Titre: Title:	Méthode de réutilisation et de couverture pour la vérification fonctionnelle des circuits numériques
Auteur: Author:	Sébastien Regimbal
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Regimbal, S. (2003). Méthode de réutilisation et de couverture pour la vérification fonctionnelle des circuits numériques [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7300/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7300/
Directeurs de recherche: Advisors:	Yvon Savaria, & Guy Bois
Programme: Program:	Unspecified

UNIVERSITÉ DE MONTRÉAL

MÉTHODE DE RÉUTILISATION ET DE COUVERTURE POUR LA VÉRIFICATION FONCTIONNELLE DES CIRCUITS NUMÉRIQUES.

SÉBASTIEN REGIMBAL
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
DÉCEMBRE 2003

Your file Votre référence
ISBN: 0-612-90856-9
Our file Notre référence
ISBN: 0-612-90856-9

UNIVERSITÉ DE MONTRÉAL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MÉTHODE DE RÉUTILISATION ET DE COUVERTURE
POUR LA VÉRIFICATION FONCTIONNELLE DES CIRCUITS NUMÉRIQUES

présenté par: REGIMBAL Sébastien
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BOYER François, Ph.D., président
M. SAVARIA Yvon, Ph.D., membre et directeur de recherche
M. BOIS Guy, Ph.D., membre et codirecteur de recherche
M. KHOUAS Abdelhakim, Doct., membre

REMERCIEMENTS

En premier lieu, je tiens à remercier mon directeur de recherche Yvon Savaria pour m'avoir proposé ce projet motivant qui m'a permis de repousser les limites de mes connaissances. Avec son soutien et ses conseils avisés, j'ai été en mesure de compléter aisément ce projet. J'aimerais également remercier mon codirecteur Guy Bois, qui m'a aussi supporté dans ce projet. De plus, je remercie les organismes, la société PMC Sierra, Micronet et le CRSNG, qui ont subventionné mon travail de recherche.

Je remercie Jean-François Lemire, mon partenaire dans l'exploration de ce domaine de recherche, qui a été directement impliqué dans la réalisation de ce projet. Nos discussions en toutes circonstances m'ont permis de progresser. Je suis aussi reconnaissant envers les membres du GRM et le personnel du GRM qui m'ont permis de travailler dans un environnement stimulant. J'aimerais aussi remercier André Baron pour sa collaboration à ce projet.

Finalement, une attention particulière va pour ma copine Chantal Galipeau qui me soutient dans tout ce que je fais.

RÉSUMÉ

Les méthodes et outils de conception de circuits numériques permettent le développement de systèmes complexes. Conséquemment la vérification fonctionnelle de tels systèmes devient de plus en plus difficile. D'autre part, puisque le coût de la fabrication des circuits a augmenté parallèlement à l'augmentation de la complexité des circuits, une erreur fonctionnelle détectée suite à sa fabrication engendre des coûts qui peuvent affecter la rentabilité commerciale d'un circuit. Il vaut donc mieux vérifier les fonctionnalités des circuits tout en respectant les échéanciers imposés par leur mise en marché. Ainsi, l'élaboration de nouvelles méthodologies, visant à accélérer les tâches de vérification et à augmenter la qualité de la vérification est essentielle afin de s'adapter aux contraintes actuelles régissant le développement des circuits numériques.

Ce mémoire apporte une contribution au niveau des méthodes de vérification fonctionnelle en introduisant deux méthodologies de vérification. La première méthode proposée est une méthode de conception de bancs d'essais favorisant la réutilisation. La méthode est basée sur l'utilisation d'un langage de vérification matérielle, le langage *e*, qui permet l'utilisation des méthodologies de conception orientées objets et aspects. Ainsi, un partitionnement orienté objets et aspects, dédié à la création de bancs d'essais, est proposé afin d'augmenter le potentiel de réutilisation d'un banc d'essais. Cette méthode permet d'accélérer le processus de conception des bancs d'essais. La deuxième méthode de vérification proposée est une technique systématique permettant de développer un module d'analyse de la couverture fonctionnelle d'un circuit numérique. La problématique reliée au développement de cette méthode est qu'il faut avoir des métriques fonctionnelles standardisées appliquées sur une description normalisée des fonctionnalités du circuit. Les fonctionnalités d'un circuit étant habituellement décrites sous forme textuelle dans un document de spécification, il est difficile de travailler de façon systématique avec ce genre de description. Ainsi, on propose d'effectuer une description standardisée de la spécification d'un circuit en utilisant un langage exécutable de haut niveau. En formulant une métrique fonctionnelle en fonction de la description

standardisée de la spécification, il est possible de créer un module d'analyse de la couverture de façon systématique. Le langage de spécification utilisé est le *Specification Description Language* (SDL). Ce langage est exécutable, ce qui permet de valider la spécification développée. La métrique fonctionnelle présentée dans ce mémoire est basée sur la technique de vérification des flots transactionnels. L'implémentation d'un module d'analyse de la couverture, en utilisant la méthode proposée, est faite en utilisant le langage *e*. Le module d'analyse de la couverture produit avec la méthode peut être incorporé dans n'importe quel banc d'essais dédié à la vérification d'un circuit au niveau *Register Transfer Level* (RTL). L'utilité de ce module est en fait de permettre la création d'une suite de tests qui permettra de couvrir la métrique définie et aussi de réduire l'application de tests redondants. Cela implique qu'avec ce module d'analyse de la couverture, il est possible de créer une suite de tests moins longue ayant aussi une plus grande puissance de détection d'erreurs

Ces méthodes ont été appliquées sur différents modèles. La méthode de conception a été utilisée par la création de bancs d'essais dédiés à la vérification d'une version d'un convertisseur de protocoles. Le projet de conversions de protocoles est développé par un groupe d'étudiants de cycles supérieurs du *Groupe de Recherche en Microélectronique* (GRM). Ensuite, la méthode de conception a aussi été utilisée pour la création d'un banc d'essais dédiés à un exemple simple d'un commutateur *Asynchronous Transfer Mode* (ATM). Le commutateur ATM a aussi servi d'exemple pour l'application de la méthode de couverture fonctionnelle. Ainsi, une spécification décrite avec le langage SDL a été développée pour le commutateur ATM. Ensuite, en utilisant la méthode de couverture développée, un module d'analyse de la couverture a été produit. En utilisant le module produit, il a été possible de développer une suite de tests efficaces qui couvre la métrique utilisée. Finalement, un module d'analyse de la couverture a aussi été produit pour un modèle fourni par la société PMC-Sierra. Les résultats de l'application sur cet exemple seront toutefois limités puisque les fonctionnalités et la structure de ce modèle sont confidentielles.

ABSTRACT

Advances in microelectronic technologies allow development of complex digital systems. Consequently the functional verification of such systems becomes increasingly difficult. Moreover, since manufacturing costs of such circuits increase at the same time as the complexity of a circuit, functional errors detected following its manufacture generate costs that can seriously affect its commercial profitability. It is thus necessary to perform a more accurate verification while respecting the schedule imposed by time-to-market constraints. Hence, the development of new verification methodologies is essential to deal with the current constraints governing the development of digital circuits. These new verification methodologies must be aimed at accelerating verification tasks and at enhancing the quality of the overall verification process.

This thesis brings a contribution to the functional verification process by introducing two verification methodologies. The first method proposed in this thesis is a test-bench design methodology that promotes reuse. This method is based on the use of a *Hardware Verification Language* (HVL), the *e* language, that allows using object-oriented and aspect-oriented programming techniques. Hence, an object-oriented and an aspect-oriented partitioning dedicated to test-benches design are proposed in order to increase their reuse potential. This method attempts to accelerate test-bench design and implementation.

The second verification method proposed is a systematic method to develop a coverage analysis module for a digital circuit. The difficulties related to the development of this method are that we need a standardized functional metric applied on a standardized description of the functionalities of the circuit. The functionalities of a circuit are usually described by a functional specification in a textual format. However, it is difficult to work in a systematic way with this kind of description. Thus, we propose to define a standardized description of the specification using a high-level executable language. By formulating a functional metric according to the standardized description of the specification, it is possible to create systematically a coverage analysis module. The

specification language used is the *Specification Description Language* (SDL). This language is executable, which makes it possible to validate the developed specification. The functional metric presented in this thesis is based on the transaction-flow verification technique. A coverage analysis module is implemented using the *e* language and it can be connected in any test-bench dedicated to verify the *Design Under Verification* (DUV). Using our methodology, we are able to provide a quantitative evaluation of test suites aimed at exercising the functionalities defined in the executable specification. The application of these test suites on the *Register Transfer Level* (RTL) design increases error detection possibilities by a better exploration of the design and raises the degree of confidence in the design.

These methods have been applied on different designs. For instance, they were used to facilitate the development of test-benches for a version of a protocol converter. A team of graduate students in the *Groupe de Recherche en Microélectronique* (GRM) works on that protocol converter. Then, we used the design method to create a test-bench for an *Asynchronous Transfer Mode* (ATM) switch. We also used the ATM switch as a case study for the coverage model development methodology. Hence, a SDL specification has been developed for the ATM switch. Then using our methodology, we produced a coverage analysis module to facilitate development of an efficient test suite. The second application on which was applied the coverage model development methodology is a real industrial design developed by PMC-Sierra. For this example, we only provide qualitative results, since this design is confidential.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES SIGLES ET DES ABRÉVIATIONS

AOP	:	**A**spect-**O**riented **P**rogramming
API	:	**A**pplication **P**rogrammation **I**nterface
ATM	:	**A**synchronous **T**ransfer **M**ode
BB	:	**B**ranche de **B**ase
BFM	:	**B**us **F**unctional **M**odel
CEFSM	:	**C**ommunicating **E**xtended **F**inite **S**tate **M**achine
CRC	:	**C**ycle **R**edundancy **C**heck
DUV	:	**D**esign **U**nder **V**erification
FIFO	:	**F**irst-**I**n **F**irst-**O**ut
HDL	:	**H**ardware **D**escription **L**anguage
HEC	:	**H**eader **E**rror **C**ode
HVL	:	**H**ardware **V**erification **L**anguage
IEEE	:	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
MAC	:	**M**odule d'**A**nalyse de la **C**ouverture
MISR	:	**M**ultiple **I**nput **S**hift **R**egister
NNI	:	**N**etwork **N**etwork **I**nterface
OOP	:	**O**bject-**O**riented **P**rogramming
RISC	:	**R**educed **I**nstruction **S**et **C**omputer
RTL	:	**R**egister **T**ransfert **L**evel
SDL	:	**S**pecification **D**escription **L**anguage
UML	:	**U**nified **M**odeling **L**anguage
UNI	:	**U**ser **N**etwork **I**nterface
VLSI	:	**V**ery **L**arge **S**cale **I**ntegrated **C**ircuit
VSIA	:	**V**irtual **S**ocket **I**nterface **A**lliance

LISTE DES ANNEXES

Chapitre 1

INTRODUCTION

La conception d'un circuit numérique se définit comme les tâches permettant de créer un circuit. La vérification fonctionnelle d'un circuit numérique se définit comme les tâches qui tentent de prouver que le circuit conçu est dépourvu d'erreurs fonctionnelles. L'importance de la vérification est incontestable puisque le dysfonctionnement d'un circuit a plusieurs conséquences. En fait, l'avenir commercial d'un circuit peut être compromis dû à une erreur fonctionnelle. D'autre part, les coûts de correction d'une erreur suite à l'entrée en marché d'un produit sont énormes. On peut mentionner, par exemple, l'erreur dans l'unité arithmétique virgule flottante du processeur Pentium de la société Intel qui a atteint de façon permanente la crédibilité de cette société [18] et qui leur a coûté des millions de dollars. De plus, dans le cas où un circuit serait utilisé dans un contexte critique, tel que dans le domaine médical ou aérospatial, le dysfonctionnement d'un circuit peut avoir des conséquences fatales.

La complexité des tâches de conception et la vérification d'un circuit numérique ont longtemps été équilibrées. Cependant, l'avancement des technologies reliées au milieu de la microélectronique permettent actuellement la réalisation de circuits de plus en plus complexes. Citons par exemple la possibilité de synthétiser automatiquement les circuits numériques à partir d'un langage de description matériel (HDL ou *Hardware Description Language*). Cela implique que les circuits entrant en production sont maintenant en mesure de réaliser un plus grand nombre d'opérations et qu'elles sont souvent plus complexes. Les techniques de vérification fonctionnelle des circuits n'ont pas évolué au même rythme que les techniques de conception. En sachant, que la complexité de la vérification fonctionnelle tend à augmenter environ comme le carré de la complexité des circuits numériques [40], il est évident qu'un écart croissant s'est créé entre la conception et la vérification d'un circuit. De ce fait, on avance qu'environ 70% de l'effort de développement d'un circuit est maintenant dédié aux tâches de vérification [6]. La

vérification est donc maintenant sur le chemin critique du processus de développement des circuits numériques. Ainsi, l'industrie et le milieu de la recherche travaillent actuellement au développement de méthodes et d'outils qui permettront de réduire le retard accumulé des méthodes de vérification par rapport aux techniques de conception des circuits numériques.

Le processus de vérification fonctionnelle sert à détecter des erreurs afin d'augmenter le degré de confiance dans le modèle sous vérification (DUV ou *Design Under Verification*). On peut améliorer ce processus en travaillant sur l'accélération de la conception et de la mise en œuvre des suites de test de vérification et sur l'augmentation de la qualité de la vérification. L'accélération de la conception et de la mise en œuvre des suites de test de vérification produit des bancs d'essais en utilisant des techniques plus efficaces. Ainsi, de nouveaux langages ont été développés afin de répondre aux besoins précis de la conception de bancs d'essais. Parmi ces langages de vérification matérielle (HVL ou *Hardware Verification Language*), on retrouve OpenVera™ [33] de Synopsys™, le langage *e* [36] de Verisity™ et TestBuilder™ [11] de Cadence™. Ces langages sont un mélange entre les langages de description de matériel et les langages de haut niveau tel que le C++. De plus, ils sont munis d'éléments spéciaux facilitant la vérification fonctionnelle. Ainsi, en utilisant un langage de haut niveau, il est maintenant possible de développer des méthodologies de conception de bancs d'essais basées sur les techniques développées dans le domaine du génie logiciel. Une approche de réutilisation au niveau des bancs d'essais permet d'accélérer leurs conception et mise en oeuvre.

En plus de la réutilisation, l'efficacité de la vérification peut être rehaussée de plusieurs façons. L'utilisation de technologies de vérification formelle permet d'obtenir une vérification de qualité [5]. Néanmoins, la complexité de la vérification formelle limite son usage sur des modèles de grande complexité. Il est cependant possible de s'inspirer des techniques de vérification formelle pour améliorer la vérification basée sur la simulation. On appelle ce type de vérification la vérification semi-formelle. Au niveau des bancs d'essais, une meilleure détection des erreurs est essentielle. Ainsi, l'analyse des réponses du modèle en cours de simulation permet de détecter les erreurs au premier

moment où elles sont détectables. Pour ce faire, l'utilisation d'assertions, des propriétés comportementales que le modèle doit respecter en tout temps, permet une détection d'erreurs en cours de simulation. La création d'assertions complexes dans un banc d'essais est supportée par les HVL et s'inspire de la technique du *Model Checking* qui est une technique de vérification formelle. De plus, en ayant à notre disposition un langage de haut niveau pour la conception des bancs d'essais, il est possible de créer facilement des modules implantant une vérification automatique des réponses du DUV.

D'autre part, une détection plus rapide des erreurs est aussi un élément essentiel à la qualité de la vérification. Une détection rapide des erreurs est en fait le corollaire d'une meilleure exploration des fonctionnalités du DUV. Pour obtenir une meilleure exploration des fonctionnalités d'un modèle, il faut utiliser des métriques fonctionnelles afin de mesurer la couverture obtenue par l'application d'une suite de tests. La création d'une suite de tests en fonction du taux de couverture actuel de la métrique utilisée permet d'éviter l'application de tests redondants. Ainsi, l'analyse de la couverture fonctionnelle est un élément essentiel pour évaluer l'avancement de la vérification. L'analyse de la couverture sert aussi de base à la création de suite de tests efficaces. La création de suite de tests efficaces peut résulter, par exemple, de l'utilisation d'algorithmes de génération automatique des tests ou de l'adoption d'une approche de génération de test pseudo-aléatoire biaisée en fonction du taux de couverture actuel.

L'objectif du travail présenté dans ce mémoire est d'apporter une contribution aux niveaux des méthodes utilisées pour la vérification fonctionnelle des circuits numériques. Ainsi, deux méthodes pour la vérification sont présentées.

La première méthode sert à concevoir des bancs d'essais en favorisant la réutilisation. La méthode est basée sur l'utilisation d'un langage HVL, le langage *e*, qui permet l'utilisation des méthodologies de conception orientée objets et aspects. Ainsi, un partitionnement orienté objets et aspects, dédié à la création de bancs d'essais, est proposé afin d'augmenter le potentiel de réutilisation des banc d'essais. Cette méthode qui permet d'accélérer le processus de conception des bancs d'essais a fait l'objet de deux communications de conférences [29] [30].

