0	-	-	-
α_{39}	1	1	0	-	-	-	1	0	-	0	0	0	1	-	-	-
α_{40}	1	1	0	-	-	-	1	1	-	0	0	-	0	-	0	-
α_{41}	1	1	0	0	-	-	0	-	-	0	0	0	-	-	-	-
α_{42}	1	1	0	1	-	-	0	-	-	0	0	0	-	-	-	-

Table 4.8: Liste des fonctionnalités non données par PMC-Sierra

4.4.2.1 Fonctionnalités manquantes couvertes par le complément

Comme le montre le tableau suivant, l'action β_3 du complément correspond en tout point à la fonctionnalité manquante α_{28} :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{28}	-	-	-	-	-	-	0	-	-	0	0	1	-	-	-	-
β_3	-	-	-	-	-	-	0	-	-	0	0	1	-	-	-	-

Ci-dessous, nous constatons que β_4 couvre α_{31} en généralisant au niveau de la propriété π_{12}. Ceci ne signifie pas que le complément ne tient pas considération de la propriété π_{12} mais qu'il représente les deux valeurs que cette dernière peut prendre. Dans la modélisation du système, PMC-Sierra a uniquement considéré le cas dans lequel π_{12} est niée, le cas contraire n'étant pas pertinent.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{31}	-	-	-	-	-	-	1	1	0	0	0	0	1	-	1	1
β_4	-	-	-	-	-	-	1	1	0	0	0	-	1	-	1	1

Similairement pour β_5 qui couvre α_{35} avec une généralisation au niveau de π_1 et π_{12} :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{35}	1	0	-	-	-	-	1	0	-	0	0	0	1	0	-	-
β_5	-	0	-	-	-	-	1	0	-	0	0	-	1	0	-	-

L'action β_6 couvre α_{37} avec une généralisation au niveau de π_{12} uniquement :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{37}	1	0	-	1	1	0	0	-	-	0	0	0	-	-	-	-
β_6	-	0	-	1	1	0	0	-	-	0	0	-	-	-	-	-

De même pour β_7 qui couvre α_{33} avec une généralisation au niveau de π_{12} unique-

ment :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{33}	0	-	-	-	-	-	1	0	-	0	0	0	1	0	-	-
β_7	0	-	-	-	-	-	1	0	-	0	0	-	1	0	-	-

Même généralisation pour β_8 couvrant α_{34} :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{34}	0	-	-	1	1	0	0	-	-	0	0	0	-	-	-	-
β_8	0	-	-	1	1	0	0	-	-	0	0	-	-	-	-	-

Et encore pour β_9 et α_{36} :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{36}	1	0	-	0	-	-	0	-	-	0	0	0	-	-	-	-
β_9	1	0	-	0	-	-	0	-	-	0	0	-	-	-	-	-

L'action β_{11} couvre les actions manquantes α_{38} et α_{39} avec une généralisation sur π_7 et π_{13} pour α_{38} ainsi que pour α_{39} pour laquelle nous avons en prime une généralisation sur π_{12}. Nous constatons que les deux valeurs de π_{13} sont respectivement couvertes par α_{38} et α_{39}.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{38}	1	1	0	-	-	-	1	0	-	0	0	-	0	-	-	-
α_{39}	1	1	0	-	-	-	1	0	-	0	0	0	1	-	-	-
β_{11}	1	1	0	-	-	-	-	0	-	0	0	-	-	-	-	-

Similairement, β_{12} couvre α_{41} et α_{42} avec une généralisation sur π_4 et π_{12} pour les deux. Nous constatons que les deux valeurs de π_4 sont respectivement couvertes par α_{41} et α_{42}.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{41}	1	1	0	0	-	-	0	-	-	0	0	0	-	-	-	-
α_{42}	1	1	0	1	-	-	0	-	-	0	0	0	-	-	-	-
β_{12}	1	1	0	-	-	-	0	-	-	0	0	-	-	-	-	-

4.4.2.2 Fonctionnalités manquantes couvertes par amalgame

L'action α_{30} est couverte par l'amalgame de β_1 avec α_5 et α_6. Nous observons une généralisation de β_1 sur π_7 et π_7 ainsi qu'une spécialisation sur π_{13}. Cette spécialisation est due au fait que les actions α_5 et α_6 couvrent déjà le cas dans lequel π_{13} est niée. De ce fait, le complément illustre simplement le cas dans lequel π_{13} n'est pas niée. En réalité, l'action manquante α_{30} ne considère pas cette propriété comme pertinente.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{30}	-	-	-	-	-	-	1	1	-	0	0	1	-	-	1	1
α_5	-	-	-	-	-	-	1	1	0	0	0	-	0	-	1	-
α_6	-	-	-	-	-	-	1	1	1	0	0	-	0	-	1	-
β_1	-	-	-	-	-	-	-	-	-	0	0	1	1	-	1	1

Similairement au cas précédent, α_{29} est couverte par l'amalgame de β_2 avec α_{18}. Nous observons une spécialisation de β_2 sur π_{13}, due au fait que le cas où cette propriété est niée est couvert par α_{18}. L'action α_{29} quant à elle ne considère pas cette propriété comme pertinente.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{29}	-	-	-	-	-	-	1	0	-	0	0	1	-	-	-	-
α_{18}	1	0	-	-	-	-	1	0	-	0	0	-	0	-	-	-
β_2	-	-	-	-	-	-	-	0	-	0	0	1	1	-	-	-

Même principe que précédemment avec la combinaison de β_{10} et α_3 qui couvre l'action α_{40}, laquelle considère pas la négation de π_{12} comme étant pertinente :

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{40}	1	1	0	-	-	-	1	1	-	0	0	-	0	-	0	-
α_3	-	-	-	-	-	-	1	1	-	0	0	1	-	-	0	-
β_{10}	1	1	0	-	-	-	-	-	-	0	0	0	0	-	0	-

Dernier cas, l'action manquante α_{32} n'est pas détectée comme étant manquante. En effet, cette dernière est couverte par α_1 qui montre une généralisation sur π_9. Il se trouve que α_{32} est un cas particulier qui ne pouvait pas être détecté par le processus d'extraction des exigences manquantes.

	π_1	π_2	π_3	π_4	π_5	π_6	π_7	π_8	π_9	π_{10}	π_{11}	π_{12}	π_{13}	π_{14}	π_{15}	π_{16}
α_{32}	-	-	-	-	-	-	1	1	1	0	0	0	1	-	0	-
α_1	-	-	-	-	-	-	1	1	-	0	0	0	1	-	0	-

Chaque fonctionnalité manquante extraite couvre une, si ce n'est deux, des fonctionnalités qui n'avaient pas été données par PMC-Sierra. Seulement une fonctionnalité n'a pas été détectée, laquelle était une spécialisation d'une fonctionnalité déjà existante. Le tableau 4.9 donne un résumé statistique des résultats obtenus.

Nous constatons que 93.3 % des fonctionnalités manquantes ont été pour le moins partiellement détectées, avec 73.3 % l'ayant été totalement. Le taux de généralisation pour les fonctionnalités totalement détectées n'est que de 8.2 %. Ce taux correspond au pourcentage de propriétés pour lesquelles le complément est un "don't care". Le résultat obtenu signifie qu'en moyenne, une fonctionnalité extraite correspond à 91.8 % à la fonctionnalité manquante qu'elle représente, c'est-à-dire que les concepteurs doivent raffiner, en moyenne, moins de 10 % de chaque fonctionnalité extraite.

Cas	Nombre	Fraction	Généralisation	Spécialisation
α totalement couvertes	11	73.3 %	8.2 %	0 %
α partiellement couvertes	3	20 %	0 %	1.3 %
α non couvertes	1	6.7 %	NA	NA

Table 4.9: Résumé statistique

Le taux de généralisation oscille entre 0 % et 12.5 %. D'autre part, le taux de spécialisation pour les fonctionnalités partiellement couvertes n'est que de 1.3 %. Ceci signifie que, même dans un cas de couverture partielle, plus de 98 % de la fonctionnalité est couverte. De tels résultats confirment la pertinence du processus d'extraction des fonctionnalités manquantes.

4.5 Résumé

Dans ce chapitre, nous avons présenté l'implémentation des concepts théoriques énoncés au chapitre 3. Cette implémentation consiste en un prototype organisé en trois modules : pré-traitement linguistique, vérification de cohérence, et extraction des exigences manquantes. Le fonctionnement de notre implémentation est appuyé par trois études de cas.

La première étude de cas présente la formalisation et l'analyse d'un système de contrôleur de porte. Cette étude permet d'illustrer le fonctionnement du prototype et les résultats qui peuvent être obtenus par les processus d'analyse de cohérence et d'extraction des exigences manquantes.

Notre deuxième étude porte sur un ensemble d'exigences d'envergure industrielle tirés d'une partie de la spécification de l'architecture RapidIO. Celle-ci illustre les résultats et temps de calcul pouvant être obtenus sur un nombre important d'exigences.

Notre dernière étude de cas porte sur un ensemble d'exigences industrielles fournies par PMC-Sierra. Il s'agit d'un sous-ensemble des fonctionnalités de routage d'un dispositif *cross-connect*. Cette étude prouve l'utilité de notre méthodologie sur des exigences industrielles. Une analyse en profondeur des résultats est donnée.

L'implémentation d'un prototype viable ainsi que les résultats obtenus avec les trois études de cas présentées, appuient la pertinence des concepts théoriques énoncés au chapitre 3 et leur utilité dans le cadre de formalisation et d'analyse de spécifications concrètes.

CHAPITRE 5

TOLÉRANCE AUX PANNES – THÉORIE

Ce chapitre présente les fondements de notre méthodologie de vérification formelle des techniques de tolérance aux pannes. Nous ne considérons pas la vérification des techniques portant sur la vérification des données. Les techniques ciblées sont uniquement celles dédiées à la détection d'erreurs causées par des SEU (*Single Event Upset*) [7] et altérant le flux de contrôle. Plus particulièrement, nous ciblons l'ensemble de celles qui sont dites "sans mémoire", c'est-à-dire ne tenant compte que du bloc source et du bloc destination lors d'un transfert de contrôle.

Notre méthodologie se veut être un environnement de vérification automatisé permettant aux concepteurs de telles techniques de s'assurer rapidement de leur fiabilité pour un ensemble de programmes génériques[1] de taille réduite, et de les améliorer, puis de les vérifier à nouveau, si nécessaire. Cette méthodologie doit être vue comme complémentaire des techniques (matérielles, logicielles, et simulation) d'injection de pannes couramment utilisées. Il ne s'agit en aucun cas d'une alternative à ces dernières, mais bien d'un complément destiné à être greffé à la racine du cycle de vérification des techniques de tolérance aux pannes.

Nous fournissons un ensemble de programmes génériques de taille réduite ainsi qu'un langage pseudo-assembleur, dans lequel ces programmes sont implémentés, suffisamment puissant pour permettre l'implémentation des techniques à vérifier au sein des programmes fournis. En outre, nous fournissons un compilateur générant, à partir d'un programme en pseudo-assembleur, un modèle PROMELA [66] pour le *model-checker* SPIN [66], lequel est alors utilisé pour effectuer la vérification formelle du modèle PROMELA suivant un ensemble de formules LTL [20] spécifiques dédiées à la détection des problèmes de fiabilité de la technique implémentée.

[1] Par programme générique, nous entendons un programme implémentant l'ensemble des types de transferts de contrôle spécifiques à une classe de programmes. Nous revenons sur cette définition dans la section 5.3.

Les étapes de traduction, vérification, extraction des résultats et génération des rapports sont conduites automatiquement par un ensemble de scripts.

L'avantage d'une approche basée sur la vérification formelle par rapport aux approches basées sur les injections de pannes est que la vérification formelle permet d'effectuer la vérification complète des programmes donnés, c'est-à-dire de capturer toutes les conséquences des SEU (*Single Event Upset*) et d'évaluer la fiabilité de la technique implémentée pour chacune d'entre elles. Bien évidemment, dans la mesure où nous travaillons avec un ensemble de graphes de taille réduite, nous ne sommes pas sujet au problème d'explosion d'états.

Néanmoins étant donné la taille de nos programmes génériques, il est aussi possible que la liste des problèmes de fiabilité capturés ne soit qu'un fragment de la liste de problèmes réels. D'un autre coté, l'ensemble de programmes génériques que nous fournissons n'est pas fixe ; libre aux concepteurs de s'en inspirer pour en créer de nouveaux implémentant certaines caractéristiques types spécifiques à leur application. D'autre part, même si tous les problèmes ne sont pas capturés, la rapidité de détection de ces derniers est telle que notre approche permet d'améliorer grandement les techniques vérifiées avant de passer à l'utilisation de méthodes traditionnelles plus coûteuses, ne serait-ce qu'en termes de temps.

Les résultats expérimentaux obtenus, présentés dans le chapitre 6, ainsi que nos publications [13] [14] [15] confirment la pertinence de notre approche.

Ce chapitre s'articule autour de quatre sections. En premier, nous présentons les détails du flux de vérification que nous proposons. Suit la présentation de l'ensemble de programmes génériques. La section suivante détaille notre langage pseudo-assembleur ainsi que les mécanismes de traduction en PROMELA. Enfin, le modèle de panne et les formules LTL dédiées à la vérification formelles sont présentés. Une dernière section résume les contributions de ce chapitre.

5.1 Méthodologie

Notre méthodologie est illustrée par la figure 5.1. Il s'agit d'un cycle d'implémentation-vérification itératif destiné à être placé avant les méthodes d'injection de pannes dans le cycle de vérification.

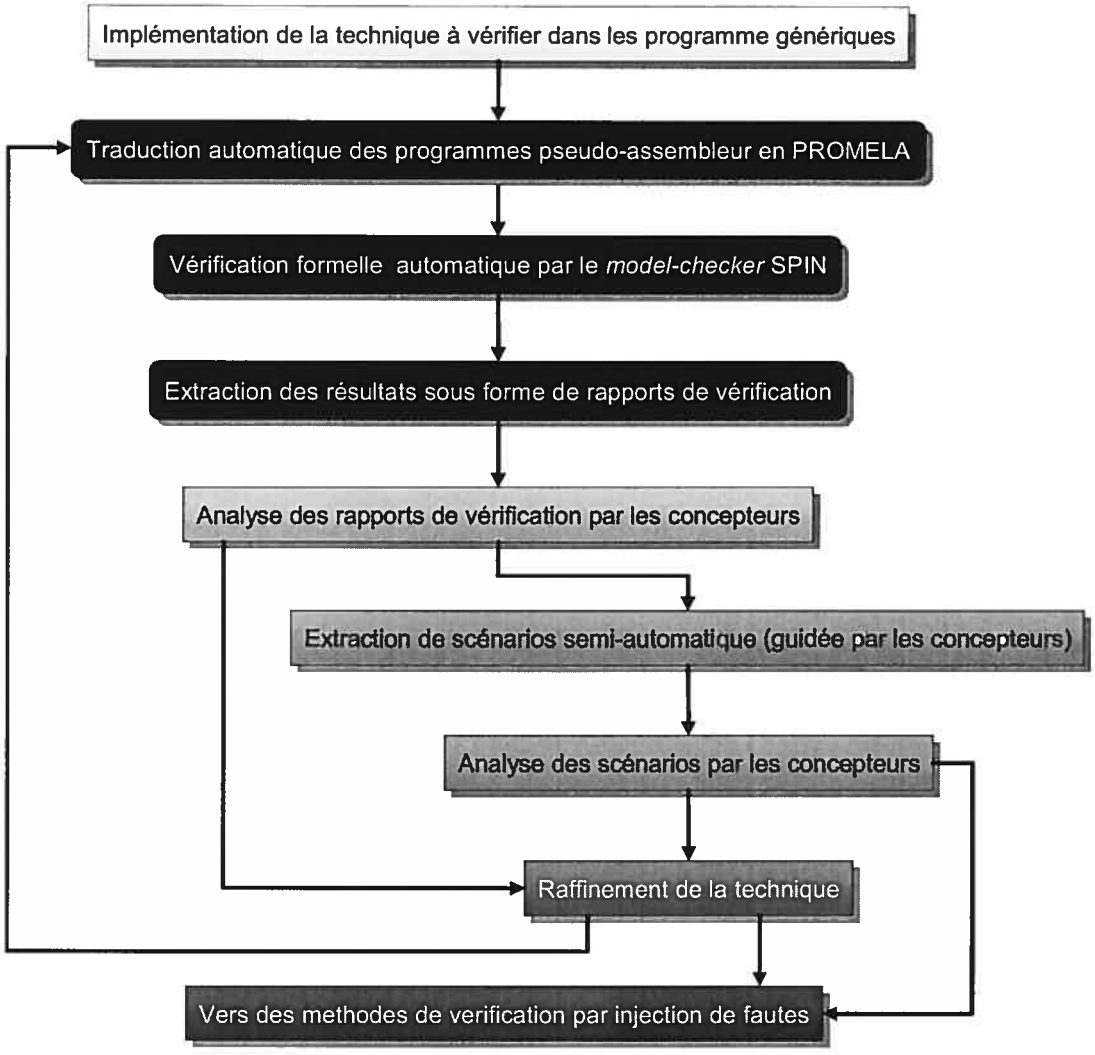

Figure 5.1: Méthodologie

Implémentation de la technique – Cette phase consiste en l'ajout, dans les programmes génériques en pseudo-assembleur, des instructions relatives à la technique de tolérance aux pannes devant être analysée. Cette phase est

manuelle et peut être effectuée très rapidement par tout concepteur connaissant bien la technique à implémenter et ayant des rudiments de langage assembleur. Le langage pseudo-assembleur est présenté dans la section 5.5.

Traduction en PROMELA – La phase de traduction en PROMELA est automatique. Elle consiste à effectuer une analyse lexicale du programme pseudo-assembleur durant laquelle la génération du code PROMELA est effectuée à la volée. La section 5.5 détaille la phase de traduction et ses résultats.

Vérification formelle – La phase de vérification formelle est automatique. Un ensemble de scripts se charge d'effectuer la compilation et la vérification du modèle PROMELA en utilisant SPIN. Un script conduit automatiquement les vérifications successives des différents programmes génériques. Cette phase est détaillée dans la section 5.6.