La deuxième méthode de vérification proposée est une technique systématique permettant de développer un module d'analyse de la couverture fonctionnelle d'un circuit numérique. La problématique reliée au développement de cette méthode est qu'il faut avoir des métriques fonctionnelles standardisées appliquées sur une description normalisée des fonctionnalités d'un circuit. Ces fonctionnalités sont habituellement décrites sous forme textuelle dans un document de spécification, il est difficile de travailler de façon systématique avec ce genre de description. Ainsi, on propose d'utiliser une description standardisée de la spécification d'un circuit en utilisant un langage exécutable de haut niveau. En formulant une métrique fonctionnelle en fonction de la description standardisée de la spécification, il est possible de créer un module d'analyse de la couverture de façon systématique. Le langage de spécification utilisé est le *Specification Description Language* (SDL) [21]. Ce langage, qui est décrit avec une notation graphique et textuelle, est utilisé pour la description de systèmes concurrents et interactifs. De plus, il est exécutable, ce qui permet de valider la spécification développée. La métrique fonctionnelle présentée dans ce mémoire est basée sur la technique de vérification des flots transactionnels [4]. L'implémentation d'un module d'analyse de la couverture, en utilisant la méthode proposée est faite en utilisant le langage *e*. Le module d'analyse de la couverture produit avec cette méthode peut être incorporé dans n'importe quel banc d'essais dédié à la vérification d'un circuit au niveau *Register Transfer Level* RTL. L'utilité de ce module est en fait de permettre la création d'une suite de tests qui permettra de couvrir la métrique définie et aussi de réduire l'application de tests redondants. Cela implique qu'avec ce module d'analyse de la couverture, il est possible de créer une suite de tests moins longue et ayant aussi une plus grande puissance de détection d'erreurs

Ces méthodes ont été appliquées sur différents modèles. La méthode de conception a été utilisée pour la création de bancs d'essais dédiés à la vérification d'une version d'un convertisseur de protocoles. Le projet de conversion de protocoles est développé par un groupe d'étudiants de cycle supérieurs du *Groupe de Recherche en Microélectronique* (GRM). Ensuite, la méthode de conception a aussi été utilisée pour la création d'un banc

d'essais dédié à un exemple simple d'un commutateur *Asynchronous Transfer Mode* (ATM). Le commutateur ATM a aussi servi d'exemple pour l'application de la méthode de couverture fonctionnelle. Ainsi, une spécification décrite avec le langage SDL a été développée pour le commutateur ATM. Ensuite, en utilisant la méthode de couverture développée, un module d'analyse de la couverture a été produit. En utilisant, le module produit, il a été possible de développer une suite de tests efficaces qui couvre la métrique utilisée. Finalement, un module d'analyse de la couverture a aussi été produit pour un modèle fourni par la société PMC-Sierra. La présentation des résultats obtenus suite à l'application de notre méthode sur cet exemple sera toutefois limitée, puisque les fonctionnalités et la structure de ce modèle sont confidentielles.

Ce mémoire est divisé en six chapitres. Le chapitre 1 est la présente introduction de ce document. Le chapitre 2 présente une revue de littérature sur le thème de la vérification. Le chapitre 3 introduit la méthode de conception de bancs d'essais axée sur la réutilisation. La méthode est présentée avec des remarques critiques et aussi avec certaines extensions possibles. Le chapitre 4 expose la méthode de couverture fonctionnelle. Ainsi, le flot de développement d'un module d'analyse de la couverture est détaillé. De plus, le développement d'une spécification exécutable avec SDL est passé en revue et une formulation de la métrique utilisée, en fonction du format de la spécification exécutable, est donnée. Nous verrons ensuite de quelle façon est développée un module d'analyse de la couverture. Enfin, une critique du module produit à l'aide de la méthode et les extensions possibles seront exposées. Le chapitre 5 présente les applications des méthodes sur les différents exemples. Finalement, le chapitre 6 conclut ce mémoire.

Chapitre 2

REVUE DE LITTÉRATURE

Ce chapitre présente une revue de littérature sur le sujet de la vérification fonctionnelle des circuits numériques. En premier lieu, une définition de la vérification est donnée et les principes nécessaires à sa réalisation sont recensés. Ensuite, un survol des techniques de vérification est effectué en mettant l'accent sur les techniques applicables sur les circuits numériques au niveau fonctionnel. Finalement, les principes de base de la conception de banc d'essais sont exposés.

2.1 La vérification

La vérification est le processus qui tente de prouver qu'un modèle est correct et donc dépourvu d'erreur. Dans le monde de la conception des circuits VLSI (*Very Large Scale Integrated Circuit*), on s'accorde pour affirmer que la vérification est devenue un aspect important puisqu'elle consomme environ 70% du temps de développement d'un circuit [6]. Ainsi, en considérant que la complexité des circuits croît de façon exponentielle, il ne faut pas oublier que cela affecte aussi la vérification. La différence est que la complexité de la vérification augmente encore plus rapidement que la complexité de la conception des circuits. Wilson [40] avance que la complexité de la vérification augmente au carré de la complexité de la conception d'un circuit. Parallèlement à l'augmentation de la complexité de la vérification, on demande que le temps de développement soit plus court et que la vérification soit plus efficace. Il faut donc recourir à des méthodologies éprouvées qui permettent de procéder à la vérification dans des temps raisonnables et avec une efficacité maximale.

Le même problème a déjà été identifié dans le monde de la conception logicielle. Le besoin de maintenance des logiciels existants a été à la source du problème. L'augmentation de la complexité des logiciels combinée avec le manque de méthodes utilisées par les concepteurs ont vraiment fait émerger la science de la vérification logicielle. Au niveau où est rendue la vérification fonctionnelle, la conception d'un banc

d'essais dédié à un circuit numérique doit être considérée comme un projet logiciel. Ainsi, on doit s'inspirer des techniques du génie logiciel établies. La littérature au niveau des fondements de la vérification fonctionnelle des circuits numériques n'est pas exhaustive. On retrouve donc dans la littérature portant sur la vérification fonctionnelle au niveau logiciel, des descriptions beaucoup plus riches sur les techniques de vérification de base.

2.1.1 Vérification versus Validation

Il existe une confusion pour ce qui est de la signification des termes *vérification* et *validation*. Beizer [3] propose une définition de ces termes. On peut définir que les requis sont ce que le modèle devrait faire et que la spécification d'un modèle est le document qui tente de représenter fidèlement les requis. Si on accepte les définitions des requis et de la spécification, la vérification est définie comme les activités qui tentent de comparer le modèle avec la spécification et la validation est définie comme les activités qui tentent de comparer les requis avec le modèle. Cependant, les requis sont une perception de ce que le modèle devrait faire et cette perception peut être différente pour les divers intervenants dans la conception du modèle. Étant donné que la spécification d'un modèle est la formulation plus ou moins formelle des requis, il est possible que des erreurs soient introduites dans la spécification. Ce type d'erreur n'est pas du ressort de la vérification, mais plutôt de la validation. Le Tableau 2-1 résume les définitions précédemment énoncées.

Tableau 2-1: Définition des termes pour la vérification versus la validation

Terme	Définition
Requis	Ce que le modèle devrait faire
Spécification	Le document formulant les requis
Modèle	Le modèle dérivé de la spécification (Circuit, logiciel, SOC)
Système de vérification	Un autre modèle ou un banc d'essais dérivé de la spécification implémentant une méthode de vérification

Dans le domaine de la vérification, les requis n'existent pas de façon tangible. La base de la vérification étant la spécification, un système de vérification n'est pas en mesure de détecter une erreur provenant des spécifications. On peut donc avancer que la

validation est en quelques sortes la vérification des spécifications d'un modèle. Bergeron [6] complète ces définitions en résumant la vérification fonctionnelle par une hypothèse statistique. L'hypothèse sous-test est : *Est-ce que le modèle est fonctionnellement correct ?* La réponse est produite par le système de vérification. La Figure 2-1 présente l'analyse des types d'erreur qui découlent de la réponse obtenue de l'hypothèse sous test.

	Réponse à l'hypothèse	
	Erreur	**Pas d'erreur**
Modèle incorrect		Type II (Faux Positif)
Modèle correct	Type I (Faux Négatif)	

Figure 2-1: Type d'erreurs

Les cas corrects se présentent lorsque le système de vérification détecte une erreur et que le modèle est effectivement incorrect, ainsi que lorsque le système de vérification ne détecte pas d'erreur et que le modèle est correct. Cependant, lorsque le système de vérification détecte une erreur et que le modèle est correct, nous sommes en présence d'une erreur de type I, ce qui signifie qu'il y a une erreur dans le système de vérification. En fait, la vérification sert à la fois à vérifier le modèle et le système de vérification. Il faut donc être critique lors de la détection d'une erreur en se demandant si elle provient du modèle ou du système de vérification. Cependant, il faut éviter les erreurs de type II qui signifient que le système de vérification ne détecte pas d'erreur lorsque le modèle est incorrect. Une erreur de type II est induite par une mauvaise interprétation des spécifications par les créateurs du modèle et du système de vérification. Il est aussi possible qu'une erreur de type II soit une erreur dans la spécification, ce qui signifie que c'est du ressort de la validation. Toutefois, si on veut réduire la fréquence des erreurs de type II au niveau de la vérification, il est important de respecter le schéma présenté à la Figure 2-2 tiré de [6].

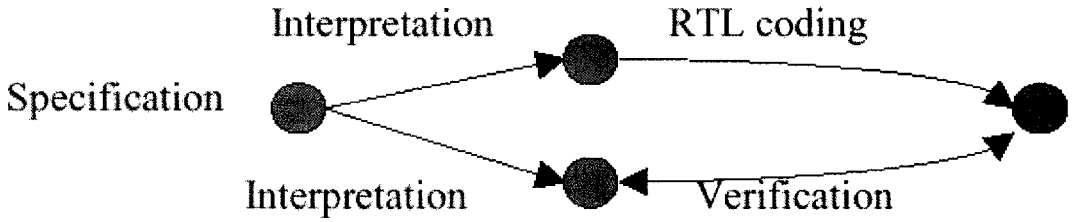


Figure 2-2: Indépendance de la vérification

En ayant deux interprétations différentes de la spécification, on réduit la probabilité d'obtenir des erreurs de type II. Cette approche est en fait un principe de base de la vérification.

2.1.2 Principes du test

Les tests sont l'unité de base du processus de vérification. Un test devrait pouvoir fournir une information ou plusieurs informations sur la validité du modèle. Myers [26] a défini certains principes qui devraient être suivis si on veut obtenir une stratégie de tests efficace.

- *Les tests doivent être basés sur les spécifications du système*

  Les spécifications sont ce que les clients attendent du produit. Ainsi, il faut se concentrer sur cela pour leurs fournir ce qu'ils veulent.

- *La planification des tests doit être faite avant le début des tests*

  Cela permet d'accélérer le processus de conception (parallélisme avec la conception du système) et de donner des mises en garde à l'équipe de conception du système. Plus une erreur est détectée tôt dans le processus de conception, moins les coûts de correction sont élevés.

- *Le principe de Pareto s'applique aux tests (80% des erreurs dans 20% des modules) ou la probabilité de trouver une erreur dans un module est proportionnelle au nombre d'erreur trouver dans ce même module*

  Il faut tenter de découvrir les modules suspects. La leçon qu'il faut en tirer est que lorsqu'on trouve plusieurs erreurs dans un module, il faut l'étiqueter comme suspect et le vérifier plus intensément.

- *Les tests doivent être écrits autant pour les cas invalides et inespérés que pour les cas valides et espérés*

  Un testeur écrit naturellement des cas de test pour vérifier les spécifications, mais l'inverse des spécifications fait autant partie des spécifications. Un test doit tenter de trouver des succès lorsqu'il introduit des entrées incorrectes aussi bien qu'il doit tenter de trouver des erreurs lorsqu'il introduit des entrées correctes.

- *Examiner ce que le design devrait faire est seulement la moitié du travail. L'autre moitié est d'examiner ce que le design ne devrait pas faire.*

  C'est en quelque sorte la suite du principe précédent. Un système bancaire qui produit les paies pour tous ses employés est correct. Cependant, si le système produit aussi des paies pour des employés inexistants, il y a une erreur.

- *Pour être le plus efficace possible, les tests devraient être conçus par une équipe indépendante de l'équipe de conception*

  Premièrement, les tests sont un processus destructif, où cependant le but ultime est constructif. Ainsi la personne qui a conçu quelque chose n'est pas la mieux placée pour tenter de la détruire. Ensuite, une erreur peut survenir du fait que le concepteur n'a pas bien compris ce qu'il avait à faire. Ainsi les tests qu'il élaborera contiendront la même erreur que le design.

D'autre part, un bon test doit posséder les caractéristiques suivantes :

- *Il doit avoir une haute probabilité de trouver une erreur;*
- *Il n'est pas redondant;*
- *Il doit être reproductible;*
- *Il ne doit pas être ni trop simple ni trop complexe.*

2.1.3 Testabilité du modèle

Tout modèle doit passer par une étape de vérification. Il est donc préférable d'en tenir compte lors de sa conception. Pressman [27] énumère les caractéristiques requises afin d'améliorer la testabilité et de faciliter la vérification des modèles.

- *Opérable :* *Mieux le système fonctionne, plus il est facile à tester*

Il ne doit pas y avoir d'erreurs qui bloquent l'exécution du modèle. De plus, le développement doit évoluer par fonctionnalité, afin de permettre de tester et de développer simultanément.

- *Observabilité :* *Ce que tu vois est ce que tu testes*

  Les sources du code sont préférablement accessibles. L'interne du système est visible afin d'effectuer des tests de boîte blanche. Les erreurs internes sont automatiquement détectées et reportées.

- *Contrôlabilité :* *Plus on contrôle le modèle, plus on peut automatiser et optimiser la vérification*

  Toutes les sorties peuvent être générées à partir de certaines combinaisons des entrées. Toutes les parties du modèle doivent être accessibles. Les états et les variables du modèle doivent pouvoir être contrôlés par l'ingénieur de test. Par exemple, un modèle qui possède un compteur de 64 bits devrait fournir une fonction de pré-chargement. Les formats des entrées et des sorties doivent être structurés et consistants.

- *Modularité :* *En contrôlant l'étendue des tests, on peut plus facilement isoler les problèmes et faire des tests plus intelligents*

  Les modules qui composent le modèle doivent pouvoir être testés indépendamment.

- *Simplicité :* *Moins on en vérifie, plus la vérification se fait rapidement*

  Des fonctionnalités simples définies par une structure simple et aussi un codage simple permettent des tests plus efficaces.

- *Stabilité :* *Moins on a de changements, moins la vérification sera perturbée*

  Les changements sur le modèle ne doivent pas être fréquents, ils doivent être contrôlés et ne doivent pas invalider les tests existants. Le modèle doit bien récupérer suite à une erreur.

- *Compréhension : Plus on a d'information, plus la vérification sera intelligente*

Il faut bien comprendre le modèle. Les dépendances entre les composants internes et externes doivent être bien comprises. Les changements sur le modèle doivent être communiqués à l'ensemble des intervenants concernés. La documentation doit être instantanément accessible, bien organisée, spécifique, détaillée et précise.

2.1.4 Type de vérification

La vérification d'un modèle peut être effectuée en utilisant trois approches. Dans cette section, il sera question de ces trois approches qui sont les tests structurels, les tests fonctionnels et les méthodes de preuves formelles.

2.1.4.1 Test structurel et test fonctionnel

Les tests structurels et fonctionnels utilisent généralement la simulation afin d'exécuter les tests. Les tests structurels et les tests fonctionnels sont complémentaires et visent à obtenir un meilleur degré de confiance dans un modèle, mais d'une manière différente. Pour les tests structurels, on travaille avec des métriques qui sont directement dérivées de la structure du modèle. On peut considérer la structure d'un circuit au niveau porte logique ou encore au niveau de la description RTL. Ainsi, lorsqu'on vise à couvrir les branches, les énoncés, les états ou les transitions dans une machine à états, on parle de tests structurels. Les tests structurels sont finis mais, selon la métrique utilisée, une couverture totale devient plus ou moins accessible. Par exemple, la couverture complète des énoncés d'un modèle est facile à atteindre, car si on ne peut pas l'atteindre, c'est qu'il y a des énoncés inutiles ou insérés pour des fins d'optimisations. Les métriques structurelles peuvent se compliquer et le nombre de cas à couvrir augmente alors exponentiellement. Un exemple serait la couverture complète des chemins d'un modèle qui devient difficile lorsqu'il y a plusieurs éléments de décision et des boucles qui entrent en jeu. Ces métriques pourraient être tout de même qualifiées de métriques simples, car il existe une façon d'obtenir une couverture complète.

La couverture structurelle se complique vraiment lorsque le modèle est concurrent, car les métriques structurelles qu'on peut alors définir sont par exemple, la couverture de

couples ou de triplets d'états lorsqu'il y a plusieurs machines à états ou encore, la couverture de points de synchronisation entre plusieurs chemins. Ainsi, dans le cadre d'un modèle intensif, la couverture de ces métriques devient extrêmement complexe et impossible à obtenir dans un temps raisonnable. C'est ce qu'on pourrait appeler des métriques structurelles complexes.

Par contre, le fait qu'il soit ardu de couvrir convenablement ces métriques complexes n'empêche pas qu'il soit essentiel de couvrir certains cas pour obtenir un degré de confiance convenable dans le modèle. Les tests fonctionnels servent à définir des tests qui effectuent une couverture partielle de métriques structurelles dites complexes. Ainsi, en utilisant des techniques de tests fonctionnels, on tente de cibler les cas dans un large ensemble de cas qui seraient susceptibles de provoquer une erreur dont la conséquence serait significative. La manière utilisée pour caractériser les cas fonctionnels à vérifier est de créer un modèle fonctionnel. Un tel modèle est en fait une abstraction du modèle sous vérification. Les métriques de couverture fonctionnelle sont un mélange de comportement temporel et de données. Pour donner des exemples de test fonctionnel, il y a l'analyse transactionnelle et l'analyse de machines à états comportementales. L'analyse transactionnelle définit les chemins comportementaux que peut emprunter une transaction dans un modèle. Une transaction est une séquence d'actions appliquées sur le modèle. La technique de vérification de machines à états comportementales permet la couverture des états, des transitions, des séquences de transition ou encore du temps d'une transition. Les techniques de test fonctionnel seront exposées plus en détail à la section 2.2.

2.1.4.2 Méthode de preuve formelle

Les méthodes de preuve formelle tentent de vérifier un modèle en se basant sur une représentation mathématique. Si on est en accord avec l'affirmation que les tests peuvent seulement détecter la présence d'erreurs et qu'ils ne peuvent pas prouver leur absence [14], cela implique que les tests devraient être abandonnés et seulement des méthodes de preuves formelles devraient être utilisées. Cependant, les méthodes de preuve ont des limitations et les tests ont des avantages importants par rapport à ces méthodes [20]. Premièrement, il n'y a pas de méthode de preuve absolue qui prouve qu'un modèle est

correct, il faut plutôt parler de preuve d'équivalence. Lorsqu'on sait que la preuve d'équivalence se fera sur la base d'une spécification, il faut se demander ce que vaut la spécification. Howden [20] explique que tous les modèles possèdent des fonctions explicites qu'on retrouve dans la spécification, mais aussi des fonctions implicites qu'on ne retrouve généralement pas dans la spécification. Ainsi, une spécification complète doit définir les fonctions implicites et explicites du modèle. Ensuite, lorsqu'on a une spécification complète, une preuve formelle tente de prouver que l'interprétation du concepteur est correcte et non de prouver l'intention de la spécification. De plus, les méthodes de preuve sont seulement applicables à des modèles de complexité modérée.

Rashinkar & al. [28] résument les méthodes de preuve formelle utilisées et développées dans un contexte de recherche. Premièrement, il y a la méthode formelle de preuve de modèle. Cette technique utilise des propriétés comportementales directement dérivées de la spécification du modèle et exprimées avec une logique temporelle. Un outil de preuve formelle de modèle tentera de prouver statiquement que les propriétés définies sont respectées dans le modèle. Ensuite, il y a la technique de preuve de théorème qui est encore au stade de la recherche. Cette technique est utilisée afin de prouver formellement des théorèmes qui représentent le comportement du modèle. Finalement, la technique de preuve d'équivalence est utilisée pour prouver de façon statique que deux modèles sont logiquement équivalents.

2.1.5 Critère d'arrêt

Théoriquement le critère d'arrêt de la vérification survient lorsque le modèle sous vérification est complètement exempt d'erreur. Il existe trois façons de parvenir au critère d'arrêt, quel qu'il soit: les tests structurels, les tests fonctionnels et les preuves formelles. La combinaison de ces trois méthodes de vérification mène à une vérification complète si on tient compte des limites théoriques de la vérification que Manna et Waldinger [24] ont énoncées :

- *On ne peut jamais être sûr que les spécifications sont correctes;*
- *On ne peut jamais être certain qu'un système de vérification est correct;*

- *Aucun système de vérification ne peut vérifier un modèle correct. En fait cela découle des deux énoncés précédents.*

Ainsi, le but du critère d'arrêt n'est pas de déterminer le point où on est certain qu'il n'y a plus d'erreur dans le modèle, mais le point où la probabilité qu'une erreur survienne est acceptable. Cette probabilité acceptable est définie selon différents facteurs tel que l'environnement où sera utilisé le circuit. Si nous prenons un circuit devant être utilisé dans un stimulateur cardiaque, le critère d'arrêt doit faire en sorte que la probabilité qu'une erreur survienne soit très faible, puisque dans le cas d'un dysfonctionnement, une erreur peut avoir des conséquences fatales.

Cependant, dans le but de fixer un critère d'arrêt concret dans le processus de vérification, la technique utilisée est la création d'un plan de vérification. L'élaboration d'un plan de vérification est une méthode systématiquement conseillée pour spécifier l'effort de vérification d'un projet [6][12][19][22][22][27][28][41]. Le plan de vérification est considéré comme la spécification de la vérification. On y définit, entre autre, les techniques de vérification utilisées, les bancs d'essais nécessaires à la vérification, les cas de test, les métriques utilisées, les méthodes de preuves utilisées pour certaines parties du modèle, les ressources allouées à la vérification et les échéanciers qui devront être respectés. C'est en se basant sur les objectifs définis dans le plan de vérification qu'on détermine concrètement l'atteinte du critère d'arrêt de la vérification. En fait, la couverture du plan de vérification est la métrique globale utilisée pour mesurer la complétude de l'effort de vérification.

2.2 Techniques et mesures de la vérification

La vérification basée sur une technique de simulation peut se résumer par l'application de tests sur le modèle sous vérification et par la vérification des réponses produites par le modèle. Les tests sont aussi appelés les cas de tests. Ces derniers sont définis dans le plan de vérification et doivent être définis de façon structurée. La définition des cas de test est effectuée en utilisant des techniques de vérification. Les techniques de vérification définissent en quelque sorte un modèle qui servira à la création des tests. La vérification est mesurée en utilisant des métriques. Il émane de chaque

technique de vérification une métrique qui sert à mesurer le taux de couverture résultant de l'application des tests. De plus, pour chaque technique de vérification il faut être en mesure de vérifier les réponses produites par l'application des tests sur le modèle.

Dans cette section, certains aspects reliés aux techniques de vérification seront exposés. En premier lieu, il sera question de la visibilité et des différents niveaux d'abstraction relatifs aux techniques de vérification. Ensuite, certaines techniques de vérification seront exposées. Finalement, il sera question de la production des tests et des techniques de vérification des réponses.

2.2.1 Visibilité

L'utilisation d'une technique de vérification requiert une certaine connaissance du modèle auquel on applique l'effort de vérification. En fonction de la visibilité qu'on a sur le modèle, trois approches sont définies. Il s'agit des approches dites :

- *Boîte blanche;*
- *Boîte noire;*
- *Boîte grise.*

Une technique de vérification dite de type boîte blanche implique une visibilité complète du modèle, c'est-à-dire qu'on connaît ses interfaces externes et sa structure interne. Une approche de type boîte blanche permet d'avoir une contrôlabilité et une observabilité totale sur le modèle. Les tests structurels utilisent une approche boîte blanche, car ce type de test dépend directement de la structure du modèle.

L'approche boîte noire implique une connaissance des interfaces externes du modèle et des fonctionnalités du modèle, mais sans connaître sa structure interne. Ainsi, une approche de type boîte noire permet de vérifier le modèle sous l'angle de ses fonctionnalités sans tenir compte de l'implantation. Les tests fonctionnels sont habituellement effectués en adoptant une approche boîte noire. Cependant, une approche boîte noire pose trois problèmes : le contrôle, l'observation, et les limites. Ayant seulement un contrôle sur les interfaces externes du modèle, il peut être difficile de déterminer certaines séquences complexes qui permettront de conduire à une erreur. D'autre part, l'observation des réponses du modèle se faisant toujours par les interfaces

externes du modèle, il est possible qu'une erreur interne soit stimulée, mais sans qu'on puisse l'observer dû à un masquage interne de l'erreur. Ce dernier problème, relié à l'approche boîte noire, est dû aux limites imposées par la structure interne du modèle, car plusieurs erreurs sont provoquées aux limites fonctionnelles et structurelles du modèle.

Le compromis qui permet de tirer avantage à la fois de l'approche boîte noire et boîte blanche est d'utiliser une approche boîte grise. Cette dernière permet de définir les tests de la même façon qu'une approche boîte noire, mais en ayant une certaine connaissance de la structure interne du modèle. Ainsi, la contrôlabilité et l'observabilité sont augmentées et on peut déterminer des tests qui permettront d'explorer les limites fonctionnelles et structurelles du modèle.

2.2.2 Niveau d'abstraction

Le concept de niveau d'abstraction est très important lors de l'étape de vérification d'un circuit. La complexité actuelle des circuits numériques requiert qu'on se serve d'une certaine forme d'abstraction afin de statuer sur la complétion de la vérification. Par exemple, au niveau porte logique différents modèles de fautes sont utilisés afin de modéliser des défectuosités au niveau physique. Cependant, il existe plusieurs niveaux d'abstraction pour les modèles de pannes. Il y a le modèle de collage simple qui sert à vérifier si une broche d'une porte logique est collée à 0 ou à 1 suite à la fabrication du circuit. Ce modèle de panne devient inadéquat lorsque la complexité du circuit augmente. Il faut donc travailler avec un modèle de panne à plus haut niveau d'abstraction. Par exemple, il existe un modèle de panne spécialisé pour les processeurs qui considère les registres et les fonctions entre les registres au lieu de la valeur binaire des signaux. Ce dernier exemple est considéré comme étant du test fonctionnel, car on utilise un modèle fonctionnel pour abstraire des portes logiques.

De façon similaire à la vérification d'un circuit, la vérification d'un modèle HDL se fera à différent niveau d'abstraction. Il existe plusieurs techniques de vérification qui peuvent être considérées selon le niveau d'abstraction auquel on considère le modèle. En général, il s'agit du test fonctionnel ou du test structurel. Par exemple, la vérification d'une machine à états est une technique de vérification. Cette technique de vérification

sera du test structurel si on vérifie une machine à états implantée en RTL. Cependant, si on considère une machine à états comportementale représentant les fonctionnalités du modèle, on fait alors de la vérification fonctionnelle. Il est parfois ambigu de différencier si on est au niveau du test fonctionnel ou du test structurel. En fait, le niveau d'abstraction utilisé pour le développement des tests est le déterminant qui juge si on est au niveau fonctionnel ou structurel.

2.2.3 Techniques et métriques

Dans cette section, un survol des techniques de vérification est effectué. Cette section considère les techniques applicables sur un modèle HDL, on ne détaille pas les techniques de vérification applicables au niveau circuit. Une technique de vérification définit un modèle qui sera utilisé pour produire les tests. De plus, une technique fournit une métrique qui sert à mesurer l'avancement de la vérification en fonction de la technique utilisée. Les techniques décrites dans cette section sont la revue de code, les techniques basées sur le code, le test par mutation, l'analyse partitionnelle et aux limites et les techniques basées sur les graphes.

2.2.3.1 Revue de code

La technique de revue de code est une technique de type boîte blanche qui ne requiert pas l'exécution du modèle. Cette technique de vérification a été proposée par Weinberg [38]. Le principe de la revue de code est d'exécuter mentalement le code source d'un modèle dans le but de trouver des erreurs. Premièrement, cette technique peut être appliquée par la majorité des programmeurs. De plus, un programmeur peut appliquer cette méthode formellement avec une liste de critères de revue de code, mais la méthode la plus efficace est celle des revues de code par groupe. Habituellement, une procédure est établie pour les revues de code de groupe. On forme un groupe de trois ou quatre personnes. Le premier est ordinairement un ingénieur d'assurance qualité qui agira comme modérateur. Ensuite, il y a le programmeur, qui passera sous révision, et les deux autres peuvent être des ingénieurs concepteurs ou testeurs. La procédure est la suivante :

1. *Le programmeur proclame à voix haute chaque énoncé du code.*

2. *À tout moment, les membres du groupe peuvent interrompre le narrateur lorsqu'on suspecte une erreur.*
3. *Le point est alors débattu pour confirmer ou infirmer s'il s'agit d'une erreur.*
4. *À la fin de la séance, la liste des erreurs trouvées est remise au programmeur pour qu'il corrige les erreurs après la séance de revue de code.*

L'expérience a démontré que la personne la plus efficace dans ces rencontres est le programmeur qui trouve ses propres erreurs. Un suivi des erreurs détectées doit être fait au début de la prochaine rencontre dans le but de confirmer la correction des erreurs. Il est aussi important dans ces revues de code de ne pas commencer à analyser le programmeur à la place du code, sinon on s'éloigne du but premier qui est de trouver des erreurs dans le code. Cette technique donne de très bon résultat. Plusieurs publications et études contrôlées [26] le confirment en avançant qu'on peut détecter environ 30% à 70% des erreurs. Il faut comprendre que ce pourcentage est pris sur le total des erreurs détectées et non sur le total des erreurs, car il est impossible de déterminer le nombre total absolu d'erreur dans un modèle. Un autre avantage de cette méthode est qu'elle permet la détection des erreurs tôt dans le projet et plus une erreur est détectée tôt dans le processus de conception, moins ça coûte cher [27]. Aussi, il est prouvé que le programmeur corrige mieux les erreurs lorsqu'elles sont trouvées au début du processus de conception comparativement aux corrections apportées vers la fin d'un projet où la pression est maximale. La métrique utilisée avec cette technique de vérification est la quantité de code passée sous revue.

Cette technique peut aussi être appliquée à tous les niveaux du développement d'un circuit. Par exemple, il est possible d'effectuer des revues pour les spécifications, le plan de vérification, l'architecture du circuit et pour la conception des bancs d'essais.

2.2.3.2 Techniques basées sur le code

Les techniques basées sur le code font partie des tests structurels et utilisent une approche boîte blanche. Chaque technique définit une métrique structurelle qui sert à vérifier le code. Le Tableau 2-2 résume les métriques structurelles énumérées dans le

document de taxonomie de la vérification fonctionnelle produit par *Virtual Socket Interface Alliance* (VSIA) [37].

Tableau 2-2: Métriques Structurelles

Métrique	Description
Énoncé	Mesure les énoncés exécutés dans le code.
Bit	Mesure que chaque bit de tous les signaux du modèle ont été basculés.
Transition	Mesure les transitions exercées dans une machine à états
État visité	Mesure les états visités dans une machine à états
Liste de sensibilité	Mesure que chaque processus a été déclenché par chaque signal définit dans sa liste de sensibilité.
Couverture de branche	Mesure que les branches découlant des éléments décisionnels tel que les *case* et les *if-else* ont été exercées.
Couverture d'expression	Mesure que toutes les combinaisons possibles des expressions logiques dans les conditions des éléments décisionnelles tel que les *if* ont été exercées.
Couverture de chemin	Mesure que les chemins définis par les séquences de branches empruntées ont été exercés.
Couverture de signal	Mesure que les signaux d'états ou les adresses d'une mémoire ont été exercées.

Les techniques de couverture structurelle sont bien développées dû à l'indépendance qu'elles possèdent face au modèle. Cette indépendance permet la création d'outils automatiques qui permettent de mesurer la couverture de ces métriques structurelles. Il existe plusieurs outils commerciaux qui sont dédiés à l'analyse de la couverture de certaines de ces métriques pour des descriptions VHDL ou Verilog.

2.2.3.3 Test par mutation

Les tests par mutation sont une technique de tests qui diffère des autres techniques de test. Le principe de base est qu'on ne choisit pas les tests en fonction du modèle sous vérification, mais plutôt en appliquant les tests sur des mutants du modèle. Un mutant est le modèle original dans lequel une modification a été introduite intentionnellement. Quelques exemples d'opérations mutationnismes peuvent être réalisés par le remplacement d'opérations arithmétiques, logiques, relationnelles ou encore le remplacement d'une variable. Ensuite, on tente de déterminer les tests qui seront assez agressifs pour tuer le plus de mutants possible. Le Tableau 2-3 expose un exemple simple de test par mutation.

Tableau 2-3: Exemple de test par mutation

Modèle		Test ($x=3$)	Test ($x=2$)
Original	$y = x + 2$	5	4
Mutant 1	$y=x\times2$	6	4
Mutant 2	$y=x$	3	2

Dans l'exemple présenté, le test $x=2$ n'est pas un bon test, puisque le mutant 1 résiste à ce test. Cependant, le test $x=3$ permet de tuer tous les mutants. Une méthode d'évaluation des erreurs fonctionnelles par l'analyse des mutants sur les modèles VHDL a été proposée par Zhang et Harris [42]. Leur approche a pour but de concurrencer les techniques de couverture de codes. Ils ont développé un outil qui permet de faire l'examen du code source du modèle HDL dans le but de générer automatiquement les mutants. Les résultats de couverture qu'ils ont obtenus sont comparables aux résultats obtenus avec les techniques de couvertures de code. Il est aussi possible d'appliquer les tests par mutation à d'autres niveaux d'abstraction de la vérification.