Extraction et analyse des résultats – Une fois l'ensemble des vérifications effectué, un script procède automatiquement à l'extraction des résultats et les présente dans un format adéquat. Les différents taux d'erreurs sont donnés et le concepteur peut, en fonction de ces résultats, décider a) de se diriger vers des méthodes de vérification par injection de pannes, ou b) de procéder à une extraction de scénarii d'erreurs.

Extraction et analyse des scénarii – L'extraction des scénarii, quant à elle, est semi-automatique : le *model-checker* SPIN doit être utilisé manuellement pour procéder à l'extraction des scénarii, lesquels sont produits automatiquement par SPIN. Les scénarii sont donnés sous forme de MSC (*Message Sequence Charts*) [125], un format graphique dont un exemple est donné dans le chapitre 6. La lecture et l'analyse de ces scénarii permettent au concepteur de comprendre les fuites[2] de la technique. Ce dernier peut alors choisir de

[2]Par *fuites*, nous faisons référence aux altération du flux de contrôle qui ne sont pas détectées par une technique de tolérance aux pannes.

raffiner la technique et de relancer une itération de vérification, ou bien de passer à l'utilisation de méthodes d'injection de pannes.

Notre méthodologie se base sur le principe "diviser pour mieux régner", à savoir qu'au lieu de vérifier les techniques de tolérance aux pannes sur des programmes comportant un nombre d'instructions important, nous utilisons des programmes génériques de taille réduite, chacun permettant de capturer les erreurs relatives à certains types de transfert de contrôle. Les fondements sur lesquels reposent ce principe de division sont présentés dans la section 5.2.

Les erreurs sur le flux de contrôle sont générées durant la vérification formelle par un mécanisme permettant de représenter toutes les conséquences possibles des SEU sur le flux de contrôle. Les principes de ce mécanisme sont présentés dans la section 5.3 et son fonctionnement est détaillé dans la section 5.6.

5.2 Abstractions

Rappelons tout d'abord que les techniques ciblées sont uniquement celles dédiées au contrôle du flux de contrôle ne tenant pas compte des données. Plus particulièrement, l'ensemble de celles qui sont dites "sans mémoire", c'est-à-dire, ne tenant compte que du bloc source et du bloc destination lors d'un transfert de contrôle. L'implémentation de ces techniques repose sur quelques principes d'abstraction présentés dans cette section.

Bloc de base – Un bloc de base, est une séquence d'instructions linéaire, c'est-à-dire ne contenant pas d'instruction de transfert. La figure 5.2 illustre un programme fictif écrit en assembleur RISC au sein duquel nous pouvons identifier trois blocs de base, ainsi que trois instructions de transfert.

Bloc – Les blocs sont constitué d'un bloc de base suivi d'une instruction de transfert quelconque vers une autre série d'instructions (un autre bloc). La figure 5.3 illustre le premier bloc du programme présenté à la figure 5.2. Il s'agit

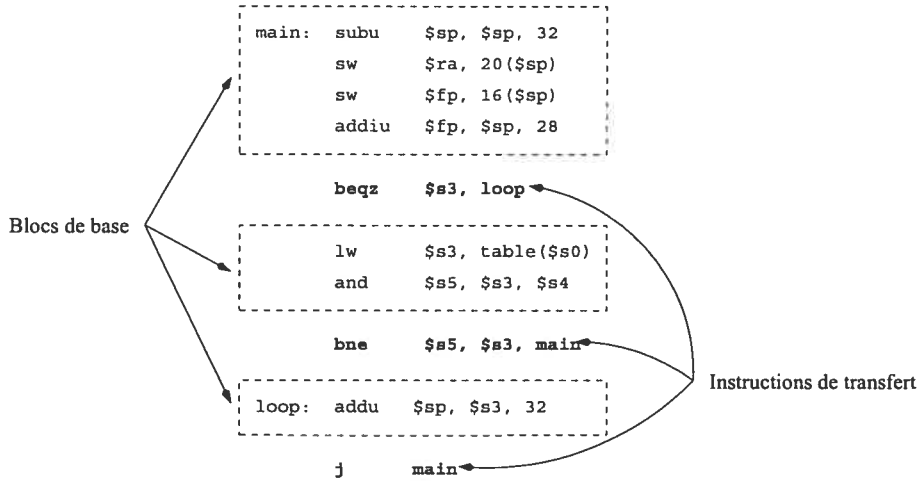

Figure 5.2: Blocs de base

du regroupement du bloc de base et de l'instruction de transfert qui suit le bloc dont il est question.

```
main: subu   $sp, $sp, 32
      sw     $ra, 20($sp)
      sw     $fp, 16($sp)
      addiu  $fp, $sp, 28
      beqz   $s3, loop
```

Figure 5.3: Bloc

Entrée/Sortie – L'entrée et la sortie d'un bloc sont respectivement la première et la dernière instruction à devoir être exécutées. Un transfert de contrôle a lieu entre la dernière instruction du bloc source et la première instruction du bloc destination. En guise d'exemple, dans le programme présenté à la figure 5.2, le transfert du premier au dernier bloc est effectué de l'instruction "`beqz $s3, loop`" du premier bloc à l'instruction "`addu $sp, $s3, 32`" du dernier bloc.

La définition d'un programme générique consiste donc en un ensemble de blocs suivant le sens défini préalablement ainsi que de transferts de contrôle de différents types. En outre, dans la mesure où les techniques ciblées ne tiennent pas compte des

erreurs pouvant altérer les données mais focalisent uniquement sur les transferts, nous effectuons les abstractions supplémentaires suivantes :

- L'ensemble des instructions d'un bloc qui portent sur les données sont remplacées par une seule instruction NOP,
- Les variables utilisées dans les tests conditionnels des transferts sont remplacées par des variables dédiées dont les valeurs seront gérées par le *model-checker* durant la vérification,

```
1  main:  ; Inst. tech. tol. fautes
          NOP
          ; Inst. tech. tol. fautes
          beqz    $var_c0 loop

2         ; Inst. tech. tol. fautes
          NOP
          ; Inst. tech. tol. fautes
          beqz    $var_c1 main

3  loop:  ; Inst. tech. tol. fautes
          NOP
          ; Inst. tech. tol. fautes
          j       main
```

Figure 5.4: Décomposition

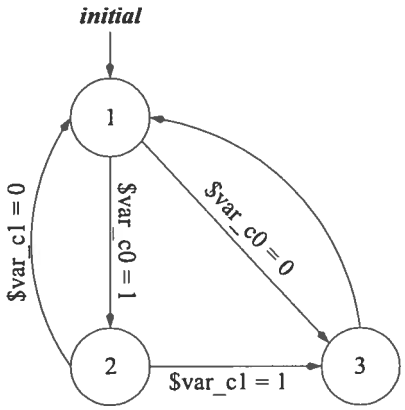

Figure 5.5: Abstraction

Les endroits propices à l'ajout des instructions des techniques de tolérance aux pannes, à savoir après et avant l'instruction NOP, seront comblés lors de l'implémentation des techniques. La figure 5.4 illustre le résultat de l'abstraction du programme fictif présenté précédemment. Nous retrouvons les trois blocs, lesquels ne contiennent plus qu'une instruction NOP encadrée par deux commentaires et suivie d'une instruction de transfert qui est conditionnelle à une variable dans le cas des blocs 1 et 2, et inconditionnelle dans le cas du bloc 3.

Afin de faciliter la perception et l'analyse du flux de contrôle d'un programme, nous représentons ce dernier sous forme d'un graphe orienté dont les noeuds représentent

les blocs, les arcs représentent les transferts de contrôle et leurs étiquettes les conditions de ces transferts. La figure 5.5 illustre la représentation graphique du programme donné à la figure 5.4. Nous retrouvons les transferts conditionnels des blocs 1 et 2 ainsi que le transfert inconditionnel du bloc 3. L'emploi d'un graphe permet une visualisation rapide des types de transferts ainsi que de l'organisation du flux de transferts de contrôle dans le programme.

La section 5.3 qui suit présente le modèle de panne sur lequel repose la génération de toutes les conséquences des SEU sur le flux de contrôle.

5.3 Modèle de panne

Considérons les deux blocs de la figure 5.6 liés par un transfert inconditionnel du bloc A au bloc B. Ces derniers sont composés d'une série d'instructions linéaire. Un transfert valide consiste en : a) un transfert inconditionnel, b) un transfert conditionnel, c) un appel ou retour de fonction partant de la dernière instruction du bloc courant (bloc source) et menant à la première instruction du prochain bloc auquel le contrôle doit être transféré (bloc destination). La figure 5.7 illustre le transfert inconditionnel du bloc A au bloc B.

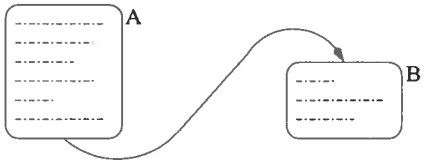

Figure 5.6: Blocs d'instructions

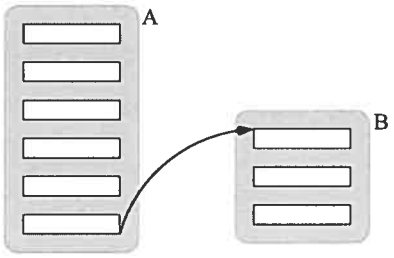

Figure 5.7: Transferts valides

Un SEU (*Single Event Upset*), appelé aussi *bit-flip* est l'inversion de la valeur d'un bit d'un registre ou d'un composant mémoire. Les SEU peuvent ainsi affecter les données qui sont en mémoire ainsi que les registres de contrôle tels que le PC (*Program Counter*).

De ce fait, les conséquences d'un SEU peuvent résulter aussi bien en une

altération du flux d'exécution que des données. Nous nous intéressons ici uniquement au flux d'exécution. De multiples raisons peuvent justifier une altération de ce dernier, que ce soit l'altération d'une variable utilisée dans un transfert conditionnel, l'altération du contenu du registre PC, ou toute autre raison possible.

Quelle qu'en soit la raison, l'altération du flux d'exécution se traduit par le fait que la prochaine instruction exécutée diffère de celle qui aurait dû l'être. L'instruction exécutée peut se trouver aussi bien dans le bloc courant (et même être l'instruction qui vient d'être exécutée) que dans tout autre bloc du programme. La figure 5.8 illustre les cas d'altération du flux d'exécution (flèches en pointillés) pouvant se produire après l'exécution de la dernière instruction du bloc A, tandis que la figure 5.9 illustre les altérations pouvant se produire à la fin de l'exécution de la deuxième instruction du bloc B.

Le bloc C peut être un bloc du programme en exécution mais peut aussi être un bloc d'un autre programme se trouvant en mémoire et n'étant pas en exécution. Ainsi, notre modèle de pannes intègre les altérations du flux étant internes et externes au programme en exécution. Les pannes amenant à des zones mémoire ne contenant pas de programme (et donc contenant des instructions invalides) ne sont pas représentées. Ce choix repose sur le fait que dans le cas d'un tel saut, l'interruption déclenchée par l'exécution de l'instruction invalide à cette adresse résultera en un saut à la routine de recouvrement.

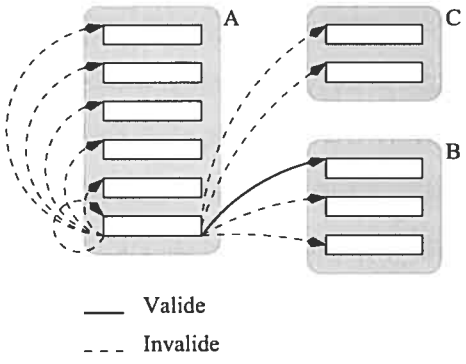

Figure 5.8: Transferts invalides

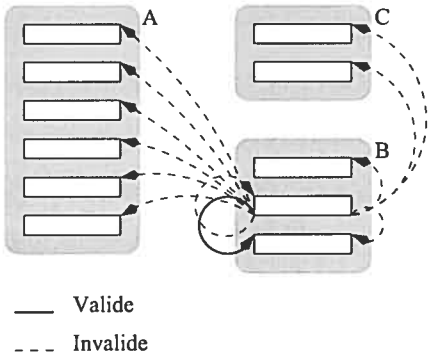

Figure 5.9: Transferts invalides

Notre modèle de panne consiste à permettre toutes les altérations possibles du

flux d'exécution. Toute altération du flux d'exécution ne dégénère pas forcément en une altération du flux de contrôle. En effet, la répétition d'une instruction de test par exemple n'aura aucune influence sur le flux de contrôle du programme dans lequel elle a lieu. Néanmoins, il est impératif de considérer toutes les altérations possibles de manière à pouvoir déterminer leurs conséquences sur le flux de contrôle.

Contrairement aux techniques de vérification par injection de pannes qui consistent à injecter des SEU, nous nous attardons uniquement sur les conséquences que ces derniers ont sur le flux d'exécution des instructions. Ainsi, en injectant l'ensemble des conséquences possibles, nous pouvons déterminer les répercussions sur le flux de contrôle et analyser le comportement des techniques de tolérance aux pannes face à ces altérations.

Afin d'illustrer notre propos, considérons l'ensemble des instructions d'un programme comme étant les noeuds d'un graphe et considérons que le passage d'une instruction à la suivante est un arc. L'injection des conséquences des SEU dans ce graphe consiste à ajouter des arcs invalides (représentant les transferts invalides) de manière à ce que tout état possède un arc vers chaque autre état du graphe, y compris lui-même. Le graphe ainsi obtenu est un *graphe complet*[3] auquel un arc bouclant sur chaque état a été ajouté.

Un tel modèle de panne nécessite l'ajout d'un nombre très important de transitions invalides. En effet, considérons une application de taille très moyenne comme notre programme d'extraction des fonctionnalités manquantes présenté au chapitre 4. Le résultat de la compilation du noyau (en C) en assembleur génère environ 3200 lignes de code. Le modèle de panne nécessitant l'ajout d'un saut invalide de toute instruction vers toute autre, nous obtenons donc un total de $3200 * 3200$, c'est donc un peu plus de 10 millions ($10.24 * 10^6$) de sauts invalides qui devront être injectés et dont les conséquences sur le flux de contrôle devront être analysées durant la vérification. Bien évidemment, la vérification formelle des techniques de tolérance aux pannes sur des programmes usuels mènerait invari-

[3]Un graphe complet est un graphe dont tous les sommets sont reliés deux à deux [126].

ablement à un problème d'explosion d'états dû au nombre trop important de sauts invalides générés.

Pour contrer ce problème, nous proposons de réduire les programmes utilisés à leurs caractéristiques principales, à savoir les types de transitions, et d'utiliser un ensemble de programmes génériques implémentant les instructions de transfert typiques d'une classe de programmes. Les programmes génériques que nous proposons sont présentés dans la section 5.4 qui suit. Ils sont illustrés sous forme de graphes orientés suivant les principes d'abstraction énoncés dans la section 5.2.

5.4 Programmes génériques

Les techniques de tolérance aux pannes ciblées par notre méthodologie focalisent donc uniquement sur les transferts de contrôle. Un programme quelconque au sein duquel une technique de tolérance aux pannes est implémentée comporte un certain nombre de transferts de contrôle en fonction desquels la technique de tolérance aux pannes agit de manière à détecter les erreurs dans le flux de contrôle. En identifiant les caractéristiques des transferts de contrôle, il est possible de les regrouper en différents types distincts. Nous en identifions quatre :

1. Les transferts de contrôle inconditionnels ; branchements non sujets à une quelconque condition,

2. Les transferts de contrôle conditionnels ; branchements en fonction d'une condition (valeur d'un registre, d'une variable, etc.),

3. Les appels et retours de fonctions,

4. Les retours de fonctions dépendant du bloc d'où provient l'appel.

Ayant identifié ces types, il est possible d'implémenter l'un d'eux, ou plusieurs, au sein d'un programme de taille réduite. Nous appelons un tel programme un programme générique dans la mesure où ce dernier implémente le (ou les) type(s) de

transfert(s) de contrôle spécifique(s) à l'ensemble des programmes qui les implémentent aussi. En d'autres termes, un graphe générique représente donc une classe de programmes vis-à-vis d'un (ou plusieurs) types de transferts de contrôle.

Les sections 5.4.1 à 5.4.5 présentent les cinq programmes génériques de taille réduite que nous proposons. Les concepteurs peuvent utiliser ces derniers au besoin et peuvent s'en inspirer pour dériver de nouveaux programmes implémen-tant certains transferts types spécifiques à une application particulière. Dans le cadre de la modélisation d'un système multi-processeur robuste, un concepteur pourrait, par exemple, implémenter un transfert de contrôle relatif à la migration d'un programme vers un autre processeur.

Chaque programme présenté est en réalité dédoublé sous la forme de deux programmes; l'un étant exécuté et l'autre ne l'étant pas. Nous respectons ainsi notre modèle de pannes en permettant une couverture des altérations du flux étant internes et externes au programme en exécution.

5.4.1 $\mathcal{P}_1$ – Transferts inconditionnels

Le premier programme générique, présenté par la figure 5.10, implémente uniquement les types de transferts inconditionnels. Il se compose de trois blocs (états 1, 2, et 3) ainsi que de trois transferts inconditionnels effectuant une boucle entre les différents blocs.

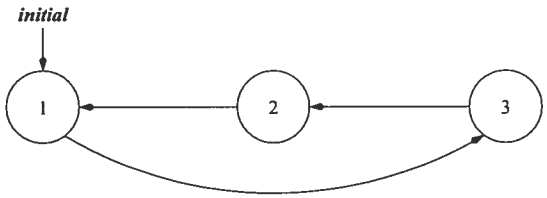

Figure 5.10: Transferts inconditionnels

Bien que ce programme générique soit de taille réduite, il représente un cas typique de programme ne comportant que des transferts inconditionnels. De ce fait, ce dernier permet d'évaluer la fiabilité qu'une techniques de tolérance aux pannes

pourrait avoir face à des altérations du flux de contrôle qui affecteraient un programme ne comportant que des transferts inconditionnels entre les différents blocs.