Les tests par mutations comportent des désavantages qui sont plutôt incontournables. Premièrement, la production des mutants implique qu'on produise beaucoup de changements dans les codes sources, ce qui n'est pas vraiment désirable dans un projet de grande envergure [6]. De plus, le fait d'avoir plusieurs mutants nécessite des compilations multiples et des exécutions à répétition de l'original et de ses mutants pour chaque cas de tests [41]. Ainsi, cette méthode est particulièrement inappropriée pour un modèle HDL où l'exécution est déjà ralentie par l'ordonnancement des divers processus qui permettent de simuler la concurrence d'un modèle HDL.

2.2.3.4 Analyse partitionnelle et test limites

L'analyse partitionnelle et les tests limites sont, à l'extrême, une vérification exhaustive. Plus précisément, on vérifie toutes les possibilités de tests pouvant être appliquées aux interfaces externes du modèle. L'analyse partitionnelle permet d'appliquer le concept *diviser-pour-régner* au domaine des tests du modèle. En fractionnant le domaine, plusieurs partitions peuvent être construites pour réussir à

couvrir tout l'espace de valeur possible. Ensuite, la métrique découlant de cette technique peut être la couverture d'une partition particulière ou la couverture d'un ou plusieurs cas dans chaque partition définie. L'analyse partitionnelle permet une couverture complète du domaine lorsque les partitions sont mutuellement exclusives et que l'union des partitions est égale au domaine total. L'analyse partitionnelle est en fait une sorte d'abstraction qui permet de partitionner le domaine de façon fonctionnelle. En ayant une étendue moins importante à couvrir, les tests fonctionnels deviennent plus efficaces. Il est à noter que l'analyse partitionnelle est aussi appelée le test du domaine.

Les tests limites font partie de l'analyse partitionnelle en étant en fait une manière de diviser le domaine, c'est-à-dire, que les limites du modèle représentent une ou des partitions spécifiques auxquelles une attention particulière est portée. Les tests limites sont très efficaces, car il est fréquent que les concepteurs oublient de vérifier les valeurs aux limites du modèle. Au niveau des limites fonctionnelles, la création des tests limites doit être basée sur les spécifications fonctionnelles. Par exemple, si un paramètre appartient à un intervalle de 1 à 100, les valeurs de tests limites seront 0, 1, 2, 99, 100 et 101. Ainsi, ces valeurs sont des partitions limites du domaine du paramètre d'intérêt.

2.2.3.5 Techniques de vérification basées sur un graphe

Une technique de vérification basée sur un graphe est une technique de test fonctionnel utilisant une approche boîte noire. Cela consiste à définir un graphe qui représente les fonctionnalités du modèle. La production de tests est basée sur la structure du graphe et la métrique fournie par cette technique dépend du type de graphe utilisé. Il existe plusieurs techniques ayant recours à la théorie des graphes [4][41].

Tableau 2-4: Type de techniques basées sur le graphe

Technique	Description
Flot de contrôle	Cette technique met l'accent sur les éléments de contrôle qu'on retrouve dans le modèle.
Flot de donnée	Cette technique met l'accent sur le cheminement des données dans un modèle.
Machine à états	Cette technique considère les états du modèle et les transitions entre les différents états. Il est possible d'imbriquer les machines à états.
Flot transactionnel	Cette technique est un hybride de techniques basées sur le graphe. On considère les transactions pouvant être appliquées au modèle. Les transactions sont des structures de données transigeant dans le graphe. Il y a aussi la notion d'état qui est considérée

	dans le graphe.

En ayant une structure sous la forme d'un graphe, on peut dériver plusieurs métriques qui consistent à couvrir des éléments du graphe défini. Par exemple, il y a la couverture des nœuds, des liens et des séquences de nœuds.

2.2.4 Production de tests et couverture

La production de tests est directement liée à la technique de vérification utilisée. De plus, l'application des tests produits permet d'obtenir un taux de couverture de la métrique définie par la technique de vérification. Il existe trois approches pour la production de test, il s'agit des approches : déterministe (manuelle), pseudo-aléatoire (manuelle) et automatique.

2.2.4.1 Test déterministe

La création du test déterministe consiste à définir manuellement des tests basés sur la technique de vérification utilisée. L'avantage des tests déterministes est de pouvoir cibler les éléments importants sélectionnés par les ingénieurs de test. En connaissant les tests appliqués sur le modèle, on connaît la couverture obtenue par la suite de tests. Ainsi, le degré de confiance obtenu par des tests déterministes dépend de la quantité et de la qualité des tests produits. Le désavantage de cette technique de production de tests est que la suite de tests déterministes est limitée aux tests auxquels les ingénieurs de tests auront pensés. De plus, la production de la suite de tests nécessite une certaine expertise de la part des ingénieurs de tests. Le développement de cette suite de tests peut demander un temps de développement considérable en fonction de la complexité du DUV.

2.2.4.2 Test pseudo-aléatoire

La génération des tests de manière pseudo-aléatoire permet une création rapide des tests. Dans le cas d'une approche de génération pseudo-aléatoire, il est nécessaire d'évaluer la couverture de la métrique utilisée afin d'obtenir le taux de couverture induit par les tests générés. La génération peut être complètement pseudo-aléatoire ou bien biaisée. Certains modèles sont résistants aux tests pseudo-aléatoires, il faut donc biaiser la

génération afin de cibler les endroits inexplorés du modèle. Le test pseudo-aléatoire a l'avantage de produire des tests auxquels un ingénieur n'aurait pas pensé en adoptant une approche de test déterministe. L'opposé est cependant aussi vrai, c'est-à-dire qu'une approche pseudo-aléatoire peut négliger des cas de test précis qui n'auraient pas été négligés par une approche déterministe. Il est donc important de combiner les deux approches de production de tests afin d'obtenir la meilleure couverture possible du modèle. D'autre part, le test pseudo-aléatoire peut être redondant, c'est-à-dire qu'il génère des vecteurs qui n'augmentent pas le taux couverture de la métrique utilisée.

2.2.4.3 Tests automatiques

On peut automatiser la production de test en se basant sur les deux approches précédemment exposées. Pour automatiser le test déterministe, il faut développer une méthode systématique pour définir un test. Cette méthode doit être basée sur la technique de vérification utilisée et viser à couvrir la métrique définie. Pour ce qui est de l'automatisation du test pseudo-aléatoire, cela consiste à produire automatiquement des contraintes pour le générateur pseudo-aléatoire afin de biaiser la génération. La technique développée doit aussi tenir compte de la technique de vérification utilisée et du taux de couverture actuel. Le test automatique est une technique qui a été explorée et développée au niveau circuit. Cependant, au niveau de la vérification d'un modèle HDL, il s'agit d'un domaine en émergence qui n'a touché que quelques techniques de vérification.

2.2.5 Vérification des réponses

Les réponses produites par l'application de tests doivent être vérifiées afin de statuer si le modèle est correct ou s'il y a erreur. Il existe plusieurs façons de vérifier les réponses d'un modèle. Dans cette section, nous allons énumérer quelques méthodes connues. Les méthodes de vérification des réponses peuvent être séparées en deux catégories : les méthodes en post-simulation et les méthodes en cours de simulation. Il est à noter que dans le cadre de la vérification d'un modèle, plusieurs techniques de vérification des réponses peuvent être utilisées simultanément.

2.2.5.1 Méthodes en post-simulation

Les méthodes en post-simulation consistent à recueillir les réponses du modèle dans un format quelconque et de vérifier les résultats suite à l'application d'une suite de tests.

Premièrement, si on accumule les résultats dans un format pouvant être visualisé sous forme de chronogrammes, on peut procéder à une inspection visuelle des chronogrammes. Cette méthode est rapide et efficace pour de petits modèles, mais complètement irréalisable lorsqu'on doit examiner des milliers de cycles de simulation. De plus, la probabilité de faire une erreur dans l'inspection des traces est très grande. Une approche plus structurée serait de comparer automatiquement les résultats obtenus avec une référence connue comme étant correcte. La manière utilisée pour produire cette référence pourrait être, par exemple, l'exécution de la suite de tests sur une version comportementale équivalente du modèle sous vérification.

Une autre option est d'accumuler les réponses dans un format texte. Ensuite, le procédé est le même qu'avec un format sous forme de chronogramme : une inspection manuelle ou une comparaison automatique avec une référence.

L'avantage des méthodes en post-simulation est de permettre des simulations rapides. Cependant, la détection des erreurs étant faite suite à la simulation, il est possible qu'on gaspille du temps de simulation, en particulier dans le cas d'une longue simulation pour laquelle un problème apparaît tôt dans la simulation.

2.2.5.2 Méthodes en cours de simulation

Les méthodes de vérification des réponses en cours de simulation permettent la détection des erreurs plus rapidement et ainsi, possiblement, elles réduisent le temps de simulation dans le cas où une erreur serait détectée. De plus, n'ayant plus besoin de conserver les réponses pour un traitement post-simulation, une économie de mémoire appréciable peut être obtenue. Les méthodes exposées dans cette section sont décrites dans le Tableau 2-5.

Tableau 2-5: Méthodes de vérification des réponses en cours de simulation

Méthode	Description

Comparaison avec une référence	Cette méthode consiste à être en possession d'un fichier contenant les réponses espérées. En cours de simulation, les réponses obtenues sont comparées avec les réponses espérées contenues dans le fichier.
Modèle de référence et fonction de transfert	Un modèle de référence ou une fonction de transfert existe afin de représenter une version fonctionnelle du modèle. Les tests sont appliqués sur le modèle et la référence. Ensuite, les réponses sont comparées automatiquement.
Assertions	Des expressions logiques statiques ou temporelles sont définies à l'interne ou à l'externe du modèle. Un outil est utilisé pour vérifier que les expressions logiques sont respectées en tout temps lors de la simulation.
Analyseur de signature	Cette méthode est semblable à l'utilisation d'un *Mutiple Inputs Shift Register* (MISR) utilisé pour le test intégré au niveau circuit. On est en possession d'un fichier contenant les signatures associées à une suite de tests. En introduisant les réponses du modèle dans un MISR, on obtient une signature qu'on compare à une signature de référence. Il est à noter qu'on n'intègre pas le MISR dans le modèle, mais seulement dans le banc d'essais.

2.3 Conception de bancs d'essais

Un banc d'essais est utilisé afin de réaliser la vérification d'un modèle en cours de simulation. Les bancs d'essais nécessaires à l'effort de vérification sont spécifiés dans le plan de vérification. Les bancs d'essais implantent les techniques de vérification utilisées.

Dans cette section, nous survolerons certains sujets relatifs à la conception de banc d'essais. En premier lieu, les langages utilisés pour la vérification seront exposés. Ensuite, les éléments généralement inclus dans un banc d'essais seront détaillés. Finalement, nous terminerons avec une discussion sur la réutilisation qui est effectuée dans le cadre de la conception de banc d'essais.

2.3.1 Langage de vérification

La réalisation d'un banc d'essais requiert l'utilisation d'un langage de programmation. Plusieurs options sont disponibles pour réaliser cette tâche. Il n'est pas recommandé de produire un banc d'essais synthétisable[1], car ce type d'implantation prend beaucoup de temps à simuler. Ainsi, une approche comportementale est la méthode à adopter pour la réalisation d'un banc d'essais. Donc, un banc d'essais est en fait plus un logiciel qu'un modèle matériel. Cependant ce logiciel doit être en mesure d'interagir facilement avec un modèle matériel ce qui implique que le langage doit supporter la

[1] Un banc d'essais synthétisable est cependant nécessaire lorsqu'on veut simuler le banc d'essais et le modèle dans un accélérateur matériel.

notion de concurrence, d'événement et de type binaire. En fait, un banc d'essais peut être considéré comme un système temps réel dominé par le contrôle du modèle sous vérification.

Les langages de conception de matériel tel que VHDL et Verilog sont grandement utilisés afin d'implanter les bancs d'essais. Ces langages sont adéquats pour la réalisation de banc d'essais. Cependant, en considérant un banc d'essais comme un logiciel, les langages HDL ne possèdent pas les attributs nécessaires à une implantation efficace d'un logiciel. Par exemple, un langage HDL ne supporte pas de structures de données qui permettent d'adopter une stratégie de réutilisation. D'autre part, les langages orientés objets, tel que C++ ou Java, possèdent les attributs nécessaires pour le développement de logiciel, mais ils ne supportent pas naturellement les éléments nécessaires pour interagir avec un modèle matériel. *SystemC* est une bibliothèque de classes en C++ qui a été développée pour permettre la modélisation de systèmes autant matériel que logiciel. *SystemC* est une bonne alternative pour la création de bancs d'essais, puisque c'est toujours du C++ et en plus, ce langage fournit un support pour la description de matériel. D'autre part, il a été remarqué que pour la création de bancs d'essais, il serait préférable d'avoir certains mécanismes supplémentaires qui facilitent les tâches de vérification. Ainsi, les HVL ont été développés spécifiquement pour la création de bancs d'essais. Parmi ces langages, on retrouve le langage *e*, OpenVera et *TestBuilder*. *TestBuilder* n'est pas un langage en soi, mais une bibliothèque de classes en C++ tout comme *SystemC*. Il est à noter que tous ces langages sont ouverts au public sauf le langage *e*. Cependant le langage *e* est engagé dans un processus de normalisation de l'*Institute of Electrical and Electronics Engineers* (IEEE) qui ouvrira le langage au public. Les mécanismes de vérification fournis avec les HVL sont exposés dans le Tableau 2-6

Tableau 2-6: Mécanismes de vérification fournis par les HVL

Mécanismes	Descriptions
Générateur Pseudo-Aléatoire	Un générateur pseudo-aléatoire est fourni. La génération des tests est effectuée en accord avec des contraintes de génération décrites dans le banc d'essais.
Algèbre temporelle	Une algèbre temporelle permet de définir des événements complexes qui sont évalués sur une période de temps. De plus, un vérificateur interne permet de vérifier des assertions définies en se basant sur des événements
Couverture	Des modèles de couvertures peuvent être définis, c'est-à-dire, qu'en se basant sur

	une technique de vérification, on peut définir ce qui doit être recueilli pendant une simulation afin de mesurer la métrique utilisée. De plus, un système d'échantillonnage interne est fourni pour échantillonner les éléments définis dans le modèle de couverture.

L'annexe A expose en détail les mécanismes de vérification d'un langage de vérification spécifique: le langage *e*.

2.3.2 Implantation d'un banc d'essais

Un banc d'essais sert à implanter une stratégie de test définie dans le plan de vérification. Un banc d'essais doit permettre la réalisation des techniques de vérification utilisées et aussi la vérification des réponses.

Ainsi, dans un banc d'essais, on retrouve habituellement un module servant à l'application des tests sur le modèle et un module servant à l'extraction des réponses. Dépendamment du niveau d'abstraction de la technique de vérification, il peut être nécessaire d'avoir ce qu'on nomme communément un *Bus Functional Model* (BFM) afin d'appliquer les tests. Un BFM est un module qui prend une structure de données définie à haut niveau d'abstraction et applique la structure de données sur le modèle. L'application se fait en respectant le protocole d'application de la structure définie autant au niveau binaire que temporel. Par exemple, on applique des paquets sur une unité de routage. Un BFM prendra les paquets, les transformera en vecteurs binaires et les appliquera sur le modèle en respectant le protocole temporel défini pour l'application d'un paquet. L'inverse est semblable pour recueillir les réponses du modèle.

Un module est nécessaire afin de produire les tests. Dans le cas où les tests seraient produits à l'interne du banc d'essais, tel qu'avec une approche de génération pseudo-aléatoire, un module est nécessaire afin de produire les tests. Autrement, les tests peuvent être décrits de façon déterministe dans le module ou simplement chargés à partir d'une source externe.

Dans le cas où une approche de vérification des réponses en cours de simulation est adoptée, il sera nécessaire d'implanter un ou des modules réalisant cette tâche. D'autre part, il est possible qu'on inclue les éléments de vérification directement dans les autres

modules du banc d'essais. Par exemple, on veut vérifier que le protocole temporel utilisé aux sorties du modèle est en respect avec la spécification. On peut donc inclure dans le BFM servant à recueillir les réponses, des assertions qui permettront de vérifier ces propriétés du modèle. Maintenant, pour ce qui est de mesurer la couverture des métriques définies par les techniques de vérification utilisées, il est nécessaire de développer des modules qui permettront de mesurer l'avancement de la couverture. Dans le cas où on utiliserait seulement des tests déterministes, il n'est pas nécessaire d'avoir ce type de module puisqu'on sait d'avance ce qu'on couvre. Cependant, une approche pseudo-aléatoire ou automatique requiert ce type d'élément.

2.3.3 Réutilisation

La réutilisation s'applique à tous les niveaux dans la vérification : l'élaboration du plan de vérification, la conception des bancs d'essais ou encore la documentation relative à la vérification. Des procédures formelles doivent être définies pour permettre la réutilisation à tous les niveaux des étapes de la vérification. Pour la conception des bancs d'essais, il y a deux types de réutilisation possibles: dans un même projet et à travers d'autres projets. Dans un même projet, on peut réutiliser les composants qu'on a conçus pour la vérification de sous-module d'un système complet et ensuite réutiliser cela lors des tests d'intégration. Pour ce qui est de la réutilisation à travers d'autres projets, les bancs d'essais d'un circuit peuvent être réutilisés pour une nouvelle génération du circuit. De plus, pour des spécificités qu'on retrouve dans des projets semblables, tel que des protocoles standardisés, plusieurs composants peuvent être réutilisés.