5.4.2 $\mathcal{P}_2$ – Transferts conditionnels

Le second programme générique, présenté par la figure 5.11, implémente uniquement des transferts conditionnels. Il représente donc la classe de programmes au sein desquels tous les transferts sont d'ordre conditionnel.

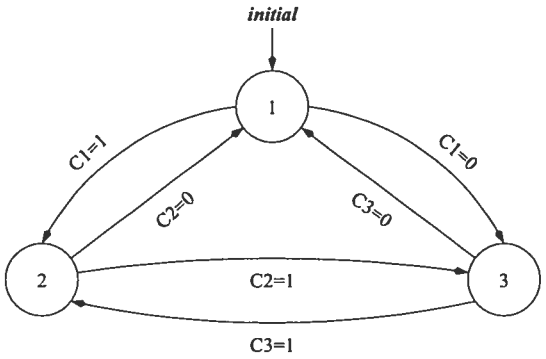

Figure 5.11: Transferts conditionnels

Suivant le même principe que précédemment, ce programme permet d'évaluer le comportement d'une technique de tolérance aux pannes face à des altérations du flux de contrôle dans un programme possédant uniquement des transferts de contrôle de type conditionnels.

5.4.3 $\mathcal{P}_3$ – Appels de fonctions

Notre troisième graphe, implémente un appel de fonction et son retour. En outre, nous avons ajouté une transition inconditionnelle allant du bloc 2 au bloc 1 pour permettre une exécution en boucle.

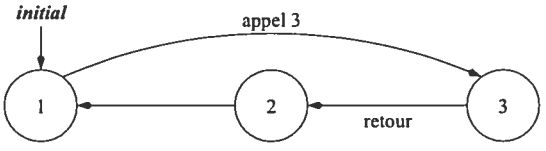

Figure 5.12: Appels/retour de fonctions

L'organisation des blocs en mémoire correspond à leur numéro d'identification. Ainsi, la dernière instruction du bloc 1 est suivie par la première instruction du bloc 2 et ainsi de suite. Ainsi, le retour de l'appel du bloc 3 aboutira à l'instruction qui suit la dernière instruction (appel) du bloc appelant, c'est-à-dire la première instruction du bloc suivant, qui se trouve être le bloc 2 dans le cas de ce programme.

5.4.4 $\mathcal{P}_4$ – Retours de fonctions dépendant du bloc appelant

Le quatrième graphe implémente un retour de fonction dépendant du bloc appelant. Suivant le fait qu'il ait été appelé par le bloc 1 ou par le bloc 3, le retour de l'appel du bloc 5 aboutira respectivement au bloc 2 ou au bloc 4

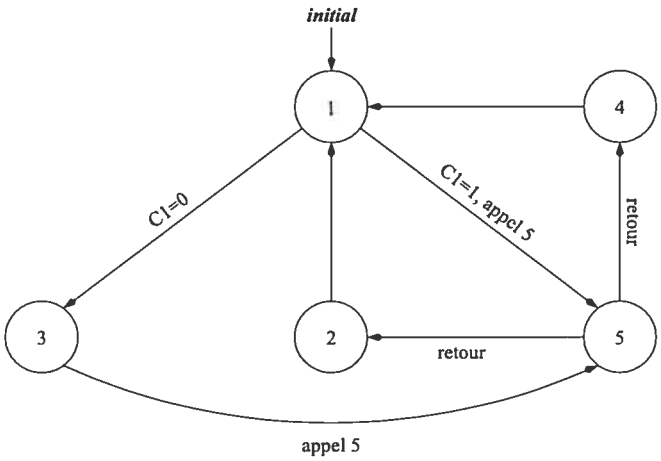

Figure 5.13: Retours de fonction dépendant du bloc appelant

Nous avons dû ajouter un transfert conditionnel à partir du bloc 1 de manière à permettre de choisir un chemin ou l'autre. En outre, nous avons aussi ajouté deux transferts inconditionnels vers le bloc 1 à partir des blocs 2 et 4 de manière à permettre une exécution en boucle.

5.4.5 $\mathcal{P}_5$ – Tous types confondus

Ce dernier programme implémente tous les types de transfert rencontrés dans les graphes précédents. Tandis que les graphes précédents permettent d'évaluer le comportement des techniques de tolérance aux pannes sur des classes de programmes

précises, ce graphe permet de les évaluer sur un programme représentant une classe de programmes implémentant tous les types de transitions. En d'autres termes, ce graphe est un graphe générique complet dans la mesure où il permet de représenter tous les programmes. Les quatre autres graphes restent néanmoins utiles puisqu'ils permettent d'évaluer des faiblesses d'une méthode de manière incrémentale et partitionnée.

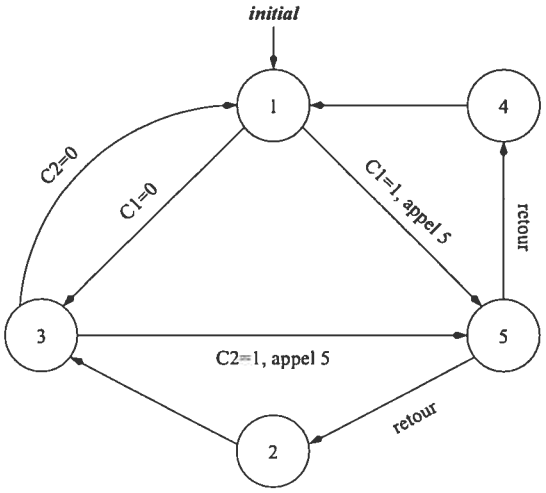

Figure 5.14: Tous types confondus

Bien entendu, de multiples programmes de ce type pourraient être utilisés et les concepteurs sont libres, selon la nécessité, de créer des programmes plus ou moins complexes, présentant divers pourcentages de certains types de transitions, voire même présentant d'autres types de transitions spécifiques à une application particulière. Comme mentionné au début de la section 5.4, un concepteur pourrait, dans le cadre de la modélisation d'un système multi-processeur robuste, implémenter un transfert de contrôle relatif à la migration d'un programme vers un autre processeur.

5.5 Pseudo-assembleur et traduction en PROMELA

Cette section présente notre langage pseudo-assembleur ainsi que les mécanismes de traduction permettant la génération d'un modèle PROMELA pour le *model-*

checker SPIN.

5.5.1 Pseudo-assembleur

Notre pseudo assembleur consiste en un sous-ensemble du langage PROMELA enrichi d'un certain nombre d'instructions supplémentaires. Les instructions se terminent par un retour à la ligne, le caractère ; à la fin d'une instruction est interdit. Le sous-ensemble du langage PROMELA se compose des éléments suivants :

- Déclarations de variables restreintes à `int`, `byte`, `bool` (seuls les nombres décimaux sont permis, les nombres hexadécimaux ne sont pas supportés),
- Toutes instructions d'affectation et toutes opérations sur les variables qui sont permises en PROMELA,
- Les étiquettes (`[A-Za-z][A-Za-z0-9]*:`) utilisées pour les sauts,
- Instructions de saut `goto`,
- Instruction conditionnelle `if :: (`*`condition`*`)` restreinte à sa forme la plus simple où le `else` n'est pas supporté et où seulement la ligne suivante est exécutée si la condition est vraie :

  ```
  if :: (D >= 3)
    G = G + D
  G = 0
  ```

  Si la condition `D >= 3` est vraie, la ligne `G = G + D` sera exécutée, dans le cas contraire, ce sera la ligne `G = 0`.

- Commentaires `/* ... */` avec comme restriction le fait qu'un commentaires doit commencer et se terminer sur la même ligne et ne peut être placé à la suite d'une instruction.

Les éléments qui ont été retirés du langage sont principalement ceux relatifs aux algèbres de processus. Ces éléments sont les suivants :

- Déclarations de processus, tâches, et fonctions.
- Mécanismes de canaux et opérations de transmission sur les canaux.
- Déclarations de types utilisateur.
- Boucles `do ... od`.

Les éléments ajoutés concernent non seulement les instructions spécifiques à un langage assembleur mais aussi certaines instructions nécessaires aux abstractions définies dans la section 5.2 :

- Étiquette de début de bloc `:BLOCK`
- Instruction `NOP`, abstraction des instructions sur les données. Néanmoins, rien n'empêche le concepteur d'ajouter de telles instructions s'il le souhaite.
- Appel de fonction `call`. Le retour se fait sur l'instruction suivant celle de l'appel. Ainsi, si l'appel se trouve à la fin d'un bloc, le retour se fera donc sur la première instruction du bloc suivant en mémoire.
- Retour de fonction `return`.
- Variables conditionnelles : bits à déclarer dans un fichier séparé qui sont modifiés automatiquement par le *model-checker* durant la vérification. De plus amples explications sont données dans la section 5.6 ainsi que dans le chapitre 6 portant sur l'implémentation de notre méthodologie.
- Instruction de détection d'une modification du flux de contrôle `seu_detected` permettant à la technique de tolérance aux pannes de signaler au système qu'elle a détecté une erreur.
- Étiquettes `:CI` (*Checking-Instruction*) et `:CV` (*Checking-Variable*) à placer obligatoirement devant les instructions relatives à la technique de tolérance aux pannes de manière à permettre une différentiation entre ces dernières et les instructions du programme. Cette différentiation est cruciale pour la phase

de compilation générant le modèle PROMELA, elle permet non seulement la différentiation entre le modèle sujet aux erreurs et le modèle de référence mais aussi un certain nombre d'optimisations qui sont discutées dans la section 5.6

À titre d'exemple l'implémentation de la technique CFCSS [127] dans le bloc 1 du programme générique 4 contenant un retour de fonction dépendant du bloc appelant est donné ci-dessous :

```
:BLOCK 1
block1:

  :CI G = G ^ 109
  :CI G = G ^ D
  :CI if :: (G != 109)
    :CI seu_detected
  :CI D = 109

  NOP

  if :: (C1 == 0)
    goto block3
  call block5
```

5.5.2 Mécanismes de traduction

Le flux de traduction d'un programme pseudo-assembleur est illustré par la figure 5.15 : Les différentes étapes du processus de traduction sont détaillées dans les explications qui suivent.

Programme pseudo-assembleur – Le point d'entrée du flux de traduction est le programme pseudo-assembleur implémentant la technique de tolérance aux pannes devant être vérifiée.

Analyse lexicale – La première phase de traduction consiste en une analyse lexicale du programme pseudo-assembleur à l'aide d'un analyseur lexicale. Cette analyse lexicale génère deux processus PROMELA :

1. Un processus dit "complet" correspondant exactement au programme pseudo-assembleur donné en entrée,
2. Un processus dit "simplifié" duquel les instructions relatives à la technique de tolérance aux pannes ont été retranchées. Ce processus contient uniquement les instructions de saut du programme donné en entrée.

Les deux processus utilisent les mêmes variables conditionnelles, leur permettant ainsi de prendre les mêmes décisions de transfert de contrôle.

Routines de saut – La suite du processus consiste en la génération des routines de saut qui sont ajoutées aux processus générés par la phase d'analyse lexicale, donnant ainsi :

1. Le processus complet appelé processus-fautif, contenant la technique de tolérance aux pannes est sujet aux transferts erronés (conséquences de SEU). Sa routine de sauts est conçue de telle manière que le passage de toute instruction à la prochaine devant être exécutée peut être altérée résultant ainsi en un saut vers n'importe quelle autre instruction, y compris celle venant d'être exécutée.
2. Le processus simplifié, appelé processus référence, n'est pas sujet aux transferts erronés, sa routine de sauts n'altère pas le flux d'exécution. Ce processus devient ainsi le modèle de référence dont le flux de contrôle sera comparé à celui du processus-fautif, permettant ainsi au *model-checker* de se rendre compte des altérations du flux de contrôle dans le processus-fautif.

Mise en commun – La phase de génération des routines de saut est suivie par la phase de mise en commun au cours de laquelle les deux processus générés ultérieurement sont mis en commun avec un processus d'arbitrage dans un même modèle PROMELA.

Modèle final – Le modèle résultant de l'ensemble de cette traduction est un modèle PROMELA contenant deux processus implémentant le programme donné en entrée. L'un des processus contient l'implémentation de la technique de tolérance aux pannes et est sujet aux transferts erronés, l'autre n'implémente pas ladite technique et n'est pas sujet aux transferts erronés. Un troisième processus, appelé processus-arbitre, surveille le flux de contrôle des autres processus, et signale les désynchronisations au *model-checker*.

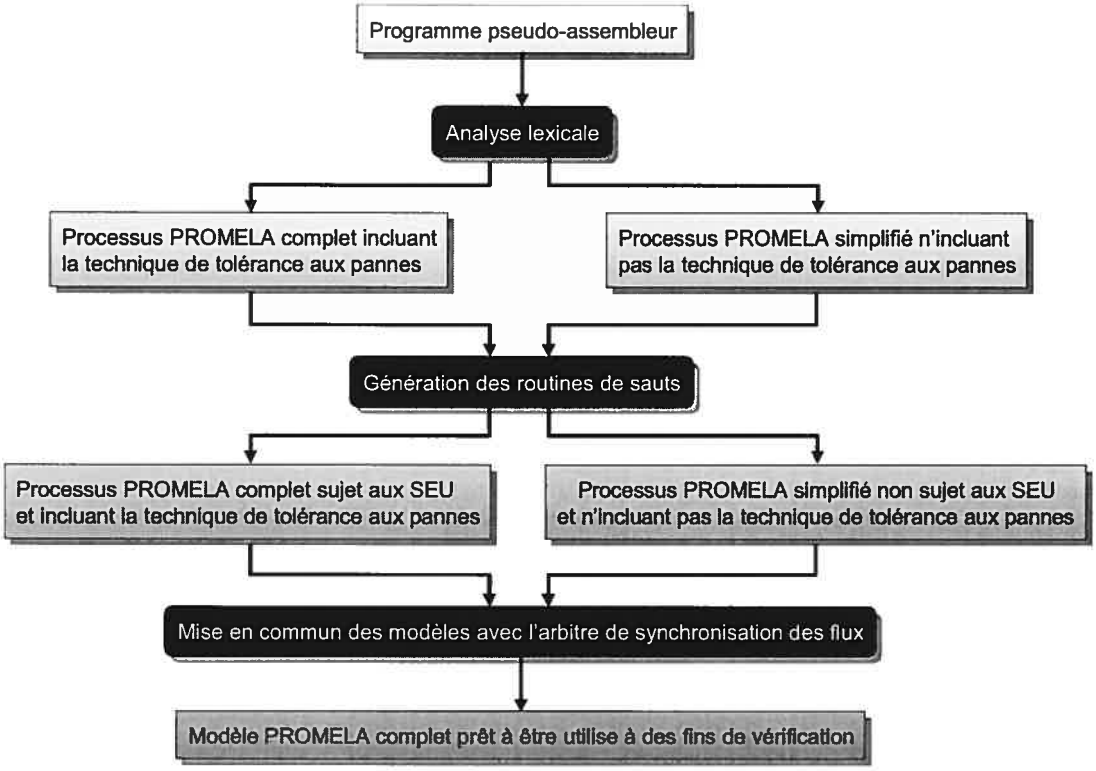

Figure 5.15: Mécanisme de traduction

Le processus de traduction est suivi du processus de vérification formelle utilisant le *model-checker* SPIN tel que présenté en détails dans la section qui suit.

5.6 Vérification formelle

La vérification formelle permet de s'assurer, de manière statique, c'est-à-dire sans avoir recours à des simulations, qu'un modèle évoluant dans le temps satisfera

toujours une propriété, quand bien même le nombre ou la longueur des scénarii d'exécution de ce modèle seraient infinis. Par exemple, considérons un système producteur-consommateur échangeant des données *ad vitam eternam* à travers une file d'attente de type FIFO de taille bornée. Il nous sera possible de vérifier formellement que le producteur n'ajoutera jamais d'élément dans la file d'attente si celle-ci est pleine. Bien que le producteur et le consommateur soient des processus évoluant dans le temps, un outil de vérification formelle ne procédera en aucun cas à une vérification par simulation. Il effectuera une sorte de preuve mathématique de manière statique. De plus amples détails sont donnés un peu plus loin dans ce chapitre.

Dans le cadre des techniques de tolérance aux pannes, la propriété principale que nous désirons vérifier est que tout transfert erroné altérant le flux de contrôle du processus-fautif sera détecté par la technique de tolérance aux pannes implémentée dans ce processus. Pour ce faire, nous devons tout d'abord être en mesure de détecter les altérations du flux de contrôle. Ceci est pris en charge par le processus-arbitre. Le fonctionnement des échanges entre le processus-arbitre et les autres processus, illustré par la figure 5.16, suit un protocole très simple :

1. Lors de l'entrée dans un nouveau bloc, le processus-fautif, ainsi que le processus référence envoient le numéro du bloc en question au processus-arbitre. Une fois ce numéro envoyé, ils attendent la confirmation du processus-arbitre pour continuer leur exécution.

2. Lorsque le processus-arbitre a réceptionné le message de synchronisation de chaque processus, il compare les numéros de blocs et détermine alors si les flux de contrôle sont toujours synchronisés (numéros de bloc reçus identiques), ou s'ils ne le sont plus (numéros de blocs reçus différents). Dans le cas où les flux sont désynchronisés, cela signifie que celui du processus-fautif a été altéré, auquel cas le processus-arbitre le signale au *model-checker*.

3. Dans la mesure où les deux processus sont en attente du signal du processus-arbitre, ce dernier modifie les valeurs des variables utilisées dans les condi-

tions de transfert, permettant ainsi l'exploration de toutes les combinaisons possibles par le *model-checker*

4. Une fois l'état de la synchronisation déterminé, le processus-arbitre envoie un message de continuation aux deux processus, leur permettant ainsi de continuer leur exécution jusqu'à leur arrivée dans un nouveau bloc où l'étape de vérification de synchronisation sera à nouveau effectuée.

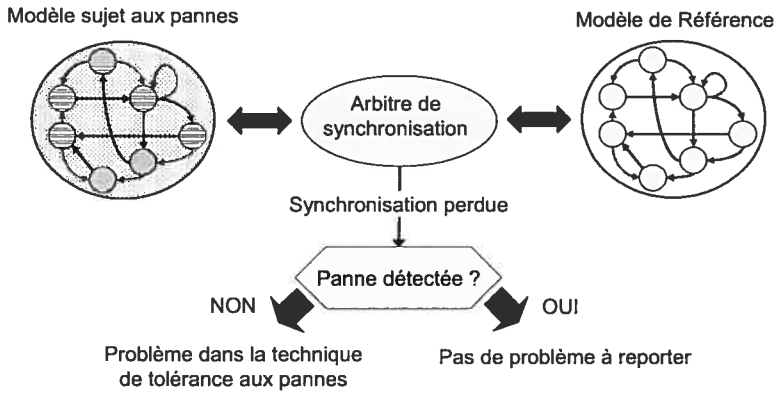

Figure 5.16: Détection des altérations du flux de contrôle du le processus-fautif

Le fait de retrancher les instructions relatives à la technique de tolérance aux pannes du processus-référence permet de ne pas s'encombrer de ces dernières dans la mesure où elles sont inutiles au processus-référence. Réduisant la complexité du modèle, cette suppression s'accompagne aussi d'un gain de performance, tant en occupation mémoire qu'en temps de vérification.

La vérification formelle consiste à vérifier un certain nombre de propriétés relatives au programme donné par *model-checking*. Le *model-checking* est une technique permettant de vérifier formellement les systèmes décrits sous forme d'automates. Son utilisation est largement répandue dans les domaines matériel et logiciel. Sa puissance repose sur le fait qu'elle s'appuie sur des concepts mathématiques qui permettent d'effectuer une preuve complète de la validité d'un modèle, c'est-à-dire de vérifier que ce dernier satisfait un certain nombre de propriétés exprimées en logique temporelle.

La logique linéaire temporelle (LTL), utilisée par SPIN, est un champ des mathématiques portant sur la faculté d'exprimer des propriétés du présent, du futur et leurs rapports, sur des séquences d'exécution σ. Une formule LTL ϕ est satisfaite par une séquence σ si et seulement si elle est satisfaite à partir de la première position (état) s_0 de σ. Cette satisfaction est dénotée $\sigma \models \phi$. Une formule LTL est construite à partir de variables propositionnelles mises en relation en utilisant les connecteurs de la logique booléenne ainsi que les opérateurs unaires $(\mathbf{X}, \Box, \Diamond)$ et l'opérateur binaire $(\mathbf{U})$ suivants :

$\mathbf{X}\phi$ – Next

Le prochain état de la séquence doit satisfaire la formule ϕ.,

$\Box\phi$ – Always

Tous les états de la séquence doivent satisfaire ϕ,

$\Diamond\phi$ – Eventually

Il finira indubitablement par exister un état de la séquence qui satisfera ϕ.

$\phi\mathbf{U}\psi$ – Until

ψ finira indubitablement par être satisfaite et en attendant ce moment, ϕ sera toujours satisfaite.

Le *model-checker* SPIN prend en entrée un modèle écrit en PROMELA. La représentation interne de ce modèle consiste en un graphe orienté constitué de noeuds reliés par des arcs. Les noeuds représentent les états du modèle tandis que les arcs représentent les exécutions possibles faisant changer le modèle d'état. Un ensemble de propositions atomiques est associé à chaque état et représente les propriétés qui sont satisfaites à ce point de l'exécution. Étant donné un modèle PROMELA $\mathcal{M}$ doté d'un état initial s_0 et étant donné une formule LTL ϕ, la phase de *model-checking* consiste à vérifier mathématiquement que $\mathcal{M} \models \phi$.

Étant donné une formule LTL, il est possible de générer un automate $\mathcal{B}$, appelé automate de Büchi, tel que $\mathcal{B} = (S, T, S_0, F, \Sigma)$ où [66] [20] :

- S est l'ensemble d'états et $|S| \leq 2^{O|\phi|}$,
- S_0 est l'ensemble d'états initiaux,
- $T \subseteq S \times \Sigma \times S$ est l'ensemble des transitions sur l'ensemble S,
- $F \subseteq S$ est l'ensemble d'états finaux,
- $\Sigma = 2^{Prop}$ est l'alphabet,
- Le langage $L(A)$ reconnu par $\mathcal{B}$ est l'ensemble exact de tous les modèles de ϕ.

L'algorithme de *model-checking* repose sur LTL et la théorie des automates. La vérification du fait que le modèle $\mathcal{M}$ satisfait ou non une propriété ϕ est effectué de la manière suivante :

1. La négation $\neg\phi$ de la formule ϕ est calculée,
2. $\neg\phi$ est traduite en un automate de Büchi $\mathcal{B}_{\neg\phi}$
3. Le produit synchronisé $\mathcal{M} \times \mathcal{B}_{\neg\phi}$ est calculé. Ce produit synchronisé est en réalité le produit cartésien des états et les transitions de $\mathcal{M}$ et $\mathcal{B}_{\neg\phi}$, restreint aux transitions qui sont compatibles, c'est à dire qui peuvent être exécutée simultanément par les deux automates [20]. En fonction du résultat, deux verdicts sont possibles :
   (a) $\mathcal{M} \times \mathcal{B}_{\neg\phi} = \emptyset$ indique que $\mathcal{B} \models \phi$
   (b) $\mathcal{M} \times \mathcal{B}_{\neg\phi} \neq \emptyset$ indique que $\mathcal{B} \not\models \phi$ et un ensemble de contre exemples peut être généré à partir de l'automate résultant de $\mathcal{M} \times \mathcal{B}_{\neg\phi}$

En procédant de cette manière, le *model-checking* assure une exploration totale du modèle à vérifier, ainsi que de tous les scénarii relatifs à une formule LTL donnée. Contrairement aux méthodes reposant sur la simulation, cette vérification ne consiste pas en une exploration exhaustive de tous les scénarii possibles mais en une

preuve mathématique. Rappelons néanmoins que, malgré son efficacité, le *model-checking* s'accompagne de coûts non négligeables. La vérification d'une formule LTL est *p-space complète* ; la quantité de mémoire nécessaire augmente exponentiellement avec la taille du modèle (états, structures de données, transitions, etc.).

Dans le cadre de notre vérification de techniques de tolérance aux pannes, l'utilisation de programmes génériques de taille réduite nous permet de rester dans des proportions raisonnables. La description des propriétés que nous vérifions par *model-checking* repose sur deux variables booléennes globales (c'est-à-dire accessibles à tous les processus du modèle PROMELA) et sur l'utilisation de la logique linéaire temporelle (LTL) pour décrire leurs relations sous forme de propriétés à vérifier. L'ensemble des trois variables suivantes caractérise l'état du modèle à tout moment :

seu-injecté – Cette variable permet de savoir si un transfert erroné a été injecté. La valeur initiale de cette dernière (*false*) est changée à *true* par la routine de saut du processus-fautif au moment de l'injection du transfert erroné.

synchro-perdue – Cette variable permet de savoir si le flux de contrôle du processus-fautif n'est plus synchronisé avec celui du processus-référence. La valeur initiale de cette dernière (*false*) est changée à *true* par le processus-arbitre lorsque celui-ci détecte une perte de synchronisation.

seu-détectée – Cette dernière variable permet de savoir si le processus-fautif a détecté une altération du flux de contrôle. Sa valeur initiale (*false*) est changée à true par le processus-fautif lors de la détection d'une altération du flux. Le processus de traduction remplace l'instruction `seu_detected` par une instruction PROMELA changeant la valeur de cette variable.

La vérification du modèle consiste principalement à s'assurer que tout transfert erroné altérant le flux de contrôle du processus-fautif sera détectée par la technique de tolérance aux pannes implémentée dans ce processus. En cas de fuites, il est possible d'extraire les scénarii représentatifs. En réalité, étant donné un SEU, différentes conséquences sont envisageables :

1. Le SEU altère le flux de contrôle, deux résultats sont alors possibles :
   (a) L'altération est détecté par la technique de tolérance aux pannes,
   (b) L'altération n'est pas détecté par la technique de tolérance aux pannes
2. Le SEU n'altère pas le flux de contrôle, deux autres résultats sont alors possibles :
   (a) La technique de tolérance aux pannes ne détecte pas d'altération,
   (b) La technique de tolérance aux pannes détecte une altération qui ne s'est pas produite.

Seul la considération du cas 1.a est nécessaire pour s'assurer qu'une technique est fiable. Néanmoins, de manière à extraire des données plus détaillées et à pouvoir calculer des pourcentages représentatifs, nous avons défini et utilisé les propriétés qui suivent :

Φ_1 – **Implémentation correcte** – Premièrement, de manière à nous assurer que la technique implémentée ne détecte pas de fausses altérations du flux de contrôle, nous désactivons l'injection de transferts erronés et vérifions ainsi qu'aucune altération ne soit détectée par la technique. Formellement, ceci est exprimé en LTL par la formule suivante :

$$\Box(\neg\texttt{synchro-perdue} \wedge \neg\texttt{seu-détectée})$$

qui stipule qu'en tout temps, la synchronisation entre les flux de contrôle des deux processus ne sera pas perdue et qu'aucune altération ne sera signalée. Dans le cas où la vérification échoue, le *model-checker* génère les scénarii contre-exemples.

Φ_2 – **Détection des altérations du flux de contrôle** – Une fois certains que la technique ne détecte pas de pannes lorsque qu'aucun transfert erroné n'est injecté, nous pouvons vérifier qu'elle détectera toutes les altérations du flux de contrôle. Ceci est effectué par la formule :

$$\Box(\texttt{synchro-perdue} \rightarrow \Diamond\texttt{seu-détectée})$$

qui signifie qu'en tout temps, si la synchronisation a été perdue, cette altération du flux de contrôle finira un jour où l'autre par être détectée. Dans le cas où la vérification échoue, le *model-checker* génère les scénarii contre-exemples, permettant ainsi de prendre connaissance du nombre de cas dans lesquels la technique n'est pas fiable.

Φ_3 – **Non-détection des altérations du flux de contrôle** – En complément à la précédente, cette propriété stipule que dans tous les cas, si un transfert erroné injecté cause une altération du flux de contrôle, cette altération ne sera jamais détectée par la technique en évaluation :

$$\Box((\texttt{synchro-perdue} \rightarrow \Box\neg\texttt{seu-détectée})$$

Dans la mesure où le *model-checker* génère un ensemble de scénarii contre-exemples, il est alors possible, par un simple calcul, de prendre connaissance du pourcentage du nombre d'altérations ayant échappées à la technique. En supplément, les deux formules qui suivent nous permettent d'obtenir des informations sur les cas dans lesquels la technique détecte, ou ne détecte pas les transferts erronés n'ayant pas altéré le flux de contrôle.

Φ_4 – **Non-détection de fausses altérations du flux de contrôle** – Cette formule stipule que, dans tous les cas, si un transfert erroné a été injecté et que la synchronisation n'est jamais perdue, alors la technique ne doit jamais détecter d'altération du flux de contrôle.

$$\Box((\texttt{seu-injectée} \wedge \Box\neg\texttt{synchro-perdue}) \rightarrow \Box\neg\texttt{seu-détectée})$$

Ainsi, l'ensemble de contre-exemples généré si la formule est invalidée permet d'avoir une idée du nombre de transferts erronés n'altérant pas le flux de contrôle qui sont détectés.

Φ_5 – **Détection de fausses altérations du flux de contrôle** – La formule :

$$\Box((\texttt{seu-injectée} \wedge \Box\neg\texttt{synchro-perdue}) \rightarrow \Diamond\texttt{seu-détectée})$$

permet de vérifier que pour tous les cas où un transfert erroné n'entraîne pas d'altération du flux de contrôle, la technique détectera quand même une altération, permettant ainsi, en conjonction avec les résultats de la formule précédente, d'obtenir les pourcentages de transferts erronés n'altérant pas le flux de contrôle qui sont détectés, et non détectés par la technique en évaluation.

Φ_6 – **Transferts erronés altérant le flux de contrôle** – La formule :

$$\Box((\texttt{seu-injectée} \rightarrow \Diamond\texttt{synchro-perdue})$$

permet de vérifier que dans tous les cas, un transfert erroné entraînera une altération du flux de contrôle. Les contre-exemples générés en cas d'échec nous permettent de connaître les quantités de transferts erronés n'altérant pas ledit flux.

Φ_7 – **Transferts erronés n'altérant pas le flux de contrôle** – Finalement, la formule :

$$\Box((\texttt{seu-injectée} \rightarrow \Box\neg\texttt{synchro-perdue})$$

permet de vérifier que dans tous les cas, un transfert erroné n'entraînera pas d'altération du flux de contrôle. Les contre-exemples générés en cas d'échec, en conjonction avec les ceux de la formule précédent, nous permettent de calculer les pourcentages de transferts erronés altérant et n'altérant pas le flux de contrôle.

Dans les cas où la vérification d'une formule se solde par un échec, il est possible, via l'utilisation de SPIN, d'extraire tous les scénarii illustrant les contre-exemples. Dans notre cas, nous avons ainsi été en mesure de calculer les statistiques qui sont présentées au chapitre 6. Il doit bien entendu être clair que cette extraction de tous les scénarii à des fins statistiques n'est pas obligatoire. Les concepteurs peuvent se contenter de vérifier les propriétés Φ_1 et Φ_2 et ainsi analyser les lacunes de la technique et la raffiner en conséquence. Le fonctionnement de la vérification formelle, ainsi que l'extraction des scénarii sont présentés dans le chapitre 6.

5.7 Résumé

Dans ce chapitre, nous avons présenté notre méthodologie de vérification formelle des techniques de tolérance aux pannes dédiées à la détection des altérations du flux de contrôle par des transferts erronés (conséquences de SEU).

Cette méthodologie fournit un ensemble de programmes génériques de taille réduite au sein desquels une technique à évaluer peut être implémentée. Une fois cette implémentation effectuée, une phase de traduction permet la génération d'un modèle formel PROMELA. Ce dernier est alors vérifié formellement en conjonction avec un ensemble de propriétés formelles exprimées en LTL à l'aide du *model-checker* SPIN. L'utilisation d'un ensemble de programmes de taille réduite évite le problème d'explosion d'état, permettant ainsi d'exploiter au maximum la puissance du *model-checking*.

L'utilisation d'une telle méthodologie de vérification permet aux concepteurs de vérifier de manière rapide et efficace le bon fonctionnement d'une technique de tolérance aux pannes sur un ensemble de programmes de taille réduite avant de se lancer dans de longues et coûteux processus de vérification par injection de pannes.

CHAPITRE 6

TOLÉRANCE AUX PANNES – OUTILS ET APPLICATIONS

Ce chapitre porte sur l'implantation de notre méthodologie de vérification des techniques de tolérance aux pannes ainsi que sur son évaluation. Dans un premier temps, nous présentons l'implémentation des concepts théoriques énoncés au chapitre 5 sous forme d'un outil prototype. Ensuite, nous illustrons l'utilisation de ce prototype sur trois études de cas.

Les deux premières portent sur la vérification de techniques dont des défauts de fiabilité ont été révélés par des techniques d'injection de pannes. Les résultats obtenus par vérification formelle confirment ces défauts. La troisième étude de cas porte sur la vérification d'une technique dont l'évaluation par injection de pannes a révélé la fiabilité. Dans ce cas aussi, les résultats obtenus par vérification formelle confirment les résultats obtenus par injection de pannes.

L'existence d'une implémentation permet tout d'abord d'affirmer les concepts sur lesquels notre méthodologie repose tout en rendant possible son évaluation au delà d'un cadre purement théorique. Les résultats de cette évaluation sont basés sur des études de cas pertinentes. La mise en évidence de résultats de vérification obtenus conformes à ceux obtenus précédemment via des techniques d'injection de pannes démontre la pertinence de notre méthodologie.

6.1 Implémentation

L'implémentation présentée dans ce chapitre intègre le processus de traduction ainsi que les mécanismes d'automatisation du processus de vérification formelle ainsi que de celui d'extraction des résultats, tous présentés au chapitre 5.

La majeure partie des processus de traduction et d'automatisation est implémentée sous forme de scripts pour l'interpréteur de commandes (*shell*) BASH [128] et via l'utilisation de fichiers `Makefile` [129]. Ce choix est justifié par le fait qu'il

s'agit ici du développement d'un prototype, pour lequel l'accent est mis non pas sur la performance mais sur le fonctionnement. D'autre part, les scritps BASH et les fichiers `Makefile` sont tout à fait adéquats quand il s'agit d'automatisation de processus. Leur emploi très varié (installation, maintenance, automatisation, etc.) par la communauté UNIX [130] le confirme.

L'implémentation complète de l'outil ainsi que les études de cas et un manuel d'utilisation sont disponibles sur le web à l'adresse donnée en référence [121] sous forme d'une archive en format Zip. Notre prototype peut être décomposé en trois éléments : les mécanismes de traduction, les mécanismes de vérification et extraction des résultats, et enfin, l'aspect automatisation.

Traduction du pseudo-assembleur vers PROMELA – Notre pseudo-assembleur est traduit en PROMELA par un analyseur lexical écrit en LEX [131] en conjonction avec un ensemble de scripts de génération en shell BASH.

Vérification formelle – La vérification formelle est effectuée par le *model-checker* SPIN. Aucune modification n'a été faite sur cet outil. Le mode standard est utilisé pour la compilation et la vérification du modèle PROMELA. Le mode graphique est utilisé pour l'extraction des scénarii contre-exemples.

Automatisation – L'automatisation du processus de traduction, vérification, et extraction des résultats repose sur l'emploi de fichiers `Makefile` permettant de lancer le processus entier à l'aide d'une simple commande.

Les implémentations de ces trois processus sont présentées en détail dans les sous-sections qui suivent.

6.1.1 Traduction du pseudo-assembleur vers PROMELA

L'étape de traduction est la plus complexe sur le plan du travail d'implémentation effectué. Elle est gérée par un script BASH `generate_model.sh` qui prend en charge la gestion de tout le processus de traduction. Ce processus est découpé en trois étapes bien distinctes qui génèrent les trois processus arbitre, référence, et

fautif. Une fois cette génération effectuée, les trois processus sont mis en commun, constituant ainsi un modèle PROMELA prêt à l'utilisation.