2.3.3.1 Composants réutilisables et patron de conception

Il existe sur le marché des composants réutilisables dédiés à la vérification. Ces composants sont développés afin d'être utilisés dans un banc d'essais avec un minimum de modifications. On retrouve beaucoup de composants réutilisables pour les protocoles utilisés dans la conception des circuits numériques. Par exemple, il y a des composants disponibles pour les normes suivantes: Ethernet, PCI, AMBA, PowerPC, I2C et FireWire.

Autrement, une société peut adopter une méthodologie de réutilisation afin de produire des composants réutilisables en fonction de ses besoins.

Un patron de conception est un autre artefact permettant la réutilisation [16]. Les patrons de conception définissent une méthode systématique de conception d'un certain composant. Un patron de conception n'est pas directement réutilisable, car il doit être configuré afin d'être utilisé. Un exemple de patron de conception est le BFM. On utilise des BFM dans plusieurs bancs d'essais. Cependant, nous ne pouvons réutiliser les BFM dans tous les bancs d'essais, mais la manière de le faire est réutilisable en l'adaptant à nos besoins. Habituellement, on construit un banc d'essais en utilisant des patrons de conception, mais sans vraiment le formaliser. En formalisant les patrons de conception pour la vérification, on est donc en mesure d'effectuer une réutilisation efficace des patrons de conception.

Chapitre 3

MÉTHODOLOGIE DE CONCEPTION DE BANCS D'ESSAIS AXÉE SUR LA RÉUTILISATION

La réutilisation à tous les niveaux des étapes de développement d'un circuit permet d'obtenir un gain de productivité. Ce gain de productivité vise à réduire le temps d'entrée en marché du produit. La vérification d'un circuit étant maintenant considérée comme le goulot d'étranglement du développement d'un circuit, il est nécessaire de s'attaquer à ce problème en révisant les méthodes de travail établies. La réutilisation au niveau de la vérification requiert la définition de méthodologies structurées. Cependant, la définition d'une méthodologie est inutile si on ne l'applique pas avec un certain formalisme. Les intentions de réutilisation dans un projet de vérification ne mènent à rien si elles ne sont pas appliquées.

Dernièrement, de nouveaux produits d'aide à la conception de circuits numériques ont été mis sur le marché spécifiquement pour la vérification fonctionnelle. Parmi ces produits, on retrouve les langages de vérification matérielle qui sont des outils qui rehaussent le niveau d'abstraction utilisé pour concevoir les bancs d'essais. Les langages de vérification étant des langages orientés objets, cela ajoute une nouvelle dimension à la conception des bancs d'essais. Plus précisément, les principes de l'analyse et de la conception orientée objets au niveau logiciel s'appliquent maintenant à la conception de bancs d'essais. Haque & Al. [19] et Whittemore & Al. [39] ont démontré une utilisation efficace de l'approche orientée objets pour la construction d'un environnement de vérification. Leurs approches permettent de rehausser le niveau de réutilisation de leurs bancs d'essais. Par contre, au niveau de la conception logiciel, Kiczales & Al. [23] ont indiqué quelques lacunes au paradigme orienté objets lorsqu'on veut modéliser certaines propriétés d'un système qui influencent plusieurs objets. Ils proposent donc une nouvelle technique de programmation appelée la programmation orientée aspects qui permet de capturer les aspects d'un système. La programmation orientée aspects permet d'obtenir

une meilleure modularité du logiciel et ainsi, un plus grand potentiel de réutilisation. Néanmoins, l'approche orientée aspects n'a jamais été explicitement présentée pour la réalisation d'un environnement de vérification.

Théoriquement, le meilleur test qu'un circuit peut passer est d'être mis dans son environnement opérationnel et ensuite d'observer s'il fonctionne correctement. Cependant, il est impossible de reproduire un tel état dans un environnement de simulation. Néanmoins, un environnement de vérification est tout de même requis. Un environnement de vérification est dit efficace s'il possède les caractéristiques suivantes :

- *Émuler l'environnement du modèle sous vérification;*
- *Vérifier automatiquement les réponses produites par le modèle en fonction des tests appliqués;*
- *Fournir de l'information sur la progression de la couverture des métriques utilisées;*
- *Être facilement configurable afin de permettre l'écriture de cas de tests par des ingénieurs ne connaissant pas les détails d'implantation.*

Parmi ces caractéristiques, on doit être en mesure d'appliquer des techniques de vérification choisies et de penser en terme de réutilisation.

Dans ce chapitre, une méthodologie de conception de bancs d'essais est présentée. La méthodologie formalise l'utilisation du paradigme orienté objets et aspects afin de partitionner les différents modules nécessaires à la création d'environnement de vérification. Cette méthode est générique et tend à favoriser la réutilisation. De plus, le partitionnement proposé facilite les adaptations nécessaires à un banc d'essais afin de réaliser tous les cas de test définis dans le plan de vérification. La méthodologie sera exposée à l'aide d'un exemple concret. Le langage de vérification *e* a été utilisé afin d'appliquer la méthodologie. La méthodologie a été présentée dans deux communications de conférence : la méthodologie en tant que tel dans [29] et l'application de la méthodologie avec le langage *e* est disponible dans l'annexe B [30].

Dans ce présent chapitre, la section 3.1 présente une revue des concepts orientés objets et aspects qui sont les bases de la méthodologie. Puis, la section 3.2 détaille la

méthodologie de conception de banc d'essais. À la section 3.3, nous présenterons un exemple afin de démontrer l'utilité de la méthode. Finalement, la section 3.4 présente une discussion sur la méthodologie, en présentant certaines remarques et extensions possibles. Il est à noter que dans ce chapitre, il ne sera pas question de l'implantation concrète de la méthode avec le langage *e*. Cependant, l'annexe B apporte un complément technique pour l'application de la technique avec le langage *e*.

3.1 Conception logiciel

Dans cette section, une introduction aux principes de conception logicielle utilisés par notre méthodologie est présentée. Les principes de base de la conception orientée objets et aspects seront exposés.

3.1.1 Conception Orientée Objets

Les principes de la conception orientée objets ont été résumés par Pressman [27]. La conception orientée objets est une méthode de conception utilisée en génie logiciel. La conception orientée objets consiste à considérer tous les éléments constituant le système à concevoir comme des objets. Un objet est défini à partir d'une classe. Une classe est en fait une sorte de moule à objets. Une classe définit les attributs et les méthodes appartenant à un objet. Ainsi, une instance d'un objet est un objet unique formé à partir des caractéristiques de la classe utilisée pour créer l'objet. Dans un contexte de réutilisation, des bibliothèques de classes sont créées afin de permettre la création d'un système à partir d'instance d'objets qu'on crée en utilisant les classes définies dans la bibliothèque. Un exemple de bibliothèque de classe est le *SystemC* [17]. *SystemC* fournit un ensemble de classes qui permettent de créer principalement un modèle au niveau système. Ainsi, la conception d'un modèle avec *SystemC* requiert l'utilisation des classes de la bibliothèque afin de créer des instances des objets du système. On peut, par exemple, faire une instance d'un objet *module* avec la classe *sc_module*. Un *module* est l'unité permettant de créer la structure du modèle en regroupant des processus et en offrant des interfaces pour la communication entre les différents modules du système et de l'environnement externe. Ainsi dans le module, on peut combiner des ports d'entrées

et de sorties en créant des instances d'objets *port* à l'aide de la classe *sc_port* de la bibliothèque *SystemC*. Cependant, l'utilisation seule des classes d'une bibliothèque ne permet pas de construire un système complet. La conception orientée objets consiste, en fait, à réutiliser les classes contenues dans des bibliothèques, à créer les classes qui manquent pour compléter le système et à assembler le système en créant des instances des objets découlant des classes utilisées.

La programmation orientée objets est devenue la méthode de conception la plus utilisée dans le domaine du génie logiciel dû aux concepts qu'elle apporte soit l'encapsulation, le polymorphisme, l'héritage, l'agrégation, la modularité et la réutilisation.

3.1.2 Conception Orientée Aspects

L'approche orientée aspects [13] est basée sur le principe de séparation des intentions [14]. Un aspect définit un comportement ou une fonctionnalité qui affecte plusieurs classes d'un système. L'analyse orientée objets organise un système en un ensemble concis de classes. Cependant, certains aspects d'un système ne peuvent être définis dans les limites d'une classe, donc nous pouvons dire que ces aspects entrecoupent la modularité du système. L'approche orientée aspects résout ce problème de modularité. Par exemple, les communications entre les objets sont généralement définies dans chaque classe d'un système. Toutefois, en regroupant tous les éléments de communication entre les objets dans un seul aspect qui affecte plusieurs classes, nous pouvons faire sortir tous les éléments de communications des classes originales. Le potentiel de réutilisation est considérablement augmenté, car cette approche force l'isolation de certains aspects d'un système afin d'éviter une dispersion à travers le code. Ainsi, l'approche orientée aspects permet de définir les classes d'un système par tranche, où chaque tranche définit un aspect du système.

La différence entre l'approche orientée objets et aspects peut être expliquée avec un exemple simple qui consiste en la modélisation de la carrosserie d'une voiture. La Figure 3-1 présente un diagramme de classe exprimé avec le *Unified Modeling Language*

(UML) [8] qui expose les classes définies avec une approche orientée objets pour une carrosserie de voiture.

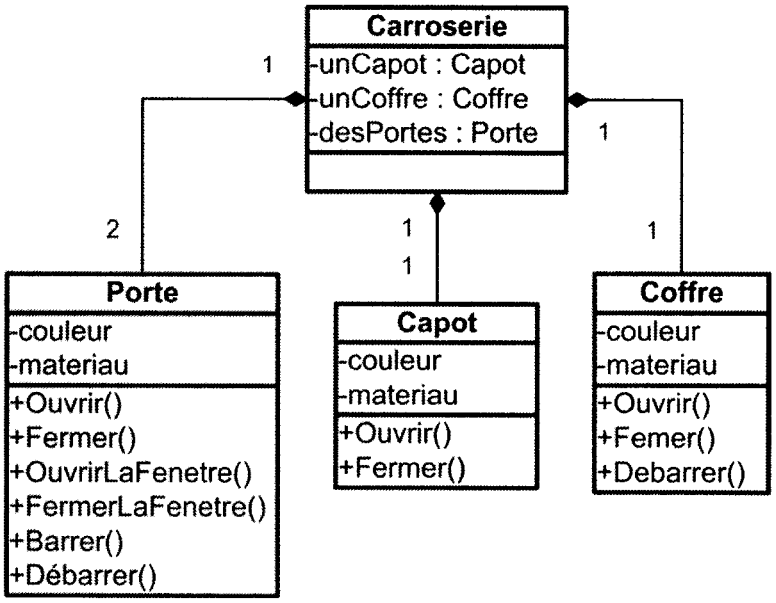


Figure 3-1: Diagramme UML orienté objets

Ainsi, des classes sont définies pour représenter une porte, un capot et le coffre arrière. Ces classes sont agrégées par la super-classe, *Carrosserie*. Maintenant, en partant de cette modélisation orientée objets, nous allons ajouter le concept d'aspect. Afin de rendre plus réutilisable les classes utilisées pour composer la carrosserie, nous pouvons extraire certains aspects communs à toutes ou à plusieurs classes. Par exemple, la porte, le capot et le coffre ont des caractéristiques communes, telles que la couleur et le matériau utilisé pour la conception de ces objets. Ce type de caractéristiques communes peut être modélisé comme un aspect de la carrosserie.

La Figure 3-2 présente un diagramme de classe UML incluant un aspect pour certaines caractéristiques. Il est à noter que le concept d'aspect ne fait pas partie de la norme UML. Cependant, la notation utilisée a été proposée par Suzuki [32].

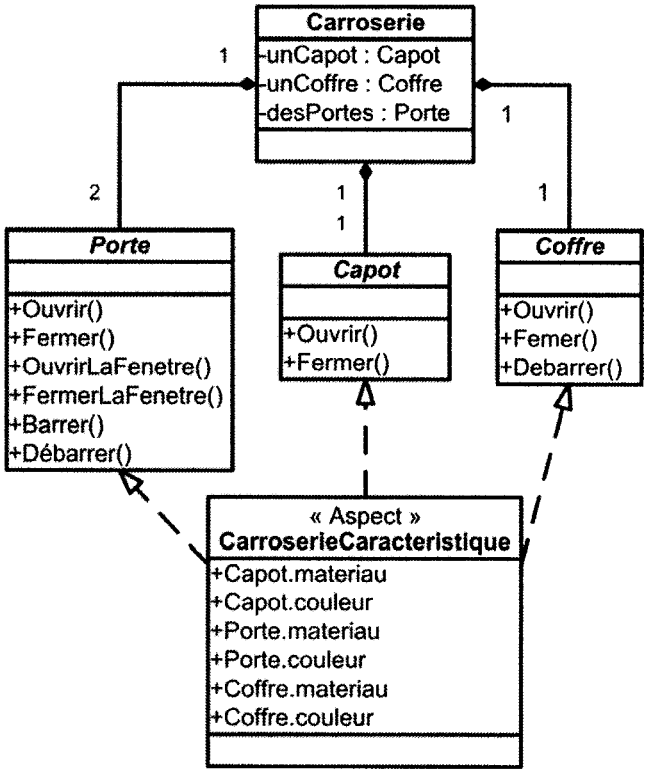


Figure 3-2: Diagramme UML orienté aspects

En ayant retiré les attributs couleur et matériau des classes *Porte*, *Coffre* et *Capot*, nous avons maintenant des classes abstraites puisque les objets produits à partir de ces classes ne peuvent exister sans ces caractéristiques. Un aspect *CarrosserieCaractéristique* est défini pour représenter un attribut ou une fonctionnalité affectant plusieurs classes du système. En l'occurrence, l'aspect *CarrosserieCaractéristique* définit la couleur et le matériau des classes *Porte*, *Capot* et *Coffre*. Ainsi, en ajoutant cet aspect à ces classes abstraites, nous avons le système équivalent qui est cependant modélisé différemment.

Maintenant, on peut comparer la réutilisation des classes modélisées avec une approche orientée objets et une approche orientée objets et aspects. On veut réutiliser cet ensemble de classes, mais en sachant que l'attribut *Couleur* doit être modifié. Avec une approche orientée objets, on ne peut pas réutiliser directement les classes puisque des modifications sont nécessaires dans trois classes. De plus, ces modifications sont les mêmes dans chaque classe. Dans le cas de l'approche orientée aspects, l'ensemble de

classes est directement réutilisable. Cependant, l'aspect *CarrosserieCaractéristique* doit être redéfini afin de configurer les classes pour les besoins précis de l'application. Dans ce cas-ci, un aspect n'est pas utilisé afin d'être réutilisé, mais plutôt afin de permettre que le reste du système soit davantage réutilisable. Ainsi, la réutilisation a été facilitée puisque l'élément non réutilisable a été défini dans un aspect.

L'utilisation des principes orientés objets et aspects peuvent être très avantageux s'ils sont bien utilisés. La prochaine section expose une méthodologie qui utilise ces concepts, dans le contexte de la conception de bancs d'essais.

3.2 Méthodologie proposée

La méthodologie de conception de bancs d'essais proposée est, en fait, un partitionnement orienté objets et aspects dédié à la conception de bancs d'essais. Le partitionnement est particulièrement efficace dans un contexte de conception pour la réutilisation. Ainsi, dans cette section, le partitionnement par objets sera présenté suivi du partitionnement par aspects.

3.2.1 Partitionnement par objets

Une décomposition orientée objets permet d'obtenir un haut niveau de réutilisation. Dans notre cas, notre méthode de décomposition propose trois types de classes qu'on retrouve dans un banc d'essais. Voici la description de chaque type de classes :

Classes d'émulation

Une classe d'émulation sert à produire le même comportement qu'un élément matériel réel devrait produire à une interface du modèle. De plus, les classes d'émulation doivent être configurables afin de produire tous les cas de test définis dans le plan de vérification. Un exemple d'une classe d'émulation est un injecteur de paquets qui génère un flot continu de paquets vers le modèle.

Classes de vérification

Les classes de vérification comprennent les classes ayant la responsabilité de rendre l'environnement de vérification auto-vérifiant. Par exemple, une classe de

vérification d'assertions qui vérifie si certaines règles comportementales du modèle sont respectées fait partie des classes de vérification.

Classes utilitaires

Les classes utilitaires comprennent les classes qui ne sont pas incluses dans les deux catégories précédentes. Nous pouvons inclure dans cette catégorie des classes qui représentent, par exemple, des structures de données utilisées pour modéliser un paquet.

La Figure 3-3 présente un banc d'essais auquel le partitionnement par objets est appliqué.

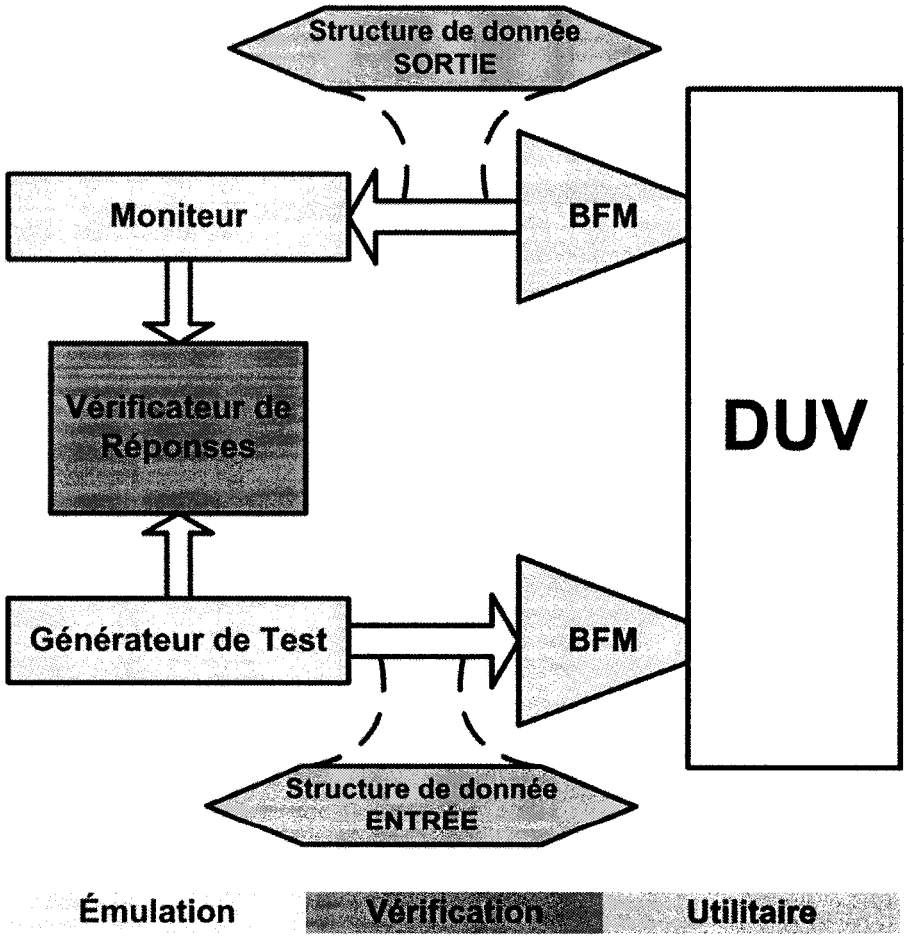


Figure 3-3: Exemple du partitionnement par objets

Ainsi, sur ce banc d'essais, on remarque qu'on a des classes d'émulation tel qu'un générateur de tests et un moniteur qui observe les réponses produites par le DUV. Le générateur et le moniteur fonctionnent à un haut niveau d'abstraction, donc des BFM sont ajoutés afin de niveler le niveau d'abstraction de ces éléments avec le DUV. Un vérificateur de réponses est défini et fait partie des classes de vérification. Des modèles de structures de données à haut niveau sont utilisés par les BFM afin d'appliquer des vecteurs binaires sur le DUV.

3.2.2 Partitionnement par aspects

Tel que mentionné précédemment, une analyse orientée aspects permet d'obtenir une séparation concise d'un environnement de vérification en définissant toutes les classes de l'environnement de vérification par tranche. La décomposition de l'environnement de vérification avec l'aide d'aspects est l'élément au cœur de la méthode. Premièrement, il est important de déterminer ce que nous voulons retirer dans notre partitionnement par aspect. Dans notre cas, l'objectif est de rehausser le niveau de réutilisation d'un environnement de vérification. Donc, les aspects ayant un faible potentiel de réutilisation doivent être isolés. Une formalisation des aspects qu'on retrouve lors de la conception d'un banc d'essais est proposée. Cinq catégories d'aspects sont définies afin de permettre une réutilisation optimale d'un banc d'essais. La Figure 3-4 présente les différentes catégories définies.

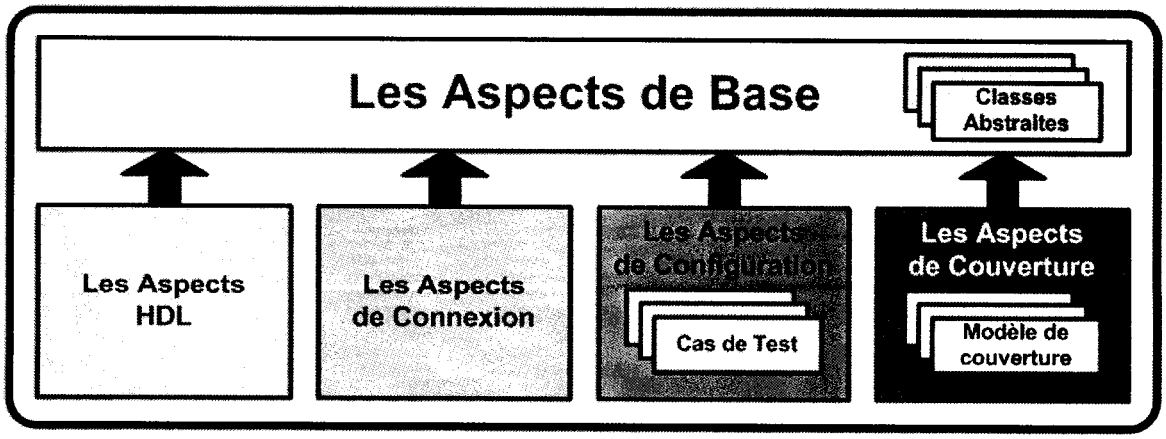


Figure 3-4: Partitionnement par aspects

Tel que nous pouvons le voir sur la Figure 3-4, au centre de notre partitionnement par aspects on retrouve les aspects de base. Durant l'analyse orientée objets, nous avons indiqué que certaines classes étaient nécessaires pour définir notre environnement de vérification. La catégorie des aspects de base inclut, en fait, la définition de toutes ces classes, mais en n'ayant pas défini les détails isolés dans les autres aspects. Ainsi, toutes les classes qu'on retrouve dans la catégorie des aspects de base sont abstraites, puisque qu'elles ne sont pas complètes. Les aspects des autres catégories apportent une extension aux classes abstraites. Ces extensions sont effectuées en ajoutant des attributs et des fonctionnalités aux classes de base. Ces fonctionnalités ont généralement un faible potentiel de réutilisation et ainsi, en les séparant dans des aspects, nous purgeons les classes des éléments non-réutilisables. Voici une description des catégories d'aspects définies afin de complémenter les classes abstraites définies dans les aspects de base.

Les Aspects HDL

Un aspect HDL sert à regrouper tout ce qui touche le DUV. Les connexions et interactions avec les interfaces du modèle sont définies par ces aspects. Ainsi, des changements sur les interfaces ou sur les interactions du modèle n'affecteront que cet aspect.

Les Aspects de Connexion

Un aspect de connexion sert à définir les connexions et les interactions entre les classes définies pour un banc d'essais, c'est-à-dire que les différents éléments d'un banc d'essais doivent être connectés ensemble afin de pouvoir fonctionner. Cependant, en définissant ces détails directement dans un élément, le contexte d'utilisation de cet élément peut être diminué.

Les Aspects de Couverture

Un aspect de couverture sert à définir les éléments dans les bancs d'essais qui servent à mesurer la couverture de la métrique utilisée. Ainsi, le modèle de couverture utilisée lors d'une simulation sera indépendant du banc d'essais.

Les Aspects de Configuration

Un aspect configuration sert à configurer l'environnement de vérification afin de

produire un test précis. Ainsi, pour l'application de différents tests, on réutilise l'environnement de vérification et une série d'aspects de configuration est définie afin d'implanter tous les tests définis par le plan de vérification.

3.3 Application de la méthodologie

Dans cette section, une application de la méthodologie est exposée afin de permettre une meilleure compréhension de la méthode. Cependant, les résultats de l'utilisation de la méthodologie seront détaillés au Chapitre 5. Le modèle sous vérification utilisé pour cet exemple sera d'abord présenté. Ensuite, le banc d'essais développé en utilisant la méthodologie sera exposé. Finalement, il sera expliqué en quoi le banc d'essais développé possède les qualités d'un environnement réutilisable.

3.3.1 Le modèle sous vérification

Le DUV est un module d'une plate-forme de conversion de protocoles en développement au GRM de l'École Polytechnique de Montréal. Ce système est conçu en vue d'effectuer la conversion de plusieurs protocoles. Le projet se concentre présentement sur les protocoles pouvant supporter des flots vidéo numériques. La première itération du système est configurée et programmée pour effectuer la conversion Firewire (IEEE 1394.b) à Ethernet (IEEE 802.3). La Figure 3-5 présente une partie du système de conversion de protocole.

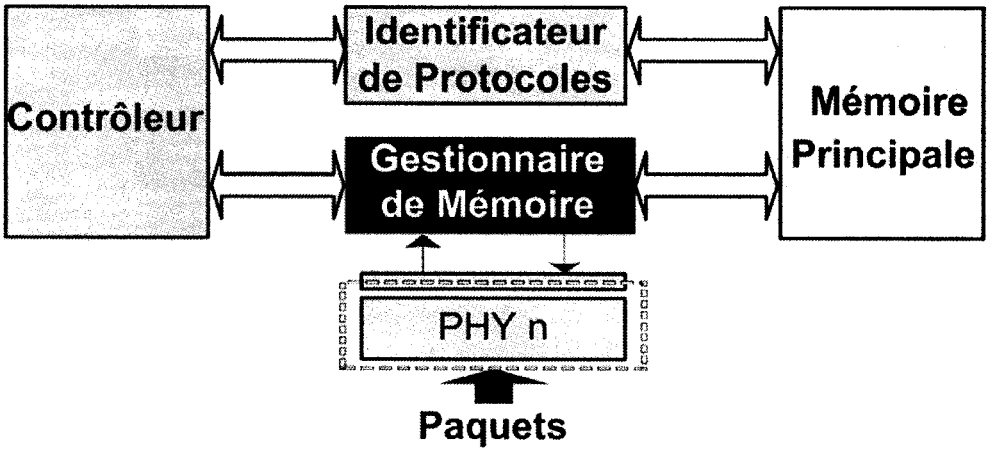


Figure 3-5: Bloc 1 du convertisseur de protocoles

Le module sur lequel nous avons concentré notre effort de vérification est le Gestionnaire de Mémoire. Ce module sert à gérer l'allocation de la mémoire pour les paquets entrant et sortant du système de conversion. La mémoire du système de conversion peut contenir cinq paquets. Lorsque la mémoire est pleine, le système n'accepte plus de paquet. Lors de la réception d'un paquet, le Gestionnaire de Mémoire le place en mémoire à une certaine adresse. De plus, le Gestionnaire de Mémoire envoie une étiquette au Contrôleur pour lui indiquer l'adresse où le paquet a été placé. Ensuite, un certain traitement est effectué sur le paquet placé en mémoire afin d'effectuer la conversion de protocoles. À la fin de ce traitement, le Contrôleur indiquera au Gestionnaire de Mémoire que l'espace mémoire alloué au paquet en question peut être libéré et donc être allouée à un autre paquet.

3.3.2 Conception d'un banc d'essais

La Figure 3-6 présente une vue fonctionnelle de l'environnement de vérification développé pour le Gestionnaire de Mémoire. Elle montre les classes principales utilisées pour stimuler le DUV et pour vérifier son comportement.

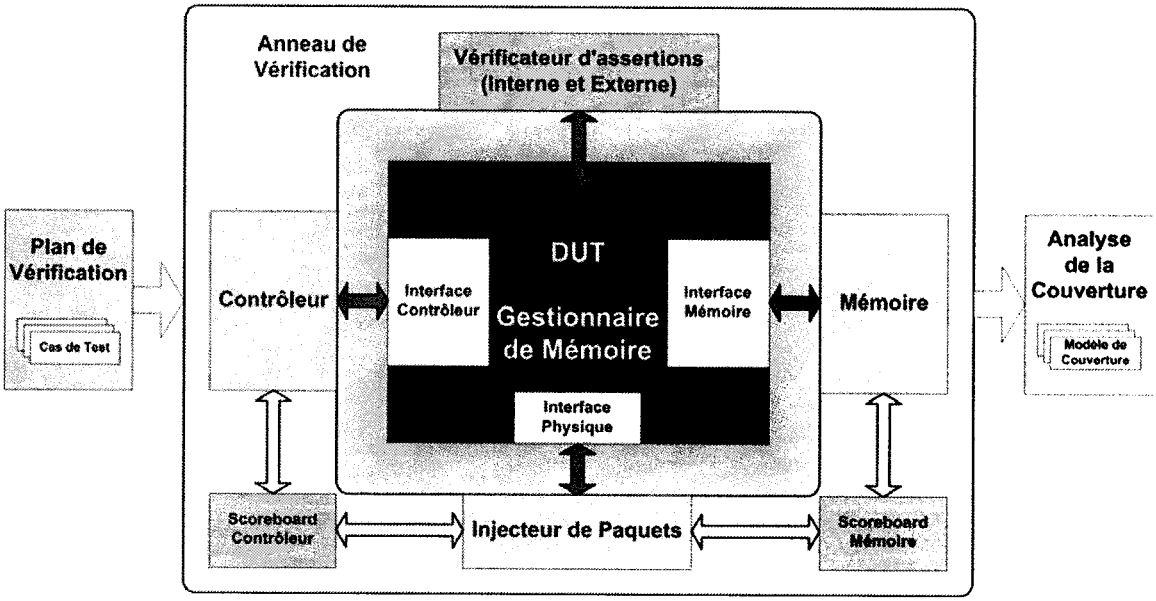


Figure 3-6: Vue fonctionnelle du banc d'essais

Ainsi, nous avons trois classes d'émulation: l'Injecteur de Paquets, le Contrôleur et la Mémoire. Le module Injecteur de Paquets sert à générer des paquets de type Firewire. Le module Mémoire émule le comportement de la mémoire utilisée dans le système de conversion de protocoles. Finalement, le module Contrôleur permet d'émuler le comportement du système de conversion. Plus précisément, le Contrôleur reçoit des étiquettes indiquant la location du paquet en mémoire. Ensuite, le contrôleur simule un temps de conversion pour chaque paquet entré dans le système. Suite à l'expiration du temps de conversion simulé, le Contrôleur indique au Gestionnaire de Mémoire la suppression du paquet en question. De cette façon, le Gestionnaire de Mémoire est placé dans un environnement imitant l'environnement opérationnel dans lequel il est supposé fonctionner. Pour ce qui est de la vérification des réponses du DUV, trois classes de vérification ont été définies : un *Scoreboard* pour le Contrôleur, un *Scoreboard* pour la mémoire et un vérificateur d'assertions. Les modules *Scoreboard* servent à s'assurer que les paquets appliqués par l'Injecteur de Paquets ont bien produit les effets désirés aux modules Contrôleur et Mémoire. Le vérificateur d'assertions sert à vérifier en tout temps des règles comportementales extraites de la spécification du Gestionnaire de Mémoire. Finalement, le banc d'essais est complété avec quelques classes utilitaires tel que la modélisation des paquets appliqués sur le DUV et l'Anneau de Vérification. L'Anneau de Vérification est une classe qui sert à combiner toutes les autres classes du banc d'essais.

L'élément distinctif de notre méthodologie est la séparation des aspects de l'environnement de vérification. La Figure 3-7 présente, en partie, le même banc d'essais sous l'angle des aspects dans notre environnement.

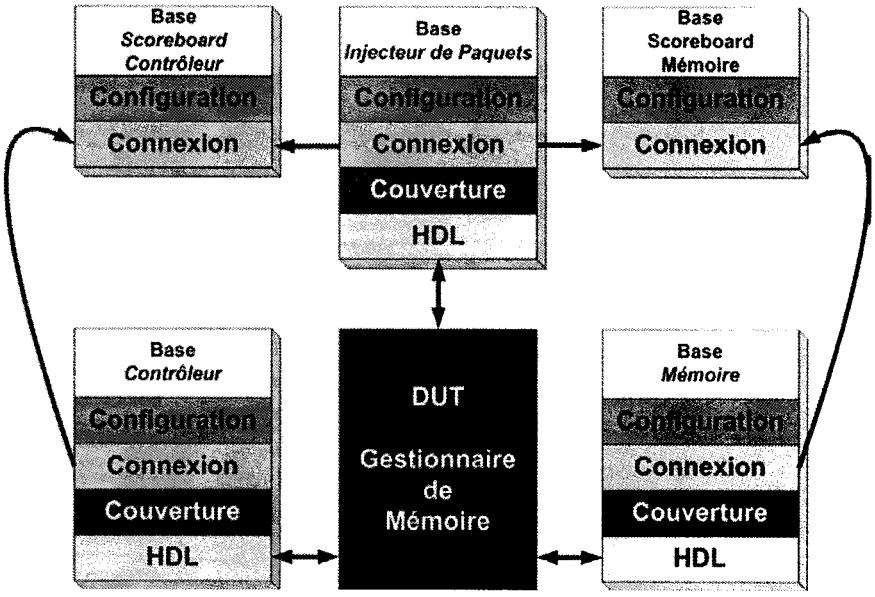


Figure 3-7: Vue fonctionnelle avec aspects

Ainsi, chaque classe est définie avec une base unique qui représente ses fonctionnalités. Ensuite, les classes sont complétées avec les aspects que nous avons définis dans notre partitionnement. Cependant, les aspects ne sont pas répartis dans chaque classe tel que montré à la Figure 3-7. En fait, la Figure 3-8 présente une meilleure vue de la structure logicielle de la conception du banc d'essais avec les aspects.