6.1.1.1 Analyseur lexical

La traduction du programme pseudo-assembleur en PROMELA est effectuée par un analyseur lexical. Cet analyseur reconnaît les diverses instructions autorisées du pseudo-assembleur et imprime leur équivalent en PROMELA. Ce faisant, il se charge de la génération des étiquettes de blocs et d'instructions ainsi que de la génération des mécanismes de synchronisation avec l'arbitre. La grammaire utilisée pour l'analyse lexicale est donnée ci-dessous :

```
EMPTY_SPACE     ({SP}\n[ \t]+|{SP}\n[ \t]+)
COMMENT         \/\*([^\*]|\*[^\/])*\*\/
SP              [ \t]*

VAR_DCL         (int|byte|bool)[^\n]+
BLOCK_DCL       {SP}:BLOCK\ [0-9]+{SP}
LABEL_DCL       {SP}[A-Za-z][A-Za-z0-9_]+:{SP}

GOTO            {SP}goto{SP}[A-Za-z0-9_]+{SP}

IF              {SP}if[ ].*

F_CALL          {SP}call{SP}[A-Za-z0-9_]+{SP}
F_RETURN        {SP}return{SP}

NOP             {SP}NOP{SP}

SEU_DETECTED    {SP}seu_detected{SP}

INSTRUCTION     [^\n]+
```

L'équivalent de l'instruction pseudo-assembleur consiste en réalité en une série d'instructions exécutées de manière atomique. Parmi ces instructions se trouve l'instruction contenue dans le programme pseudo-assembleur. Les autres instructions sont dédiées à être utilisées par le *model-checker*. L'exemple ci-dessous il-

lustre le segment de l'analyseur lexical se chargeant de la traduction d'une instruction autre que les instructions spécifiques spécifiées dans la grammaire donnée précédemment en exemple.

```
{INSTRUCTION} {
  if (new_block) {
    printf("i%04d_b%02d:\tatomic {\n", block_inst, block);
    printf("\t\t  seuSync!%d;\n", block);
    printf("\t\t  seuSync?x;\n");
    printf("\t\t  printf(\"MSC: BLOCK %d\\n\");\n", block);
    new_block = 0;
  } else {
    printf("i%04d_b%02d:\tatomic {\n", instruction, block);
  }
  if(!msc(yytext, tmp)) {
    printf("\t\t  printf(\"MSC: i%04d_b%02d: %s\\n\");\n",
   instruction, block, yytext);
  } else {
    printf("\t\t  printf(\"MSC: i%04d_b%02d: %s\\n\");\n",
   instruction, block, tmp);
  }
  printf("\t\t  %s;\n", yytext);
  printf("\t\t  src = s_i%04d;\n", instruction);
  printf("\t\t  dst = s_i%04d;\n", instruction+1);
  printf("\t\t  goto seuJump;\n");
  printf("\t\t}\n\n");
  instruction++;
}
```

Ainsi, considérons l'instruction suivante, précédée par l'étiquette `:CI`, donc relative à une technique de tolérance aux pannes.

```
:CI G = G ^ 69
```

De par sa nature, cette instruction doit faire partie du processus-fautif et le code PROMELA généré pour cette dernière est le suivant :

```
i0002_b01: atomic {
             printf("MSC: i0002_b01: G = G ^ 69\n");
             G = G ^ 69;
```

```
    src = s_i0002;
    dst = s_i0003;
    goto seuJump;
  }
```

Ce segment de code contient l'étiquette de l'instruction suivie de l'instruction `atomic` spécifiant que le segment de code encapsulé par les symboles { et } doit être exécutée de manière atomique, c'est-à-dire sans chevauchement avec aucune autre instruction. Le segment de code donné ici contient une instruction d'affichage dédiée à l'affichage des scénarii contre-exemples (si besoin est). Cette ligne est suivie par l'instruction lue, qui est ensuite suivie par la préparation du saut. La préparation du saut consiste à affecter la valeur de l'instruction source (instruction présente) et de l'instruction de destination (la prochaine à devoir être exécutée) dans des variables, puis à faire un saut à la routine de saut (dernière ligne du segment de code atomique).

6.1.1.2 Génération du processus arbitre

La génération du processus arbitre se charge de l'écriture de la routine de synchronisation avec le processus-fautif et le processus-référence suivant le mécanisme présenté dans le chapitre 5, section 5.6 ainsi que de l'écriture de la routine de changement des variables conditionnelles. Ces variables sont déclarées dans le fichier `conditional_vars.h`. La routine de changement aléatoire consiste à affecter une valeur aléatoire (0 ou 1) à l'une des variables conditionnelles. Ainsi, la déclaration de variables suivante :

```
bit C1
bit C2
```

donnera le segment de code PROMELA suivant à l'intérieur du processus arbitre :

```
do
  :: C1 = 1; break;
  :: C1 = 0; break;
  :: C2 = 1; break;
```

```
    :: C2 = 0; break;
od;
```

L'instruction do choisit aléatoirement d'exécuter l'une des quatre lignes précédées par le symbole ::, ainsi, la valeur d'une des variables sera (ou ne sera pas) modifiée. Nous rappelons que cette opération est effectuée au moment de chaque synchronisation du processus arbitre avec les deux autres processus.

Le processus arbitre contient en prime un mécanisme d'impression du contexte activé lorsqu'un transfert erroné (la conséquence d'un SEU) est injecté. Ainsi, les scénarii contre-exemples contiennent un état détaillé des variables globales ainsi que de la pile d'appel de fonction, au moment de l'injection du transfert erroné.

6.1.1.3 Routines de saut

La génération des routines de saut consiste tout d'abord à identifier toutes les étiquettes des instructions et à déclarer une variable, à laquelle on affecte sa valeur, pour chacune. Ces variables sont ensuite utilisées dans les routines de saut. Pour plus de clarté, reprenons les lignes de l'exemple donné précédemment concernant la préparation de l'exécution de la prochaine instruction :

```
src = s_i0002;
dst = s_i0003;
goto seuJump;
```

Cette préparation consiste à affecter respectivement aux variables `src` et `dst` les valeurs des étiquettes de source et destination qui sont contenues dans des variables du même nom que ces étiquettes précédées par `s_`. Une fois ces affectations faites, un saut vers la routine de transfert `seuJump` est effectué. Cette routine se charge d'injecter ou non un transfert erroné. Pour ce faire, la routine choisit aléatoirement de diriger le transfert soit vers l'instruction cible soit vers n'importe quelle autre instruction. Dans le second cas, il s'agit bien entendu de l'injection de la conséquence d'un SEU.

En outre, un script alternatif, `generate_model_noseu.sh`, est disponible pour la génération d'un modèle PROMELA dans lequel l'injection de transferts erronés

est désactivée. Le comportement de ce script est en tout point similaire à celui de `generate_model.sh` excepté que la génération des routines de sauts du processus-fautif génère une routine ne contenant pas d'injection de pannes.

6.1.2 Vérification formelle

L'implémentation de l'étape de vérification formelle est beaucoup moins complexe que celle de l'étape de traduction. En effet, la totalité de ce processus repose sur l'outil SPIN. En premier lieu, le modèle PROMELA et la formule LTL à vérifier sont compilés ensemble via la commande SPIN prévue à cet effet.

Cette commande compile le modèle PROMELA (ici, `CFCSS.s.pro`) et la formule LTL à vérifier (ici contenue dans le fichier `case_1.ltl` en programme C enregistré dans un fichier appelé `pan.c`. Ce programme est ensuite compilé avec le compilateur gcc tout en spécifiant certaines options particulières à SPIN.

Une fois cette compilation effectuée, l'étape de vérification à proprement parler consiste à démarrer le programme généré et à attendre la fin de son exécution. Une fois cette exécution terminée, si cette dernière a signalé des erreurs par rapport à la formule utilisée, le concepteur est alors libre de démarrer l'interface graphique de manière à visualiser les scénarii contre-exemples.

L'interface graphique de SPIN est intuitive et très simple. Il suffit au concepteur de charger le modèle, la formule à vérifier, et de relancer une vérification en spécifiant à SPIN de s'arrêter dès la première erreur. Dès lors, il est possible de dérouler le scénario contre-exemple présenté par SPIN. La figure 6.1 (page suivante) illustre un exemple de scénario visualisé sous SPIN.

Ce scénario se présente sous forme de *Message Sequence Chart* [125] (MSC). Le scénario se lit de haut en bas. Les trois lignes verticales représentent respectivement les processus arbitre, modèle fautif, et modèle de référence. Les blocs blancs le long des lignes représentent les unités temporelles tandis que les jaunes représentent les instructions pseudo-assembleur exécutées par les modèles.

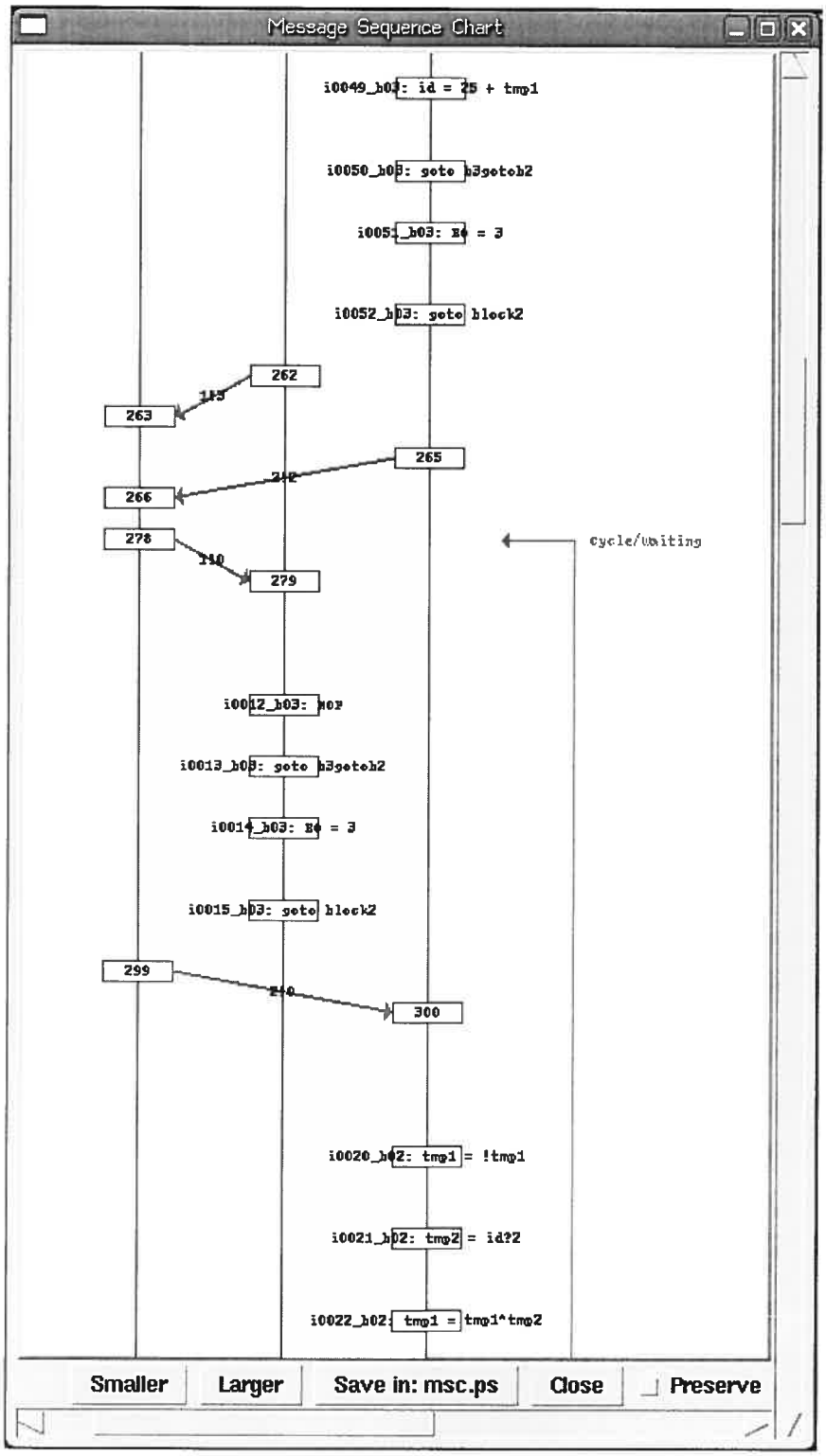

Figure 6.1: Exemple de scénario

6.1.3 Automatisation

L'automatisation complète du processus de vérification repose sur une organisation hiérarchique des fichiers :

- Un premier niveau contient cinq répertoires, un étant dédié à chaque programme générique.
  - Au sein de chacun de ces répertoires se trouvent les répertoires dédiés à l'implémentation des techniques. Ainsi, si trois techniques sont en évaluation, trois répertoires seront présents,
    * À l'intérieur de chacun de ces répertoires se trouve le programme pseudo-assembleur implémentant la technique.

Les répertoires du dernier niveau contiennent un fichier `Makefile` permettant de lancer le processus de traduction, suivi du processus de vérification de chaque formule, suivi enfin d'un processus de récolte et compilation des résultats dans un fichier ASCII.

Au niveau précédent se trouve un fichier `Makefile` permettant de déclencher l'appel au fichier `Makefile` de chacun des répertoires du dernier niveau. Il est ainsi possible de déclencher la vérification de toutes les méthodes implémentées dans un des programmes génériques.

Enfin, au premier niveau se trouve un fichier `Makefile` permettant le traitement des fichiers `Makefile` de chacun des répertoires du second niveau. Une fois les exécutions terminées, ce fichier `Makefile` se charge de l'appel des scripts BASH de récupération et compilation des résultats. Ceci permet ainsi, en une seule et même commande, de lancer toutes la vérification de toutes les techniques implémentées et d'en récupérer les résultats sous forme d'un fichier ASCII.

6.2 Étude de cas

Notre étude de cas repose sur l'implémentation et la vérification de trois techniques de tolérance aux pannes au sein de chacun des cinq programmes génériques définis

dans le chapitre 5. Les deux premières techniques, CFCSS et ECCA, sont des techniques pour lesquelles la simulation a révélé des défauts de fiabilité. La troisième technique, DSM, est une technique pour laquelle aucun défaut de fiabilité n'a été révélé par simulation.

Note : Les calcul des pourcentages donnés dans les tableaux reposent sur le nombre de scénarii extraits par SPIN lors de la vérification des formules Φ_2 à Φ_6 (chapitre 5, section 5.6). Comme nous l'avons mentionné précédemment, il n'est pas nécessaire de procéder à une telle extraction si le but est simplement de vérifier la fiabilité d'une technique.

6.2.1 CFCSS – Control-Flow Checking by Software Signatures

CFCSS [127] est une technique de tolérance aux pannes dédiée à la détection des altérations du flux de contrôle. Les auteurs de l'article [127] décrivent le fonctionnement de CFCSS comme suit.

6.2.1.1 Description

Chaque bloc du programme, dénoté B_i possède une signature unique s_i qui lui est affectée de manière statique au moment de la compilation. La validité du flux de contrôle est vérifiée en utilisant un registre G contenant une signature d'exécution (*run-time signature*) associée avec le bloc courant (celui étant en exécution). Durant l'exécution, pour un bloc donné, la valeur du registre G_i doit être égale à la valeur de la signature s_i du bloc en exécution. Dans le cas contraire, cela signifie qu'une altération du flux de contrôle a eu lieu. La mise à jour de la signature d'exécution est assumée par une fonction de mise à jour f. Cette fonction utilise deux valeurs : la signature du bloc précédent (bloc source), et la signature du bloc courant (bloc destination). Ces deux valeurs étant uniques, elles caractérisent, avec une valeur unique, le transfert T entre un bloc source et un bloc destination. Soit la fonction de signature :

$$f \equiv f(G, d_i) = G \oplus d_i$$

La valeur de $d_d \equiv s_s \oplus s_d$ est calculée en avance durant la compilation et est stockée dans le bloc B_d. Avant que le transfert T_{sd} soit effectué, $G = G_s$. Une fois le transfert effectué, G est mis à jour avec $G_d = f(G_s, d_d)$ basé sur la précédente valeur de G_s et d_d. Si $G_d = s_d \circ f\ B_d$, cela signifie que le flux n'a pas été altéré, dans le cas contraire, si $G_d \neq s_d$, cela signifie qu'un transfert erroné a altéré le flux de contrôle. Les détails de fonctionnement de cette techniques sont disponibles dans l'article [127].

6.2.1.2 Vérification formelle

Une fois le fonctionnement de la technique assimilé, nous avons implémenté CFCSS dans chacun des cinq programmes génériques en moins de deux heures, soit avec une moyenne d'environ 24 minutes par implémentation. Cette rapidité est expliquée par la taille réduite des programmes génériques. Bien entendu, il est nécessaire de maîtriser préalablement le fonctionnement de la technique à implémenter ainsi que la syntaxe du langage pseudo-assembleur.

Le tableau 6.1 présente les résultats de vérification concernant les transferts erronés causant une altération du flux de contrôle. Chaque colonne présente les résultats obtenus avec le programme générique dont le numéro est indiqué dans la case du haut, avec, entre parenthèses, le nombre d'instructions total du programme une fois la technique implémentée. Le programme $\mathcal{P}_1$ fait référence au programme générique défini au chapitre 5, section 5.4.1, le $\mathcal{P}_2$, à celui défini au chapitre 5, section 5.4.2, etc.

Prog. (inst.)	$\mathcal{P}_1$ (30)	$\mathcal{P}_2$ (36)	$\mathcal{P}_3$ (30)	$\mathcal{P}_4$ (58)	$\mathcal{P}_5$ (60)
% Altérations	99.88 %	99.85 %	99.87 %	99.98 %	99.96 %
Non détectées	6.0 %	81.47 %	14.80 %	77.65 %	86.28 %
Temps	0:00.34	0:07.00	0:00.39	0:10.73	0:22.73
Mémoire (MB)	37.369	44.230	37.471	47.302	57.849
Détectées	94.00 %	18.53 %	85.20 %	22.35 %	13.72 %
Temps	0:00.27	0:04.96	0:00.32	0:07.43	0:15.67
Mémoire (MB)	37.369	44.230	37.471	47.302	57.849

Table 6.1: CFCSS – Erreurs causant une altération du flux de contrôle

Le tableau 6.2, quant à lui, présente les résultats de vérification concernant les transferts erronés ne causant pas d'altération du flux de contrôle.