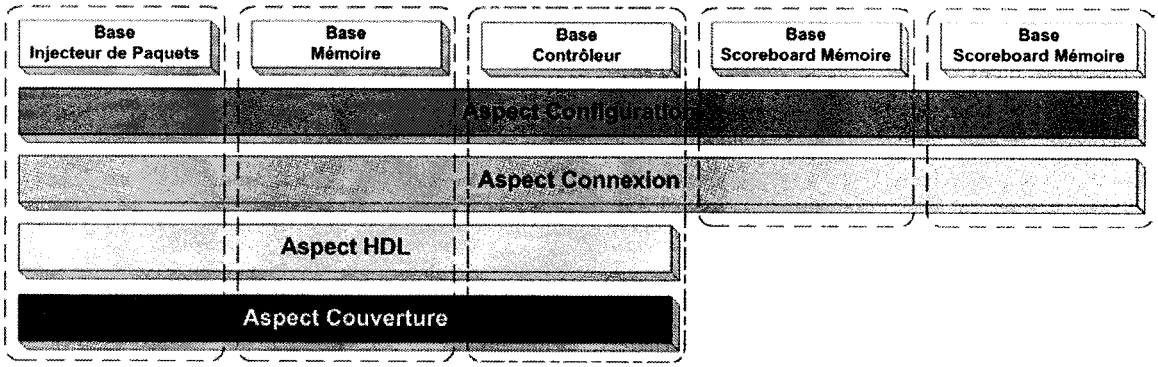


Figure 3-8: Vue structurelle du banc d'essais avec aspects

Donc, on remarque que la modélisation par aspects implique qu'un seul aspect de chaque catégorie est défini et affecte presque toutes les classes définies pour le banc d'essais. Les lignes pointillées représentent la recomposition qui sera effectuée afin de former les modules du banc d'essais. Il est à noter que tous les aspects définis par notre

partitionnement ne sont pas obligatoires pour chaque classe. Ainsi, il n'y a pas d'aspect HDL pour les classes *Scoreboard* puisque ces classes ne sont pas connectées au modèle HDL.

3.3.3 Réutilisation

L'utilisation de la méthodologie de conception de banc d'essais permet de faciliter la réutilisation à plusieurs niveaux. En premier lieu, si on veut réutiliser des parties du banc d'essais dans un même projet, la méthodologie supporte plusieurs éléments recherchés à cette fin.

Adaptation

Les adaptations nécessaires sur un banc d'essais sont circonscrites dans les catégories d'aspects définies par la méthodologie. Par exemple, dans le cas où les signaux aux interfaces du modèle seraient modifiés ou que le protocole d'interaction serait aussi révisé, les modifications nécessaires peuvent être réparties dans plusieurs modules du banc d'essais. Cependant, au niveau de la conception du banc d'essais, ces éléments seront circonscris dans l'aspect HDL.

Extension

La vérification d'un système s'effectue souvent en utilisant une approche ascendante. On vérifie les sous-modules d'un système et ensuite, on intègre les sous-modules pour vérifier leur intégration. En utilisant cette approche pour le système de conversion de protocoles, on peut intégrer le module Contrôleur avec le Gestionnaire de Mémoire. La Figure 3-9 présente un banc d'essais dédié à la vérification de ces deux modules où la modularité du précédent banc d'essais permet de faciliter la réutilisation.

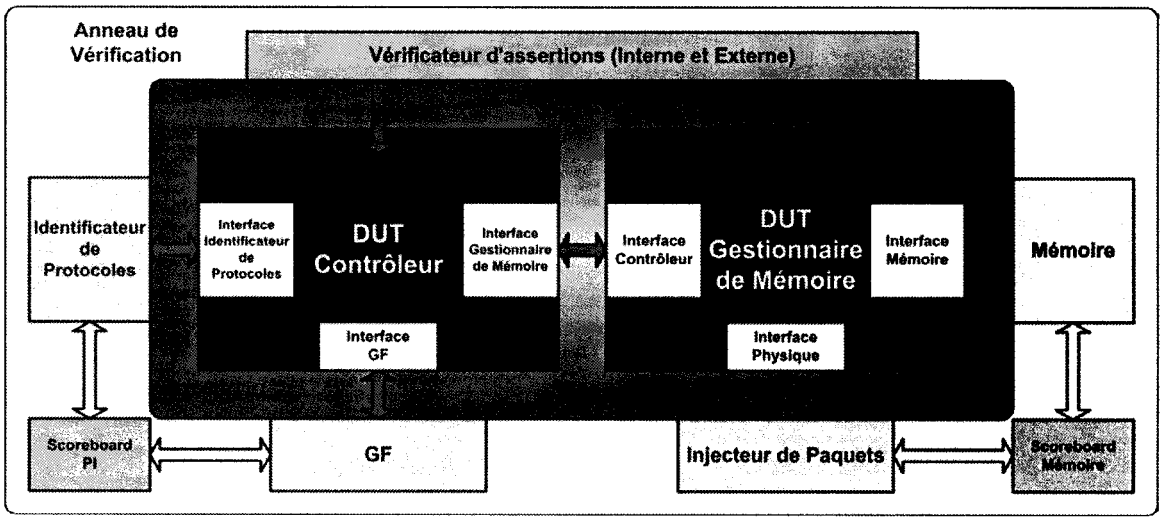


Figure 3-9: Adaptation du banc d'essais

À la Figure 3-9, la classe d'émulation du contrôleur est remplacée par le module dans une version HDL. Dans le développement de ce banc d'essais, plusieurs parties ont pu être réutilisées directement dû à l'utilisation de la méthodologie de conception. Plusieurs classes de base ont été réutilisées tel que la Mémoire et l'Injecteur de Paquets. Les nouveaux éléments requis pour ce banc d'essais sont définis toujours en utilisant la méthode. Un nouvel aspect de connexion a dû être défini afin de spécifier les nouvelles connexions et interactions entre les modules de ce banc d'essais. De plus, un nouvel aspect HDL est nécessaire afin de définir les nouvelles connexions et interactions avec le DUV.

Maintenance

Il est fréquent de créer un circuit numérique sur la base d'une version précédente d'un circuit. Ainsi en réutilisant un circuit numérique, on peut aussi réutiliser le banc d'essais dédié à sa vérification. La méthodologie de conception proposée est donc efficace pour la maintenance des bancs d'essais.

3.4 Discussion

Dans cette section, certaines remarques sur la méthodologie sont présentées. Ensuite, des extensions permettant de complémenter la méthodologie de conception sont exposées.

3.4.1 Remarques sur la méthode

Dans cette sous-section, trois commentaires relatifs à la méthode sont présentés :

L'importance de suivre les principes

Une approche orientée aspects combinée avec une approche orientée objets permet d'avoir une meilleure modularité de l'environnement de vérification et ainsi, cela facilite les tâches de maintenance. Par contre, il est nécessaire de respecter des principes de programmation stricts si on veut obtenir du succès avec cette méthode. De plus, les principes de partitionnement énoncés par la méthodologie doivent être suivis. Par exemple, une violation du partitionnement au niveau de l'aspect HDL peut endommager sérieusement la modularité établie dans l'environnement de vérification.

Conception orthogonale

La conception d'un banc d'essais est un travail d'équipe. La méthodologie proposée est particulièrement efficace pour la conception orthogonale des bancs d'essais. En fait, plus le niveau de modularité du système est élevé plus il est facile de partitionner le travail à travers une équipe de conception. Ainsi, suite à une définition concise des classes de base utilisées dans le banc d'essais, il est possible de répartir la conception des différents aspects du banc d'essais à travers une équipe de conception.

Documentation

Un document de conception est toujours nécessaire pour permettre un suivi et une compréhension d'un système modélisé. Une approche orientée aspects favorise une plus grande modularité, mais il est évident que cette approche est nettement moins compréhensible au niveau de la structure du code qu'un style de programmation classique tel que l'orienté objets. Donc, une documentation solide est requise afin de palier à cet inconvénient. L'utilisation du UML peut être fortement appréciée afin d'accomplir cette tâche.

3.4.2 Extension de la méthodologie

La méthodologie proposée est une méthode de conception de bancs d'essais générale qui permet d'isoler les éléments non-réutilisables d'un banc d'essais. Ainsi, un complément à cette méthode est d'automatiser la création de ces éléments non-réutilisables. Par exemple, l'automatisation de l'aspect HDL dérive directement du DUV. Ainsi, un outil automatique prenant en entrée les sources du DUV pourrait être utile afin de créer une base à la création de l'aspect HDL. Les aspects de configuration et de couverture pourraient être dérivés du plan de vérification. Cela requiert que le plan de vérification soit défini dans un format précis. Particulièrement, l'aspect Couverture dépend de la métrique utilisée pour mesurer l'avancement de la vérification du modèle.

D'autre part, on retrouve dans un banc d'essais des modules qui sont utilisés de façon récurrente. Par exemple, des vérificateurs d'assertions ou des générateurs de tests sont souvent utilisés dans un banc d'essais. Ces modules étant spécifiques au DUV, ils sont difficilement réutilisables. Cependant, une méthode systématique de conception de ces modules permet d'accélérer leur développement. Ainsi, il est possible de définir des méthodes systématiques de conception de modules de banc d'essais en se basant sur la méthodologie proposée dans ce chapitre.

Chapitre 4

MÉTHODE DE COUVERTURE FONCTIONNELLE

Ce chapitre présente une méthode qui permet d'évaluer la couverture fonctionnelle d'un modèle. Au chapitre précédent, une méthode de conception de banc d'essais axée sur la réutilisation a été présentée. La réutilisation permet d'accélérer le processus de développement des bancs d'essais et d'augmenter la fiabilité des modules utilisés dans un banc d'essais, puisque les modules ont déjà été utilisés et éprouvés lors d'utilisations précédentes. Cependant, la réutilisation ne garantit aucunement la qualité de la vérification effectuée. La qualité d'un banc d'essais est évaluée par sa capacité à détecter des erreurs et à évaluer l'avancement de la vérification. Les méthodes de détection d'erreurs en cours de simulation permettent de créer des bancs d'essais efficaces. D'autre part, la complétude de la vérification est déterminée avec l'aide de métriques. Les métriques permettent de mesurer l'avancement de la vérification en fournissant un taux de couverture quantitatif en fonction de techniques de vérification utilisées. L'utilisation de métriques permet de formaliser la vérification fonctionnelle basée sur la simulation.

L'utilisation de métriques directement sur un système matériel complexe mène facilement à une explosion combinatoire. Une solution connue consiste à appliquer les métriques sur une abstraction du modèle [25]. Une technique prometteuse consiste à quantifier la qualité de la vérification en utilisant des métriques appliquées sur la spécification du modèle [34], en fait une spécification exécutable. Dépendamment du niveau d'abstraction utilisé pour décrire la spécification, il est possible d'appliquer efficacement des métriques sur des systèmes matériels complexes. Shimizu et Dill [31] ont proposé une approche afin de dériver un générateur de tests, un module d'analyse de la couverture et un vérificateur de réponses à partir de la spécification formelle des interfaces d'un modèle. Arditi et Clavé [1] utilisent le langage formel Esterel pour décrire une spécification exécutable. En utilisant une machine à états finis produite par le

compilateur du langage Esterel, ils utilisent la couverture des états visités comme métrique dérivée d'une spécification.

Ce chapitre présente une méthode systématique afin d'évaluer quantitativement la couverture fonctionnelle. La méthode est basée sur l'utilisation de métriques fonctionnelles appliquées sur une spécification fonctionnelle exécutable. Le produit final de la méthode est un Module d'Analyse de Couverture (MAC) directement intégrable dans le banc d'essais dédié au modèle sous vérification. En utilisant une spécification exécutable d'un modèle, des métriques prédéfinies peuvent être appliquées sur la spécification d'une manière systématique. On applique une métrique sur la spécification en définissant un patron de vérification dédié à la spécification exécutable. Ainsi, en configurant de façon systématique un patron de vérification prédéfini, puis en se basant sur la spécification exécutable et sur des tâches précises requises d'un ingénieur de vérification, on obtient un MAC dédié au modèle sous vérification. Dans le but de valider la méthode, un patron de vérification a été développé afin d'appliquer la métrique des flots transactionnels [4] sur une spécification exécutable. Le SDL [21] est utilisé pour la description de la spécification exécutable. SDL est un langage graphique et textuel utilisé pour valider les systèmes concurrents et interactifs. Le logiciel Telelogic Tau™ [35] est utilisée pour la capture de la description SDL. Tau™ fournit aussi un engin de simulation permettant de valider le système défini. Les MAC produits avec l'aide de la méthodologie sont implantés avec le langage de vérification *e*. Les fonctionnalités de couverture de Specman Elite™ sont utilisées afin d'échantillonner les résultats de simulation. En utilisant un MAC produit par la méthode, il est possible de définir des suites de tests dont l'efficacité est mesurée sur la spécification exécutable. Ensuite, les suites de tests sont appliquées sur une version RTL du modèle afin de permettre une exploration efficace des fonctionnalités du modèle. De plus, le concept d'abstraction hiérarchique de notre métrique est introduit afin de contrôler les possibilités d'explosions combinatoires de la métrique.

La méthodologie est détaillée dans les prochaines sections de ce chapitre. En premier lieu, la méthodologie permettant de produire un MAC est présentée. Puis, le système de

couverture utilisée par la méthode est présenté. Ensuite, le patron de vérification développé est exposé en détail. Finalement, une discussion sur la méthode est présentée. Les résultats de l'application de la méthode seront présentés au chapitre 5.

4.1 Méthodologie de Couverture

L'utilisation de métriques dans le contexte de la conception d'un banc d'essais dépend de la technique de production des tests utilisée. Dans le cas où une approche déterministe est utilisée, la mesure de la couverture est déterminée en fonction des tests produits. Cependant, dans le cas où on utiliserait une approche pseudo-aléatoire ou automatique pour la production de tests, il est nécessaire d'inclure dans le banc d'essais un module permettant d'effectuer l'analyse de la couverture. Pour une génération pseudo-aléatoire des tests, un module d'analyse est essentiel afin de savoir les fonctionnalités exercées suite à l'application des vecteurs de test. D'autre part, l'utilisation d'un MAC permet de biaiser la génération en fonction du taux de couverture actuel, afin de réduire la redondance induite par une génération pseudo-aléatoire. Pour ce qui est de la production automatique des tests, un générateur de tests est nécessaire et l'analyse de la couverture est implicite, puisque le générateur de tests produira, en fait, des tests déterministes en fonction de la technique de vérification. Cependant, lorsque les approches de production de tests (déterministe, pseudo-aléatoire et automatique) sont utilisées conjointement, un MAC est nécessaire afin de quantifier le taux de couverture obtenu de la combinaison de ces trois approches.

En se situant au niveau de la vérification fonctionnelle du modèle, un MAC doit être basé sur des métriques servant à mesurer les fonctionnalités exercées lors d'une simulation. Le problème avec cet objectif est qu'en voulant travailler à la couverture des fonctionnalités d'un modèle, il faut avoir aussi un format pour représenter les fonctionnalités du modèle. Si on compare avec le test structurel au niveau porte logique, un module d'analyse doit être en mesure d'interpréter une représentation du circuit et les métriques utilisées. Par exemple, en utilisant une métrique comme un modèle de panne les collages simples, on a une représentation du circuit qui est définie avec un ensemble fini de primitives. Dans ce cas, un simulateur de pannes peut être considéré comme une

sorte de MAC. Donc, en ayant une structure normalisée des fonctionnalités d'un modèle et en spécifiant le modèle dans cette structure, il est maintenant possible de développer des MAC dédiés à l'évaluation de la couverture fonctionnelle de métriques précises. Le développement d'un MAC peut donc être effectué de façon systématique en ayant une structure fonctionnelle normalisée d'un modèle et une métrique fonctionnelle appliquée sur cette structure.

La Figure 4-1 présente le flot utilisé pour créer le MAC.