Prog. (inst.)	$\mathcal{P}_1$ (30)	$\mathcal{P}_2$ (36)	$\mathcal{P}_3$ (30)	$\mathcal{P}_4$ (58)	$\mathcal{P}_5$ (60)
% Non-altérations	0.12 %	0.15 %	0.13 %	0.02 %	0.04 %
Détectées	5.08 %	0.25 %	5.08 %	1.44 %	0.31 %
Temps	0:00.23	0:03.18	0:00.37	0:03.54	0:07.89
Mémoire (MB)	37.369	42.387	37.574	43.411	50.476
Non détectées	94.92 %	99.75 %	94.92 %	98.56 %	99.69 %
Temps	0:00.28	0:03.97	0:00.49	0:04.05	0:09.26
Mémoire (MB)	37.369	42.387	37.574	43.411	50.476

Table 6.2: CFCSS – Erreurs ne causant pas d'altération du flux de contrôle

Il est important de re-situer les pourcentages du tableau 6.2 dans leur contexte : en moyenne, seulement 0.09 % des transferts erronés injectés altèrent le flux de contrôle. Ainsi, chaque pourcentage présenté dans ledit tableau ne représente qu'une fraction des 0.09 % de transferts erronés altérant le flux de contrôle et non pas une fraction de l'ensemble des conséquences.

6.2.1.3 Interprétation

La vérification formelle de CFCSS nous permet de constater, à l'instar des résultats obtenus par ses auteurs, qu'elle n'est pas fiable. Nos résultats font état d'une moyenne de 53.24 % d'altérations du flux de contrôle non détectées par CFCSS.

Les auteurs de CFCSS font état d'une moyenne de seulement 3.08 % [127] d'altérations non détectées. Cependant, un autre article, paru environ un an plus tard fait état d'un taux de détection moyen beaucoup plus faible que celui avancé par les auteurs de la technique. En effet, les auteurs de l'article [132] présentent des bancs de test beaucoup plus solides et montrent qu'en moyenne, CFCSS ne détecte que 45.2 % des altérations du flux de contrôle. des altérations du flux de contrôle, en laissant ainsi échapper 54.8 %.

Pour calculer cette moyenne, nous divisons chacun des pourcentages donnés par les différents bancs de test par le pourcentage d'erreurs affectant le flux de contrôle.

Nous "normalisons" les résultats en les amenant à représenter des pourcentages de détection correspondant, non pas aux simplement SEU, mais précisément aux SEU altérant le flux de contrôle. Il ne reste plus alors qu'à additionner les différents résultats et à les diviser par leur nombre.

Comparativement au taux de 54.8 % de l'article [132], nous obtenons un taux 53.24 %, ce qui est légèrement plus bas. Ceci est explicable par le fait que les programmes que nous utilisons sont très réduits et que leur ensemble ne capture probablement pas tous les types de transitions existant dans les programmes des bancs de test. Les résultats obtenus sont néanmoins très proches et, de plus, la vérification formelle de la formule Φ_2, permettant de détecter les altérations échappant à CFCSS n'a requis que 40 secondes tandis que les temps de simulation présentés dans [132] font état de plusieurs heures de calcul.

6.2.2 ECCA – Enhanced Control-flow Checking using Assertions

Tout comme CFCSS, ECCA [133] est une technique de tolérance aux pannes dédiée à la détection des altérations du flux de contrôle qui est la version améliorée de CCA [134]. Les auteurs de l'article [133] définissent le fonctionnement d'ECCA comme suit.

6.2.2.1 Description

Dans CCA, un programme est partitionné en un ensemble d'intervalles sans branchements (*Branch Free Intervals (BFIs)*) et deux identificateurs sont affectés à chacun d'eux. L'identificateur d'intervalle sans branchement(*Branch free interval IDentifier (BID)*) est unique pour chaque BFI. L'identificateur de flux de contrôle (*Control Flow IDentifier (CFID)*) représente le flux de contrôle autorisé et est le même pour tous les BFIs partageant le même BFI (parent) précédent. Le flux de contrôle est vérifié en affectant et vérifiant les BID et CFID. En entrée d'un BFI, le BID courant est vérifié. Ainsi, dans le cas où un transfert résulte en une arrivée au milieu des instructions d'un BFI, le BID courant n'aura pas la valeur attendue

et l'erreur sera détectée.

L'amélioration qu'est ECCA consiste en plusieurs modifications. Au contraire de CCA, ECCA divise un programme en un ensemble de blocs. Un bloc est défini comme étant une collection de BFIs consécutifs dans lesquels existent un unique point d'entrée ainsi qu'un unique point de sortie. Une couverture maximale est obtenue en considérant chaque BFI comme un bloc. Un nombre premier unique supérieur à 2, aussi appelé BID (*Block Identifier*), est affecté à chaque bloc.

La première modification consiste à ajouter l'affectation suivante à l'entrée d'un bloc :

$$id \leftarrow \frac{BID}{\neg(id \bmod BID) \cdot (id \bmod 2)}$$

Cette affectation résultera en une division par zéro si $\neg(id \bmod BID)$ ou $(id \bmod 2)$ vaut 0, révélant ainsi qu'une erreur s'est produite dans le flux de contrôle. La seconde modification consiste à ajouter l'affectation suivante juste avant la sortie d'un bloc :

$$id \leftarrow NEXT + \neg\neg(id - BID)$$

où $NEXT$ est un nombre entier généré durant la compilation et dont la valeur est égale au produit des BID autorisés, c'est-à-dire vers lesquels un transfert est possible, à partir du bloc courant. Due à la double négation sur $(id - BID)$, le résultat sera soit zéro, soit un. Dans le cas où ce résultat est égal à un, dénotant une altération du flux de contrôle, il altérera la valeur de la variable id et cette altération sera détectée lors de l'arrivée dans le prochain bloc, mettant ainsi en évidence l'altération du flux de contrôle. Les détails de fonctionnement de cette techniques sont disponibles dans l'article [133].

6.2.2.2 Vérification formelle

ECCA est un peu plus complexe que CFCSS. Une fois le fonctionnement de la technique assimilé, nous avons implémenté ECCA dans chacun des cinq programmes génériques en trois heures, soit avec une moyenne d'environ 36 minutes par implémentation. La vitesse d'implémentation reste tout de même suffisamment élevée, dû à la taille réduite des programmes génériques. Bien entendu, comme pour

CFCSS, il est nécessaire de maîtriser préalablement le fonctionnement de la technique ainsi que la syntaxe du langage pseudo-assembleur.

Le tableau 6.3 présente les résultats de vérification concernant les transferts erronés causant une altération du flux de contrôle. Comme pour CFCSS, chaque colonne présente les résultats obtenus avec le programme générique dont le numéro est indiqué dans la case du haut, avec, entre parenthèses, le nombre d'instructions total du programme une fois la technique implémentée.

Prog. (inst.)	$\mathcal{P}_1$ (50)	$\mathcal{P}_2$ (56)	$\mathcal{P}_3$ (50)	$\mathcal{P}_4$ (91)	$\mathcal{P}_5$ (93)
% Altérations	99.89. %	99.67 %	99.94 %	99.99 %	99.96 %
Non détectées	3.12 %	50.02 %	17.14 %	50.81 %	63.99 %
Temps	0:01.06	0:11.21	0:01.12	0:27.13	0:56.46
Mémoire (MB)	38.291	48.531	38.495	60.102	82.937
Détectées	96.88 %	49.98 %	82.86 %	49.19 %	36.01 %
Temps	0:00.80	0:08.25	0:00.87	0:18.90	0:39.44
Mémoire (MB)	38.291	48.633	38.495	60.102	82.937

Table 6.3: ECCA – Erreurs causant une altération du flux de contrôle

Le tableau 6.4, quant à lui, présente les résultats de vérification concernant les transferts erronés ne causant pas d'altération du flux de contrôle.

Prog. (inst.)	$\mathcal{P}_1$ (50)	$\mathcal{P}_2$ (56)	$\mathcal{P}_3$ (50)	$\mathcal{P}_4$ (91)	$\mathcal{P}_5$ (93)
% Non-altérations	0.11%	0.33%	0.06%	0.01%	0.04%
Détectées	1.82 %	0.09 %	3.70 %	0.78 %	0.12 %
Temps	0:00.61	0:07.33	0:01.01	0:10.41	0:22.35
Mémoire (MB)	38.086	47.814	38.598	53.036	69.113
Non détectées	98.18 %	99.91 %	96.30 %	99.22 %	99.88 %
Temps	0:01.05	0:09.68	0:01.33	0:12.59	0:27.11
Mémoire (MB)	38.086	47.711	38.598	53.036	69.113

Table 6.4: ECCA – Erreurs ne causant pas d'altération du flux de contrôle

Note : comme pour CFCSS, il est important de situer les pourcentages du tableau 6.4 dans leur contexte, en fonction des pourcentages d'altérations du flux de contrôle.

6.2.2.3 Interprétation

Tout comme pour CFCSS, la vérification formelle d'ECCA nous permet de constater, à l'instar des résultats obtenus par ses auteurs, qu'elle n'est pas fiable. Nos résultats font état d'une moyenne de 37.02 % d'altérations du flux de contrôle non détectées par ECCA. Les auteurs de la technique, quant à eux, font état d'un taux de seulement 43.7 % de SEU détectés, donc 56.3 % ne l'étant pas, ce qui est beaucoup plus élevé que ce que nous détectons.

Néanmoins, les auteurs de la technique ne précisent pas le pourcentage des SEU causant des altérations du flux de contrôle. Les auteurs de l'article [132], qui précisent les taux d'erreurs altérant et n'altérant pas le flux de contrôle, présentent un taux de détection beaucoup plus proche du nôtre : 62.34 % des altérations détectées par ECCA, soit un taux de 37.66 % échappant à la détection.

Tout comme pour CFCSS, pour calculer cette moyenne, nous divisons chacun des pourcentages donnés par les différents bancs de test par le pourcentage d'erreurs affectant le flux de contrôle. Nous "normalisons" les résultats en les amenant à représenter des pourcentages de détection correspondant, non pas simplement SEU, mais précisément aux SEU altérant le flux de contrôle. Il ne reste plus alors qu'à additionner les différents résultats et à les diviser par leur nombre.

Comparativement au taux de 37.66 % de l'article [132], nous obtenons un taux 37.02 %, ce qui est légèrement plus bas. Ici encore, ceci est explicable par le fait que les programmes que nous utilisons sont très réduits et que leur ensemble ne capture probablement pas tous les types de transitions existant dans les programmes des bancs de test.

Cependant, les résultats obtenus sont bien entendu très proches et, de plus, la vérification formelle de la formule Φ_2, permettant de détecter les altérations échappant à ECCA s'effectue en 1 minute 37 secondes tandis que les temps de simulation présentés dans [132] font état de plusieurs heures de calcul, comme dans le cas de CFCSS. Les détails de fonctionnement de cette techniques sont disponibles dans l'article [135].

6.2.3 DSM – Dynamic Signature Monitoring

Troisième et dernière méthode de notre étude de cas, DSM [135] est, tout comme CFCSS et ECCA, une technique de tolérance aux pannes dédiée à la détection des altérations du flux de contrôle.

6.2.3.1 Description

Le principe de base des techniques classiques comme CFCSS et ECCA consiste à associer une signature de référence (*Reference signatures (RS)*) unique à chaque bloc d'un programme durant la phase de compilation. Durant l'exécution, les signatures en-ligne (*Online signatures (OS)*) sont comparées avec les signatures de référence pour détecter les altérations du flux de contrôle.

Contrairement aux approches classiques, DSM associe les signatures au transitions entre les blocs. La stratégie de suivi des signatures consiste à faire en sorte que la signature en ligne dépende de l'identificateur du bloc source, ainsi que de l'identificateur du bloc de destination. Le flux de contrôle est surveillé en comparant la signature en ligne à la signature de référence de la dernière transition exécutée (pour arriver au bloc présent).

En prime des signatures RS et OS, DSM introduit un ensemble de signatures additionnelles $N1$, $N2$, et $N3$, dites *signatures cumulatives locales*. Ces signatures sont des nombres entiers pré-compilés qui sont affectés aux blocs constituant le programme, l'ensemble des $N1$, $N2$, et $N3$, étant unique pour chaque bloc. Ces nombres sont additionnés durant l'exécution d'un bloc et, à la sortie du bloc, le résultat du calcul :

$$N = N1 + N2 + N3$$

doit être égal à 0.

Les endroits du programme dans lesquels les composants de N sont placés permettent de s'assurer que : a) le bloc source transfère bien le contrôle à la première instruction du bloc destination, et, b) que les instructions de suivi des signatures OS et RS sont exécutées dans le bon ordre. Ainsi, chaque fois que le

contrôle est transféré d'un bloc à un autre, la somme N doit être égale à zéro. Le cas contraire révèle une altération du flux de transfert.

6.2.3.2 Vérification formelle

DSM n'est pas plus complexe que ECCA. Son implémentation dans chacun des cinq programmes génériques s'est avérée aussi rapide que celle de CFCSS, soit environ deux heures, c'est-à-dire une moyenne de 20 minutes par programme. Le tableau 6.5 présente les résultats de vérification concernant les transferts erronés causant une altération du flux de contrôle.

Prog. (inst.)	$\mathcal{P}_1$ (82)	$\mathcal{P}_2$ (103)	$\mathcal{P}_3$ (85)	$\mathcal{P}_4$ (147)	$\mathcal{P}_5$ (154)
% Altérations	99.96 %	99.77 %	99.94 %	99.96 %	99.87 %
Non détectées	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
Temps	0:00.64	0:02.33	0:00.69	0:02.42	0:04.14
Mémoire (MB)	37.983	40.031	38.188	40.031	41.977
Détectées	100 %	100 %	100 %	100 %	100 %
Temps	0:00.60	0:02.20	0:00.65	0:02.24	0:03.92
Mémoire (MB)	37.983	40.134	38.188	40.134	42.182

Table 6.5: DSM – Erreurs causant une altération du flux de contrôle

Le tableau 6.4, quant à lui, présente les résultats de vérification concernant les transferts erronés ne causant pas d'altération du flux de contrôle.

Prog. (inst.)	$\mathcal{P}_1$ (82)	$\mathcal{P}_2$ (103)	$\mathcal{P}_3$ (85)	$\mathcal{P}_4$ (147)	$\mathcal{P}_5$ (154)
% Non-altérations	0.04 %	0.23 %	0.06 %	0.04 %	0.13 %
Détectées	11.53 %	1.31 %	11.53 %	4.83 %	1.44 %
Temps	0:00.62	0:03.65	0:01.02	0:02.69	0:07.48
Mémoire (MB)	37.983	41.055	38.495	40.441	43.923
Non détectées	88.47 %	98.69 %	88.47 %	95.17 %	98.56 %
Temps	0:00.75	0:04.67	0:01.35	0:03.55	0:09.76
Mémoire (MB)	37.983	40.953	38.393	40.441	43.923

Table 6.6: DSM – Erreurs ne causant pas d'altération du flux de contrôle

Note : comme pour CFCSS et ECCA, il est important de situer les pourcentages du tableau 6.4 dans leur contexte, en fonction des pourcentages d'altérations du

flux de contrôle.

6.2.3.3 Interprétation

Les résultats de vérification obtenus indiquent que DSM est fiable, c'est-à-dire qu'aucune altération du flux de contrôle n'échappe à sa détection. Ces résultats confirment ceux obtenus par simulation [135] par les auteurs de cette technique. Nous constatons, par contre, que DSM présente un taux de détection d'erreurs n'altérant pas le flux de contrôle beaucoup plus élevé que CFCSS et ECCA. Une fois de plus, la vérification de la formule Φ_2 nécessite des temps de calcul très court (moins de 10 secondes), bien moindre que ceux nécessités par les simulations.

6.2.4 Performances

Dans un premier temps, le graphique 6.2 présente l'évolution des taux d'altérations non détectés qui sont révélés par la vérification.

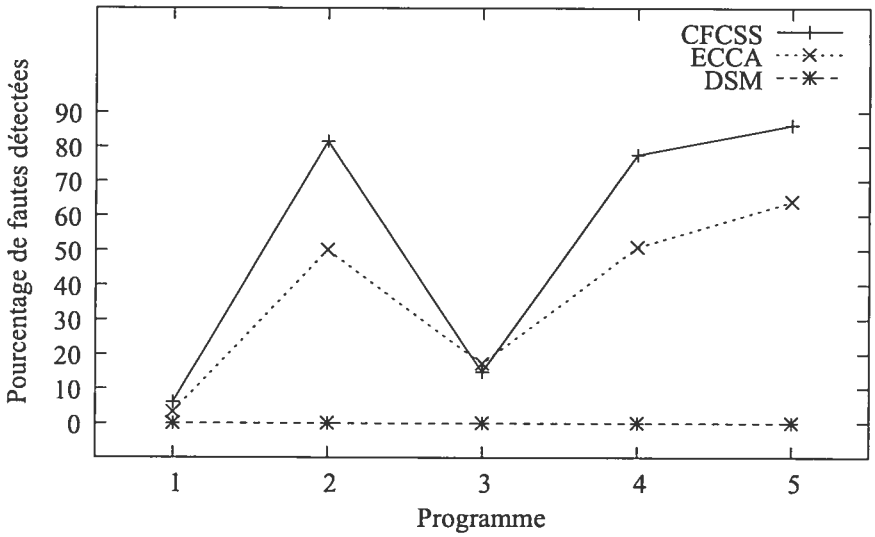

Figure 6.2: Altérations du flux non détectées

Nous constatons que les programmes $\mathcal{P}_1$ et $\mathcal{P}_3$ permettent de détecter des taux de fuites beaucoup moins élevés que ceux détectés par les trois autres programmes.

Ces deux programmes sont les deux dont le nombre de blocs ainsi que le nombre de transferts sont les moindres. Il serait, à priori, possible que cela ait une influence sur les taux de détection. Cependant, il nous semble que la raison principale réside surtout dans le fait que ces deux programmes n'implémentent que des transferts inconditionnels. En effet, c'est le cas de $\mathcal{P}_1$ et même si $\mathcal{P}_3$ implémente un appel de fonction, cet appel et son retour ne sont ni plus ni moins que des transferts inconditionnels. Il apparaît évident que l'ensemble des programmes génériques nécessaires à l'obtention de résultats probants, comme le confirment les comparaisons avec les résultat de simulation des sections précédentes.

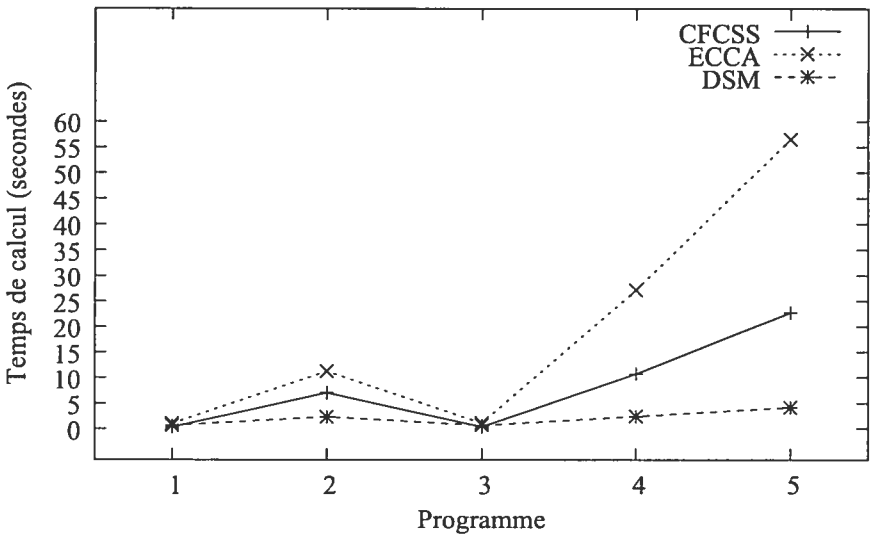

Figure 6.3: Temps de calcul

Nous constatons néanmoins que les temps de calcul, ainsi que les quantités de mémoire requise restent très raisonnables dans un contexte d'application de techniques de vérification formelle, et bien en dessous des taux standards des méthodes de simulation [132].

Il peut sembler surprenant que ce temps de calcul soit aussi bas par rapport à eux obtenus pour CFCSS et ECCA, surtout étant donné le fait que DSM soit beaucoup plus complexe. Cela est dû à l'extraction des scénarii. En effet, la vérification de la formule en elle même repose sur le produit synchronisé et son calcul prend plus de temps pour DSM que pour CFCSS et ECCA. Par contre,

une fois ce calcul effectué, l'extraction de tous les scénarii contre-exemples a pris beaucoup plus de temps pour les méthodes les moins fiables. Ainsi, le temps pris par CFCSS est plus important que celui pris par ECCA, ce dernier étant aussi plus important que celui pris par DSM pour laquelle SPIN n'a, de toute manière, pas à extraire de scénarii puisque la technique est fiable.

Sur le plan de l'utilisation de la mémoire, ainsi que du temps de calcul, nous constatons que ces taux évoluent similairement aux taux d'erreurs détectées. Nous observons que plus le programme est important en termes du nombre de blocs et de la richesse des types de transferts, plus ce dernier semble en mesure de détecter un nombre d'erreurs important et, de ce fait, plus il nécessite un temps de calcul et une quantité de mémoire élevés.

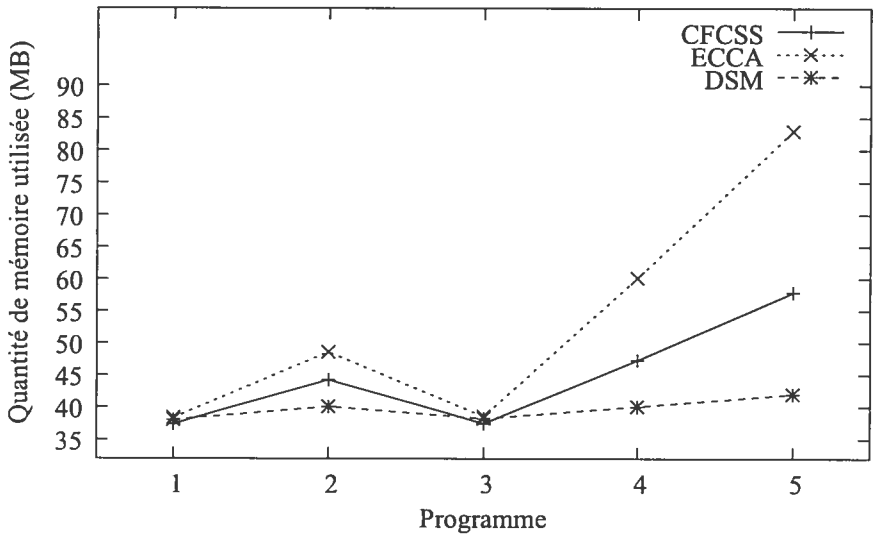

Figure 6.4: Quantité de mémoire utilisée

À titre indicatif, le temps de vérification total de l'ensemble des formules (Φ_1 à Φ_7) sur l'ensemble des techniques implémentées, soit quinze programmes, requiert un temps de calcul de mois de 20 minutes, tandis que la quantité de mémoire maximale utilisée lors d'une vérification ne dépasse pas 90 Megaoctets.

6.3 Résumé

Dans ce chapitre, nous avons présenté l'implémentation des concepts théoriques énoncés au chapitre 5. Cette implémentation consiste en un ensemble de programmes génériques ainsi qu'en un prototype permettant d'effectuer la vérification automatique complète d'un ensemble de technique de tolérance aux pannes implémentées dans les programmes génériques. Le fonctionnement de ce prototype, ainsi que la théorie sur laquelle ce dernier repose (énoncée au chapitre 5) sont appuyés par une étude de cas solide.

Notre étude de cas consiste en la vérification formelle de trois techniques de tolérance aux pannes : CFCSS, ECCA, et DSM. Les résultats de vérification par simulation obtenus par leurs auteurs respectifs montrent que les deux premières, CFCSS et ECCA ne sont pas fiables, tandis que DSM l'est. Les résultats que nous avons obtenus par vérification formelle concordent avec les résultats de simulation obtenus antérieurement. Les taux que nous obtenons par vérification formelle ne présentent qu'une différence minime de 0.5 % pour CFCSS et ECCA, et sont identiques aux résultats de simulation en ce qui concerne DSM.

D'autre part, l'implémentation d'une méthode dans un des graphes est très rapide dans la mesure où ces derniers sont très réduits. De 10 à 30 minutes pour une personne connaissant la méthode en question et le langage utilisé. L'ensemble du processus traduction-vérification est très performant, avec un temps d'exécution de l'ordre d'une vingtaine de minutes, contrairement à des simulations qui prendraient plusieurs dizaines d'heures [132].

L'implémentation de notre prototype, ainsi que les résultats obtenus avec notre étude de cas, confirment la pertinence des concepts théoriques énoncés au chapitre 5 et leur utilité pour la vérification de techniques de tolérance aux pannes.

CHAPITRE 7

CONCLUSIONS

La communauté s'accorde sur le fait qu'une élévation du niveau d'abstraction est nécessaire pour permettre une amélioration du cycle de conception. Dans cette optique, notre travail de recherche démontre, théorie et applications à l'appui, comment l'utilisation de méthodes formelles de haut niveau d'abstraction permet une amélioration drastique des capacités de conception et vérification, que ce soit à propos de la productivité ou sur le plan de la fiabilité. En effet, l'obtention de requis précoces sans erreurs permet de dériver plus rapidement des spécifications de meilleure qualité. Les méthodes que nous avons proposées sont complémentaires de celles connues à ce jour, il ne s'agit pas de remplacer certaines phases du cycle de développement par d'autres, mais de miser sur l'association des techniques applicables aux différentes phases de développement pour ainsi tirer le meilleur de chacune.

Cette thèse porte sur l'utilisation des méthodes formelles pour la conception de systèmes électroniques fiables. La croissance exponentielle de la taille et de la complexité des circuits électroniques d'aujourd'hui pose de plus en plus de problèmes, particulièrement en ce qui a trait à leur conception. Les méthodes actuelles ne sont plus suffisantes pour permettre le développement de systèmes aussi complexes et la fiabilité, ainsi que la productivité, de ces derniers en est dégradée. En effet, les méthodes de vérification actuelles se heurtent à des problèmes d'explosion combinatoire dans le cas des méthodes formelles, ou à des temps de calcul beaucoup trop longs dans le cas des méthodes de simulation.

7.1 Contributions – Ingénierie des exigences

En ce qui concerne l'ingénierie des exigences, tel que mentionné dans le chapitre 1, nos travaux ont donné lieu à deux publications dans des conférences interna-

tionales [10] [11] ainsi qu'un article récemment soumis à une revue scientifique [12].

Représentation – Nous avons défini un cadre formel de représentation des exigences. Ce cadre fournit aux concepteurs un modèle de représentation des fonctionnalités décrites dans les documents d'exigences. Les fonctionnalités sont décrites sous forme d'actions structurée dont les pré- et post-conditions sont écrites sous forme de phrases dans un anglais restreint. Un tel format est facile d'utilisation et permet non seulement une amélioration de la documentation relative aux exigences, mais aussi un meilleur partage des documents entre concepteurs. Les bénéfices d'une telle contribution s'accompagnent bien entendu d'une amélioration de la productivité.

Formalisation – Nous avons développé un mécanisme de formalisation automatique des phrases représentant les pré- et post-conditions. Ce mécanisme se base sur un ensemble de principes linguistiques solides et permet de transformer les phrases données par les concepteurs sous forme prédicative. Le résultat est une base de donnée de fonctionnalités totalement formelle propice à être vérifiée formellement.

Cohérence – Nous avons développé un ensemble de règles de cohérence permettant de s'assurer que la base de fonctionnalités formalisées (automatiquement ou directement écrite formellement par les concepteurs) ne comporte pas de problèmes de cohérence. Cet ensemble de règles permet une détection automatique des problèmes très tôt dans le cycle de développement, résultant en une réduction des coûts de vérification dans les phases subséquentes, ayant ainsi un impact sur la qualité ainsi que sur la productivité.

Complétude – Nous avons développé une méthode d'extraction des fonctionnalités manquantes basée sur la logique booléenne. L'emploi de ce mécanisme permet aux concepteurs de s'assurer que l'ensemble des fonctionnalités ayant été définies est complet. Si tel n'est pas le cas, la liste des configurations de pré-conditions pour lesquelles aucune action n'est spécifiée est donnée aux

concepteurs, leur permettant ainsi de combler les manques. Tout comme la phase de vérification de cohérence, cette phase a un impact sur la qualité des spécifications qui seront dérivées dans les phases subséquentes puisqu'elle permet de détecter les manques dès le début du cycle.

Outils – Nous avons implémenté les concepts théoriques de notre méthodologie sous forme d'un ensemble de modules utilisés par un programme principal. Ce prototype permet un traitement automatique des exigences représentées sous forme semi-formelle par les concepteurs. Nous fournissons ainsi, non seulement un ensemble de concepts théoriques, mais un outil les implémentant.

Applications – Finalement, nous avons testé notre implémentation sur un ensemble d'études de cas pertinentes, fournissant ainsi une évaluation complète et détaillée des concepts développés. Les résultats obtenus confirment la pertinence de notre approche.

7.2 Contributions – Vérification formelle des techniques de tolérance aux pannes

Tel que mentionné dans le chapitre 1, l'ensemble de nos travaux sur la vérification des techniques de tolérance aux pannes à donné lieu à une publication dans une conférence internationale [13] et une publication dans une revue scientifique [14]. Un troisième article est en préparation en vue d'être soumis à une revue scientifique [15].

Programmes génériques – Nous avons fourni un ensemble de programmes génériques représentant les différents transferts caractéristiques qui peuvent exister dans les programmes usuels. Chaque graphe capture un minimum de caractéristiques, permettant ainsi d'évaluer les techniques de manière incrémentale.

Pseudo-assembleur – Nous avons développé un langage pseudo-assembleur dédié à l'implémentation des techniques au sein des programmes génériques. Sorte

d'assembleur étendu, ce langage somme toute très simple d'utilisation permet une implémentation très rapide des techniques à évaluer.

Formalisation – Nous avons fourni un ensemble de mécanismes de traduction permettant de générer un modèle formel PROMELA à partir d'un programme en pseudo-assembleur, permettant ainsi de procéder à des vérifications formelles à l'aide du *model-checker* SPIN.

Vérification – Nous avons fourni un ensemble de formules LTL formalisant les propriétés devant être respectées par toute technique de tolérance aux pannes. Cet ensemble de propriétés est utilisé en conjonction avec SPIN pour effectuer la vérification formelle du modèle PROMELA.

Outils – Nous avons implémenté les concepts théoriques sous forme d'un ensemble d'outils autonomes. Ces outils permettent une automatisation complète de la vérification des techniques implémentées dans les programmes génériques.

Applications – Finalement, nous avons appuyé notre méthodologie en effectuant la vérification de trois techniques de tolérance aux pannes et en comparant nos résultats avec des résultats de simulation obtenus ultérieurement par des équipes d'autres équipes de recherche. Les résultat obtenus en quelques minutes concordent à 0.5% près avec les résultats de référence obtenus en plusieurs heures, confirmant ainsi non seulement le bon fonctionnement de notre approche mais aussi son efficacité.

7.3 Travaux futurs

Nombre de travaux futurs sont envisageables. Parmi les plus prometteurs nous envisageons en premier lieu de greffer nos approches à des méthodologies existantes.

Dans le cadre des exigences, nous avons entamé une collaboration avec Alexandre Chureau dans le cadre de ses travaux sur l'application de la méthodologie UML au cycle de développement matériel-logiciel [88]. La méthodologie développée par

A. Chureau prend en entrée un ensemble d'exigences. Or, ces exigences sont exprimées à un niveau d'abstraction très élevé et le fossé entre cette représentation et UML reste très important. L'ajout de notre méthodologie entre les exigences et la méthodologie développée par A. Chureau devrait permettre de fournir un ensemble de fonctionnalités formalisées et testées en entrée de la méthodologie UML, permettant ainsi de réduire le fossé tout en fournissant un ensemble d'exigences pré-vérifiées.

Toujours dans le cadre des exigences, nous nous penchons sur l'intégration de notre approche au sein du l'outil de simulation ESys.Net [32] [33] développé au sein du laboratoire LASSO. Il s'agirait, là encore, de greffer notre approche au dessus de la méthodologie de développement utilisée par les concepteurs se servant d'ESys.Net. En outre, nous voyons la possibilité de dériver, à partir des fonctionnalités de haut niveau, des ensembles d'observateurs pouvant être raffinés et utilisés au sein de la couche de vérification d'ESys.Net [34].

Frédéric Bastien, étudiant de maîtrise au LASSO à repris nos travaux sur l'ingénierie des exigences. Il travaille actuellement sur la modélisation et la vérification du protocole de communication du bus AMBA [136] tout en cherchant à améliorer notre méthodologie.

Au niveau de la vérification des techniques de tolérance aux pannes, l'axe nous paraissant le plus prometteur, et le plus sensible, concerne les programmes génériques. En effet, si ces derniers permettent de connaître les taux d'erreurs d'une technique en évaluation, ils restent limités quant à l'observation du comportement d'une technique en fonction des différentes configurations de transferts. Nous nous penchons actuellement sur la possibilité d'accroître le nombre de programmes en définissant des configurations particulières telles que des appels de fonction récurrents de manière à définir des sous-groupes de programmes dédiés à tester non pas certains transferts types mais certaines combinaisons de transferts types.

Un autre axe possible concerne le développement d'un logiciel de conception et vérification des techniques de tolérance aux pannes qui soit d'envergure industrielle.

Basé sur une interface graphique, le concepteur pourrait implémenter sa technique dans les différents graphes génériques et visualiser directement les résultats.

RÉFÉRENCES

[1] International Technology Roadmap for Semiconductor. 2004 Edition. In *http://public.itrs.net/Files/2003ITRS/Home2003.htm*, 2004.

[2] L. Macaulay. Requirements for Requirements Engineering Techniques. In *Second International Conference on Requirements Engineering*, 1996.

[3] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Conference on The future of Software engineering*, 2000.

[4] T. Kropf. *Introduction to Formal Hardware Verification.* Springer-Verlag, 1999.

[5] E. Normand. Single Event Effects in Avionics. In *IEEE Trans. On Nuclear Science, vol. 43, n. 2*, 1996.

[6] N. A. Touba and E. J. McCluskey. Logic Synthesis of Multilevel Circuits with Concurrent Error Detection. In *IEEE Trans. CAD, vol. 16, n. 7*, 1997.

[7] R. Velazco, D. Bessot, R. Eccofet, and S. Duzellier. Two CMOS memory cells suitable for the design of SEU tolerant VLSI circuits. In *IEEE Transactions on Nuclear Science, vol. 6, n. 41*, 1994.

[8] N. Gorse. The Feature Interaction Problem: Automatic Filtering of Incoherences & Generation of Validation Test Suites at the Design Stage. Master's thesis, University of Ottawa, Ottawa, 2001.

[9] N. Gorse, L. Logrippo, and J. Sincennes. Formal Detection of Feature Interactions with Logic Programming and LOTOS. In *Journal of Software and System Modeling*, November 2005.

[10] N. Gorse, E.M. Aboulhamid, and Y. Savaria. Consistency Validation of High-Level Requirements. In *4th IEEE International Workshop on System-on-Chip for Real-Time Applications*, July 2004.

[11] N. Gorse, P. Belanger, E.M. Aboulhamid, and Y. Savaria. Mixing Linguistic and Formal Techniques for High-Level Requirements Engineering. In *16th IEEE International Conference on Microelectronics*, December 2004.

[12] N. Gorse, P. Belanger, E.M. Aboulhamid, and Y. Savaria. A High-Level Requirements Engineering Methodology for Electronic System-Level Design. In *submitted to Special issue of the International Journal on Computers in Electrical Engineering*, June 2005.

[13] B. Nicolescu, N. Gorse, Y. Savaria, E.M. Aboulhamid, and R. Velazco. Validating a Dynamic Signature Monitoring Approach Using the LTL Model Checking Technique. In *Radiation Effects on Components and Systems Workshop*, September 2004.

[14] B. Nicolescu, N. Gorse, Y. Savaria, E.M. Aboulhamid, and R. Velazco. On the Use of Model Checking for the Verification of a Dynamic Signature Monitoring Approach. In *IEEE Transactions on Nuclear Science*, June 2005.

[15] N. Gorse, B. Nicolescu, E.M. Aboulhamid, and Y. Savaria. Vérification formelle des techniques de tolérance aux pannes – Une méthodologie complète reposant sur la Logique Linéaire Temporelle (LTL). In *Soumis à Technique et Science Informatiques*, Janvier 2005.

[16] S. Sutherland, D. Davidmann, and P. Flake. *SystemVerilog For Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Kluwer Academic, 2003.

[17] J. Bhasker. *A SystemC primer.* Star Galaxy Publishing, 2002.

[18] S. Palnitkar. *Verilog HDL, 2nd Ed.* Prentice Hall, 2003.

[19] S. Yalamanchili. *Introductory VHDL, From Simulation to Synthesis.* Prentice Hall, 2001.

[20] B. Berar et al. *Systems and Software Verification, Model-Checking Techniques and Tools.* Springer-Verlag, 2001.

[21] A. Donlin. Transaction Level Modeling: Flows and Use Models. In *IEEE Intl. Conf. on Hardware-Software Codesign and System Synthesis*, 2004.

[22] T. Givargis F. Vahid. *Embedded System design: A unified Hardware-Software introduction.* Wiley, 2002.

[23] Model Technology Inc. *ModelSim.* http://www.model.com, 2005.

[24] Aldec. *Active-HDL.* http://www.aldec.com, 2005.

[25] Xilinx. *Xilinx Webpac.* http://www.xilinx.com, 2005.

[26] Synopsys. *VCS.* http://www.synopsys.com, 2005.

[27] A. Siebenborn, O. Bringmann, , and W. Rosenstiel. Worst-Case Performance Analysis of Parallel, Communicating Software Processes. In *International Conference on Hardware/Software Codesign and System Synthesis*, 2002.

[28] F. Bruschi, M. Chiamenti, F. Ferrandi, and D. Sciuto. Error Simulation Based on the SystemC Design Description Language. In *Design, Automation, and Test in Europe Conference*, 2002.

[29] L. Cai, D. Gajski, P. Kritzinger, and M. Olivares. Top-Down System Level Design Methodology Using SpecC, VCC and SystemC. In *Design, Automation, and Test in Europe Conference*, 2002.

[30] Jon Connell and Bruce Johnson. Early HW/SW Integration Using SystemC v2.0. In *Embedded Systems Conference*, 2002.

[31] Confluence System Design Language. *http://www.confluent.org/.* 2004.

[32] J. Lapalme, E.M. Aboulhamid, G. Nicolescu, L. Charest, F.R. Boyer, J.P David, and G. Bois. .NET framework - a solution for the next generation

tools for system-level modeling and simulation. In *Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[33] E-Sys.Net web site. *http://www.esys-net.org/*. 2003.

[34] N. Gorse, M. Metzger, J. Lapalme, E.M. Aboulhamid, Y. Savaria, and G. Nicolescu. Enhancing ESys.Net with a Semi-Formal Verification Layer. In *16th IEEE International Conference on Microelectronics*, December 2004.

[35] E.A. Emerson. Temporal and Modal Logic. In *Handbook of theoretical computer science (vol. B): formal models and semantics.* MIT Press, 1991.

[36] M.R.A. Huth and M.D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2000.

[37] K.J. Turner. *Using Formal Description Techniques.* J. Wiley & sons Ltd, 1993.

[38] T. Bolognesi and E. Briskma. *Introduction to the ISO Specification Language LOTOS.* Elsevier Science, 1989.

[39] L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. In *Computer Networks and ISDN Systems 23(5).* Elsevier, 1992.

[40] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[41] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[42] W. Hassan. DLOTOS - A LOTOS Extension for Clock Synchronization in Distributed Systems. In *2nd Asia-Pacific Conference on Quality Software.* IEEE Computer Society, 2001.

[43] M. Faci and L. Logrippo. Specifying Hardware Systems in LOTOS. In *11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, 1993.

[44] H. Kahlouche, C. Viho, , and M. Zendri. Hardware Testing using a Communication Protocol Conformance Testing Tool. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[45] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models.* Kluwer Academic Publishers, 2000.

[46] F.I. Haque, K.A. Khan, and J. Michelson. *The art of verification with VERA.* Verification Central Fremont, 2001.

[47] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design.* Verification Central Fremont, 2001.

[48] L. Blanc, A. Bouali, J. Dormoy, and O. Meunier. A Methodology for SoC Top-Level Validation using Esterel Studio. In *Electronic Design Processes 2002*, 2002.

[49] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design on VLSI in Computer & Processors*, 1992.

[50] Synopsys Inc. *Whitepaper: Assertion-Based Verification.* 2003.

[51] R.G. Stolzman. Understanding Assertion-Based Verification - http://www.techonline.com/community/related_content/21077. TechOnline publication, 2002.

[52] OpenVera Assertions LRM. *http://www.open-vera.com/technical/OVAIPGuidelines.pdf.* 2003.

[53] Synopsys Inc. *OpenVera Assertions (OVA) and ForSpec - Technical Language Specification.* 2002.

[54] Synopsys Inc. *OpenVera Language Reference Manual.* 2002.

[55] Sasan Iman and Sunita Joshi. *The E Hardware Verification Language.* Kluwer Academic Publisher, 2004.

[56] Accellera Inc. PSL LRM. http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf, 2003.

[57] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *16th IEEE International Conference on Automated Software Engineering*, 2001.

[58] O. Drissi-Kaitouni and C. Jard. Compiling temporal logic specifications into observers. In *Technical report, Institut National de Recherche en Informatique et en Automatique, Frankreich*, 1988.

[59] P. Gastin and D. Oddoux. Fast LTL to Buchi Automata Translation. In *13th conference on Computer Aided Verification*, 2001.

[60] C. Kern and M.R. Greenstreet. Formal Verification In Hardware Design - A Survey. In *ACM Transactions on Design Automation of Electronic Systems vol. 4, n. 2*, 1999.

[61] F. Hon and D.K. Probst. *Advances in Hardware Design and Verification.* Chapman & Hall, 1997.

[62] P. Rashinkar, P. Paterson, and L. Singh. *System-on-chip Verification, Methodology and Techniques.* Kluwer Academic Publishers, 2001.

[63] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. *Symbolic Model-checking for Sequential Circuit Verification.* 1993.

[64] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model-checking: 10E20 states and beyond. In *LICS*, 1990.

[65] Cadence Design Systems. *FormalCheck User's Guide v.2.3.* 1999.

[66] G. J. Holzmann. The SPIN Model Checker. *IEEE Transactions on Software Engineering.* 1997.

[67] E.A. Lee and A. Sangiovanni-Vincentelli. Comparing models of computation. In *IEEE/ACM international conference on Computer-aided design*, 1996.

[68] L. Wos, R.A. Overbeek, E.L. Lusk, and J. Boyle. *Automated Reasonning Introduction and Applications.* Prentice-Hall, 1984.

[69] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* New-York Academic, 1973.

[70] B. Rahbaran, , A. Steininger, , and T. Handl. Built-in fault injection in hardware - the FIDYCO example. In *Electronic Design, Test and Applications, 2004. DELTA 2004. Second IEEE International Workshop on*, 28-30 Jan. 2004 Page(s):327 - 332.

[71] M. Garcia-Valderas, C. Lopez-Ongil, M. Portela-Garcia, and L. Entrena-Arrontes. Transient fault emulation of hardened circuits in FPGA platforms. In *On-Line Testing Symposium, 2004. IOLTS 2004. Proceedings. 10th IEEE International*, 12-14 July 2004 Page(s):109 - 114.

[72] C.R. Yount and D.P. Siewiorek. A methodology for the rapid injection of transient hardware errors. In *Computers, IEEE Transactions on*, Volume 45, Issue 8, Aug. 1996 Page(s):881 - 891.

[73] R.R. Some, W.S. Kim, G. Khanoyan, L. Callum, A. Agrawal, and J.J. Beahan. A software-implemented fault injection methodology for design and validation of system fault tolerance. In *Dependable Systems and Networks, 2001. Proceedings. The International Conference on*, 1-4 July 2001 Page(s):501 - 506.

[74] P. Gawkowski and J. Sosnowski. Experimental validation of fault detection and fault tolerance mechanisms. In *High-Level Design Validation and Test*

Workshop, 2002. Seventh IEEE International, 27-29 Oct. 2002 Page(s):181 - 186.

[75] D. Gil, R. Martinez, J. V. Busquets, J. C. Baraza, and P. J. Gil. Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System. In *Dependable Computing EDCC-3*, 1999.

[76] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault Injection into VHDL Models: The MEFISTO Tool. In *Fault-Tolerant Computing, FTCS-24*, 1994.

[77] J. Guethoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Symp. on Fault-Tolerant Computing*, 1995.

[78] A. Amendola, P. Marmo, and F. Poli. Experimental evaluation of computer-based railway control systems. In *Symp. on Fault-Tolerant Computing*, 1997.

[79] H. Saiedian, P. Kumarakulasingam, and M. Anan. Scenario-based requirements analysis techniques for real-time software systems - a comparative evaluation. In *Requirements Engineering Journal*, 2004.

[80] D. Hazel, P. Strooper, and O. Trayno. Requirements Engineering and Verification using Specification Animation. In *Thirteenth IEEE Conference on Automated Software Engineering*, 1998.

[81] J.M. Spivey. An introduction to Z and formal specifications. In *Software Engineering Journal, vol. 4*, 1989.

[82] H. Zhu and L. Jin. Scenario Analysis in an Automated Tool for Requirements Engineering. In *Requirements Engineering Journal vol. 5, n. 1*, 2000.

[83] P. Heymans and E. Dubois. Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements. In *Requirements Engineering Journal vol. 3. No. 3-4*, 1998.

[84] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *17th IEEE Internation Conference on Automated Software Engineering*. IEEE Computer Society, 2002.

[85] W. Shen, K. Compton, and J. Huggins. A UML Validation Toolset Based on Abstract State Machines. In *16th IEEE International Conference on Automated Software Engineering*, 2001.

[86] P. B. Carpenter. Verification of requirements for safety-critical software. In *Annual ACM SIGAda international conference on Ada*, 1999.

[87] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. High level and architectural synthesis: An object-oriented design process for system-on-chip using UML. In *15th international symposium on System Synthesis*, 2004.

[88] A. Chureau, Y. Savaria, and E.M. Aboulhamid. The Role of Model-Level Transactors and UML in Functional Prototyping of Systems-on-Chip: A Software-Radio Application. In *Design, Automation and Test in Europe conference*, 2005.

[89] M. Glinz. Problems and Deficiencies of UML as a Requirements Specification Language. In *10th International Workshop on Software Specification and Design*, 2000.

[90] S.P. Overmyer, B. Lavoie, and O. Rambow. Conceptual modeling through linguistic analysis using LIDA. In *23rd international conference on Software engineering*, 2001.

[91] H.G. Perez-Gonzalez and J.K. Kalita. GOOAL: a Graphic Object Oriented Analysis Laboratory. In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2002.

[92] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Applications of linguistic techniques for use case analysis. In *Requirements Engineering Journal. vol. 8. No. 3*, 2003.

[93] D. Richards. Merging individual conceptual models of requirements. In *Requirements Engineering vol. 8, n. 4*, 2003.

[94] A. Holt, E. Klein, and C. Grover. Natural language for hardware verification - semantic interpretation and model-checking. In *Inference in Computational Semantics: Institute for Logic, Language and Computation*, 1999.

[95] L.A. Dennis, G. Collins, and M. Norrish et al. The PROSPER Toolkit. In *International Journal on Software Tools for Technology Transfer vol. 4, n. 2*, 2003.

[96] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1997.

[97] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. In *Requirements Engineering Journal vol. 9. No. 2*, 2004.

[98] C.P. Fuhrman. Lightweight models for interpreting informal specifications. In *Requirements Engineering vol. 8, n. 4*, 2003.

[99] K.S. Barber, T. J. Graser, J. Holt, and G. Baker. Arcade - early dynamic property evaluation of requirements using partitioned software architecture models. In *Requirements Engineering vol. 8, n. 4*, 2003.

[100] A. Duran, A. Ruiz-Cortes, R. Corchuelo, and M. Toro. Supporting Requirements Verification Using XSLT. In *IEEE Joint International Conference on Requirements Engineering*, 2002.

[101] B. Benz and J.R. Durant. *XML Programming Bible.* Wiley, 2005.

[102] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: from design exploration to exhaustive fault simulation. In *Dependable Systems and Networks, 2004 International Conference on*, 28 June-1 July 2004 Page(s):189 - 198.

[103] T. Yokogawa, T. Tsuchiya, and T. Kikuno. Automatic verification of fault tolerance using model checking. In *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, 17-19 Dec. 2001 Page(s):95 - 102.

[104] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. In *IEEE transactions on software engineering*, 1993.

[105] A. Arazo and Y. Crouzet. Formal guides for experimentally verifying complex software-implemented fault tolerance mechanisms. In *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on*, 11-13 June 2001 Page(s):69 - 79.

[106] H. Volzer. Verifying fault tolerance of distributed algorithms formally-an example. In *Application of Concurrency to System Design, 1998. Proceedings., 1998 International Conference on*, 23-26 March 1998 Page(s):187 - 197.

[107] J. L. Peterson. An Introduction to Petri Nets. In *Proc. of the National Electronics Conference, Chicago, Illinois*, pages 144–148. National Engineering Consortium, Inc., 1978.

[108] Jr. Kljaich, J., B.T. Smith, and A.S. Wojcik. Formal verification of fault tolerance using theorem-proving techniques. In *Computers, IEEE Transactions on*, Volume 38, Issue 3, March 1989 Page(s):366 - 376.

[109] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault injection for formal testing of fault tolerance. In *Reliability, IEEE Transactions on*, Volume 45, Issue 3, Sept. 1996 Page(s):443 - 455.

[110] M. Zhiming Liu Joseph. Verification of fault tolerance and real time. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, 25-27 June 1996 Page(s):220 - 229.

[111] S. Ayache, E. Conquet, P. Humbert, C. Rodriguez, J. Sifakis, and R. Gerlich. Formal methods for the validation of fault tolerance in autonomous spacecraft. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, 25-27 June 1996 Page(s):353 - 357.

[112] L. Aqvist. *Introduction to Deontic Logic and the Theory of Normative Systems.* Bibliopolis, 1983.

[113] F. Dignum and R. Kuiper. Combining dynamic deontic logic and temporal logic for the specification of deadlines. In *30th Hawaii International Conference on System Sciences (HICSS)*, 1997.

[114] M. Barbuceanu, T. Gray, and S. Mankowski. How To Make Your Agents Fulfil Their Obligations. In *Third International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, 1998.

[115] M. Barbuceanu, T. Gray, and S. Mankowski. Providing Telecommunication Services Through Multi-Agent Negotiation. In *Third International Workshop on Intelligent Agents for Telecommunication Applications*, 1999.

[116] P. Roussel. *Prolog: Manuel de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle, Universite de Marseille, France, 1975.

[117] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1997.

[118] F. Pereira and D.H.D. Warren. Definite Clause Grammars for Language Analysis - a Survey of the Formalism and a Comparison with Augmented Transition Networks. In *Artificial Intelligence Journal, Vol. 13*, 1980.

[119] A. Gal, G. Lapalme, and P. Saint-Dizier. Prolog pour l'Analyse Automatique du Langage Naturel. In *Eyrolles*, 1989.

[120] A. Mel'cuk. Dependency in Linguistic Description. http://www.olst.umontreal.ca/FrEng/Dependency.pdf.

[121] N. Gorse. *Thèse de Doctorat - Applications.* http://ngorse.free.fr/~gorsen/these/, 2005.

[122] R.K. Brayton, G.D. Hatchel, C.T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[123] RapidIO organization. *RapidIO document specifications.* http://www.rapidio.org/, 2005.

[124] A Tanenbaum. *Reseaux, Ed. 4.* Pearson Education, 2003.

[125] ITU. *Recommendation Z.120: Message Sequence Chart (MSC).* 1996.

[126] R.J. Trudeau. *Introduction to Graph Theory.* Dover Publications, 1993.

[127] E.J. McCluskey N. Oh, P.P. Shirvani. Control-Flow Checking by Software Signatures. In *IEEE Transactions on Reliability*, 2002.

[128] C. Newham and B. Rosenblatt. *Learning the BASH shell.* O'Reilly, 1998.

[129] R. Mecklenburg. *Managing Projects with GNU Make, Ed. 3.* O'Reilly, 2004.

[130] A. Robbins and D. Gilly. *Unix in a Nutshell, Ed. 3.* O'Reilly, 1999.

[131] J.R. Levine, T. Mason, and D. Brown. *Lex and Yacc, Ed. 2.* O'Reilly, 1992.

[132] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error Detection Using Control Flow Assertions. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[133] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. In *IEEE Transactions on Parallel and Distributed Systems*, 1999.

[134] L. McFearing and V.S.S. Nair. Control-Flow Checking Using Assertions. In *IEEE International Symposium on Parallel and Distributed Systems*, 1995.

[135] B. Nicolescu, Y. Savaria, and R. Velazco. Software Detection Mechanisms Providing Full Coverage Against Single Bit-flip Faults. In *IEEE Transactions on Nuclear Science, vol. 51, n. 6*, 2004.

[136] ARM. *ARM AMBA 2.0 Specification.* http://www.arm.com/products/solutions/AMBA_Spec.html, 2005.