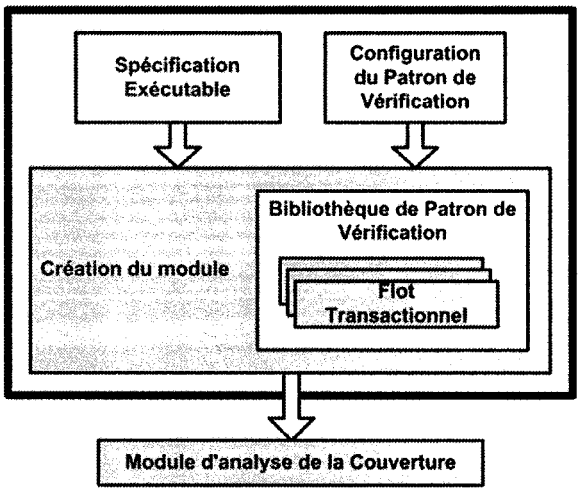


Figure 4-1: Méthode de développement d'un MAC

La Figure 4-1 expose la méthode de façon générale. Ainsi, une bibliothèque de patrons de vérification est disponible pour créer des modules selon différentes métriques. Un patron de vérification est un gabarit servant à réaliser une implantation concrète d'un module dédié à la vérification. On peut aussi définir un patron de vérification comme une solution générique à un problème récurrent. Dans le cas présent, le problème est de savoir comment appliquer une métrique précise sur une spécification exécutable dans le but de créer un MAC. En fait, la définition d'un patron de vérification est la même que celle donnée pour un patron de conception tel qu'énoncé par [16]. L'application de la métrique aux flots transactionnels servira de base pour créer un patron de vérification qui sera présenté dans ce chapitre. La création du module est donc basée sur la spécification exécutable, en l'occurrence un modèle décrit en SDL, et certaines configurations supplémentaires qui sont nécessaires. L'implantation concrète du module est effectuée en

utilisant le langage *e*. La méthode proposée dans ce mémoire n'automatise pas la création du MAC. Cependant, elle définit un ensemble de règles permettant l'élaboration manuelle d'un MAC.

La Figure 4-2 présente le processus de vérification d'un modèle incluant l'utilisation de la méthode présentée dans ce chapitre. Le processus est basé sur trois phases : La définition des requis du circuit, le développement de la vérification et la simulation.

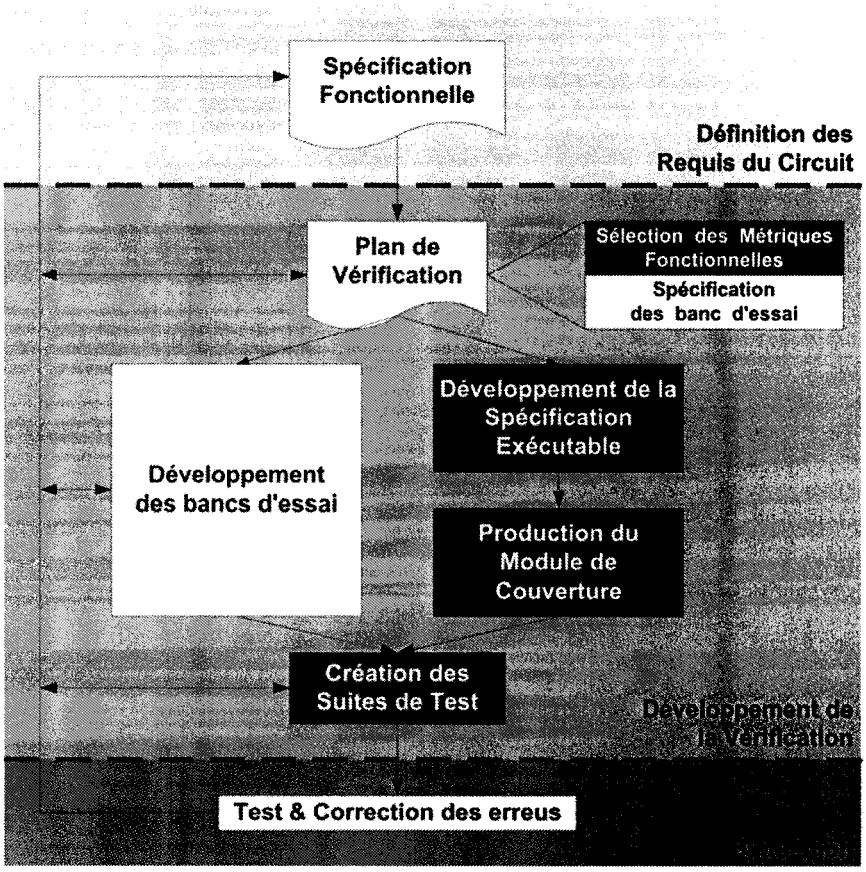


Figure 4-2: Processus de Vérification

Dans la phase de définition des requis, une spécification fonctionnelle est développée dans un format habituellement textuel. Ensuite, basé sur la spécification fonctionnelle, les développements de la vérification et du modèle sont effectués en parallèle. Dans la phase de développement de la vérification, un plan de vérification est défini afin de spécifier, entre autre, les bancs d'essais et les métriques fonctionnelles utilisées pour la vérification. Ensuite, le développement des bancs d'essais et du MAC peuvent être effectués en

parallèle. Le développement du MAC inclut la définition de la spécification exécutable et de la production du module en tant que tel. En utilisant les bancs d'essais et le MAC, la création de suites de tests est possible sans avoir la version RTL du modèle puisque la mesure de la métrique est effectuée sur la spécification exécutable. Finalement dans la phase de simulation, les suites de tests développées sont appliquées sur la version RTL du modèle afin de détecter des erreurs fonctionnelles et ainsi augmenter le degré de confiance dans le modèle.

L'application d'une métrique sur une spécification exécutable apporte plusieurs éléments désirables dans le processus de vérification. En premier lieu, cela permet de valider les requis du modèle. Ainsi, une erreur de spécification sera détectée très tôt dans le processus de développement du circuit, ce qui réduit le coût de correction d'une erreur. Ensuite, le processus de vérification est accéléré puisque les suites de tests sont développées en parallèle avec la conception du modèle.

4.2 Système de couverture

La méthode systématique que nous proposons pour produire un MAC est basée sur un patron de vérification. Dans cette section, une description des éléments nécessaires à la compréhension du patron de vérification est présentée. Le patron de vérification développé implémente une technique de vérification basée sur un graphe : la vérification des flots transactionnels. Ainsi, une revue de cette technique de vérification sera effectuée. Cette technique de vérification est appliquée sur une version standardisée de la spécification d'un circuit. En fait, une spécification exécutable est décrite en utilisant le SDL. Un résumé de la manière utilisée pour décrire un modèle SDL est donc effectué. Ensuite, la métrique dérivant la technique de vérification est formulée afin d'être applicable sur un modèle SDL. Finalement, la formulation de la métrique est complémentée en introduisant une manière d'abstraire la métrique.

4.2.1 Technique de vérification utilisée

Une technique de vérification fonctionnelle émergente est de vérifier le DUV au niveau transactionnel [9]. En fait, la technique définit le niveau d'abstraction auquel la

vérification est appliquée. Au niveau transactionnel, on considère les transactions pouvant être appliquées ou recueillies aux interfaces d'un modèle. Une transaction est une structure de donnée utilisée afin de définir une séquence d'actions sur un modèle. Par exemple, une transaction peut être l'envoi d'un paquet dans une unité de routage ou encore une séquence d'initialisation. Au niveau transactionnel, les détails tel que le protocole utilisé pour introduire un paquet dans le modèle ne sont pas nécessaires. Seulement le fait d'envoyer ou de recevoir un paquet sont considérés comme étant pertinent lorsqu'on travaille à ce niveau d'abstraction. Ensuite, on utilise dans les bancs d'essais des fonctions permettant de transformer les transactions en vecteurs binaires (les BFM). Ces fonctions permettent aussi d'appliquer les vecteurs binaires sur le DUV en accord avec le protocole temporel d'application de la transaction. Toujours en étant à ce niveau d'abstraction, les réponses du DUV sont aussi ramenées au niveau transactionnel par des fonctions qui effectuent l'opération inverse de celles décrites précédemment. La technique de vérification la plus communément utilisée en travaillant au niveau transactionnel est l'analyse partitionnelle. Ainsi, on couvre les types de transactions appliquées et recueillies.

Dans le cadre de notre méthode de couverture, nous travaillons à ce niveau d'abstraction. Tel que mentionné ci-haut, la technique de vérification utilisée est la technique de vérification des flots transactionnels. C'est-à-dire que nous considérons le DUV comme un système de traitement des transactions à haut niveau. Ce système constitue en fait une spécification exécutable du DUV. Dans ce système, nous considérerons le traitement effectué suite à l'application d'une transaction sur la spécification exécutable modélisée. Pour décrire le système de traitement des transactions, on utilise un graphe transactionnel. Le graphe transactionnel permet de définir le cheminement des transactions à l'intérieur du modèle. En fait, un graphe transactionnel combine les éléments d'un graphe de flot de contrôle et d'un graphe de flot de donnée. De plus, il supporte les notions d'état et de concurrence. La Figure 4-3 présente des exemples simples de graphe de flots de contrôle et de flot de donnée.

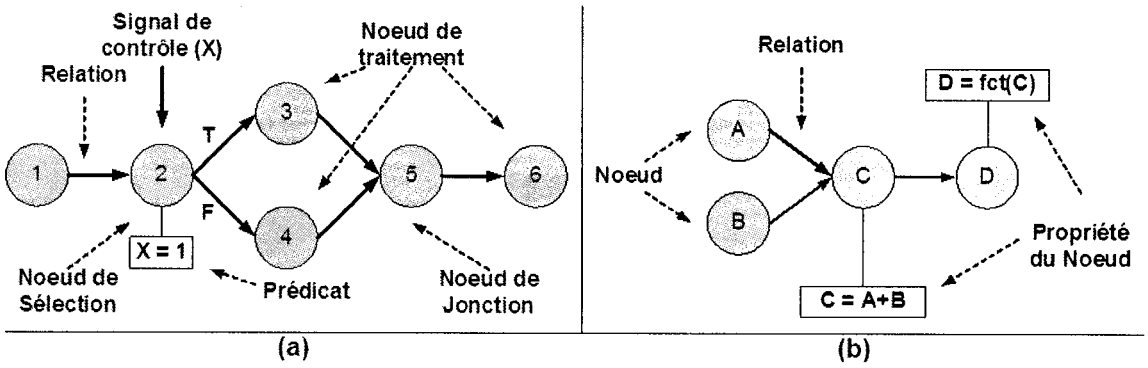


Figure 4-3: Graphe de flot de contrôle et de flot de donnée

À la Figure 4-3 (a), nous avons un graphe de flot de contrôle. Ce type de graphe met l'accent sur le traitement effectué en fonction de signaux de contrôle. Ainsi, les nœuds de traitement (1, 3, 4 et 6) incluent une séquence d'actions qui doivent être exécutées. Les relations, des liens dirigés, signifient qu'un nœud sera exécuté immédiatement à la suite d'un autre nœud. Les nœuds de sélection sont des nœuds ayant des signaux de contrôle en entrée et plusieurs liens sortants. Un attribut d'un nœud de sélection est un prédicat défini en fonction des signaux de contrôle. Le prédicat est utilisé pour déterminer le prochain nœud de traitement. Les nœuds de jonction sont des nœuds ayant plusieurs liens entrants. D'autre part, un graphe de flot de donnée, tel que présenté à la Figure 4-3 (b), met plutôt l'accent sur le cheminement des données. Un nœud représente une donnée. Une relation, un lien dirigé, signifie que la valeur de la donnée d'un nœud est utilisée pour calculer la valeur du nœud pointé par le lien. Par exemple, à la Figure 4-3 (b), A et B sont utilisés pour calculer la valeur de C. De plus, les nœuds possèdent une propriété définissant l'équation permettant de calculer la valeur du nœud. Les exemples précédents représentent des graphes de flot de contrôle et de flot de donnée de base. Il est possible de mélanger les deux types de graphe en ajoutant, par exemple, des nœuds de sélection dans un graphe de flot de donnée. Un graphe transactionnel possède en fait des propriétés provenant de ces deux types de graphe.

La création d'un graphe transactionnel consiste à définir le traitement des transactions dans un système. Nous allons utiliser un exemple simple afin d'expliquer la procédure permettant de définir le système de traitement des transactions. L'exemple consiste en un

système acceptant des paquets sur un port d'entrée. Ensuite, un certain formatage est effectué sur le paquet et ce dernier est envoyé vers deux ports de sortie. La Figure 4-4 présente l'exemple de système de traitement des transactions.

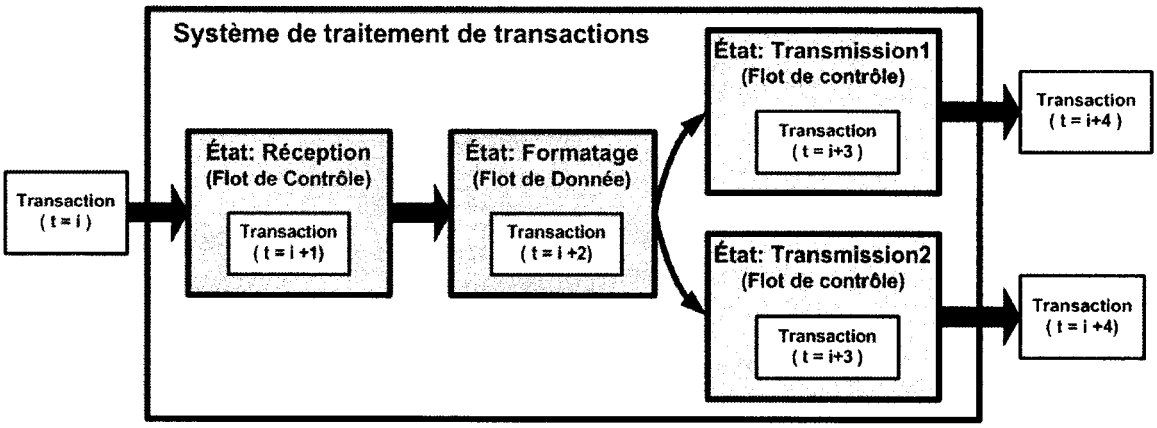


Figure 4-4: Exemple d'un système de traitement de transactions

La première étape de la création du système consiste à définir les types de transactions impliqués dans le modèle. Dans l'exemple présenté, nous allons supposer qu'il y a un seul type de transaction et que cette transaction est un paquet. Ensuite, nous devons déterminer les états de traitement des transactions dans le système. Les états traitement de ce type de transaction sont : *Réception*, *Formatage*, *Transmission1* et *Transmission2*. Dans chaque état de traitement d'une transaction, une unité de traitement est définie. Dans chaque unité de traitement, on définit les opérations effectuées sur la transaction ou produite par la transaction en définissant soit un graphe de flot de contrôle, de flot de donnée ou un hybride entre les deux. Il est aussi possible de définir une machine à états associés à une unité de traitement de la transaction. Par exemple dans l'état de traitement *Réception*, on peut définir un graphe de flot de contrôle qui sera contrôlé par un signal indiquant qu'une transaction est prête à entrer dans le système. Lorsque la transaction passera dans l'état *Réception*, le graphe de flot de contrôle peut attendre ensuite un autre signal de contrôle provenant de l'unité de formatage afin que la transaction passe dans l'état *Formatage*. Le traitement d'une transaction n'est pas seulement le passage en séquence dans les différents états de traitement du système, il peut aussi y avoir, par exemple, un dédoublement d'une transaction dans plusieurs unités

de traitement en parallèle. C'est le cas de la Figure 4-4, où suite au formatage du paquet, ce dernier est envoyé vers deux unités de transmission. La modélisation d'un système de traitement transactionnel ne peut être représentée par une machine à états, puisqu'il peut y avoir plusieurs transactions en traitement dans le système en même temps. Un processeur en pipeline traitant des instructions, les transactions du système, est un bon exemple de système pouvant être représenté sous la forme d'un système de traitement de transactions.

La représentation d'un système sous la forme d'un système de traitement de transactions permet d'observer le cheminement des transactions dans un système. En représentant chaque unité de traitement sous la forme d'un graphe, on peut extraire le chemin exercé par une transaction dans chaque unité de traitement. Le cheminement d'une transaction sera donc l'ensemble des chemins exercés dans le système complet. Le cheminement d'une transaction dans un système est appelé un flot transactionnel. Ainsi, le MAC doit être en mesure d'échantillonner les flots transactionnels exercés lors d'une simulation et de fournir un taux de couverture par rapport au nombre absolu des flots du système modélisé. Les prochaines sections détaillent comment nous décrivons concrètement le système de traitement de transactions et basé sur cette description, une formulation de métrique est effectuée.

4.2.2 Définition de la spécification exécutable

Une notation est nécessaire afin de décrire le système de traitement de transactions représentant les fonctionnalités d'un circuit. Nous appellerons la notation pour cette tâche le *langage de spécification*. Plusieurs langages de spécification peuvent être utilisés pour effectuer cette description. Notre choix s'est arrêté sur la notation SDL. SDL est surtout utilisé dans le domaine des télécommunications. Les autres options qui auraient pu être utilisées sont le UML ou encore SystemC. Cependant, le SDL a l'avantage de permettre une description graphique du système, ce qui n'est pas possible avec SystemC. Une description graphique permet une meilleure compréhension du système modélisé. D'autre part, le SDL possède un équivalent textuel de la description graphique ce qui facilite grandement un traitement sur le système capturé. Ce n'est pas le cas pour UML. Malgré

cela, le SDL a l'inconvénient d'être méconnu dans le monde de la microélectronique et il semble que SystemC ait un avenir plus prometteur. Cependant dans le cadre de ce projet, SDL a été utilisé afin de démontrer la méthode de couverture.

Dans le but de résumer simplement la modélisation avec SDL, nous allons baser notre discussion sur la Figure 4-5. Cependant, Doldi [15] présente des techniques de modélisation SDL plus élaborées.

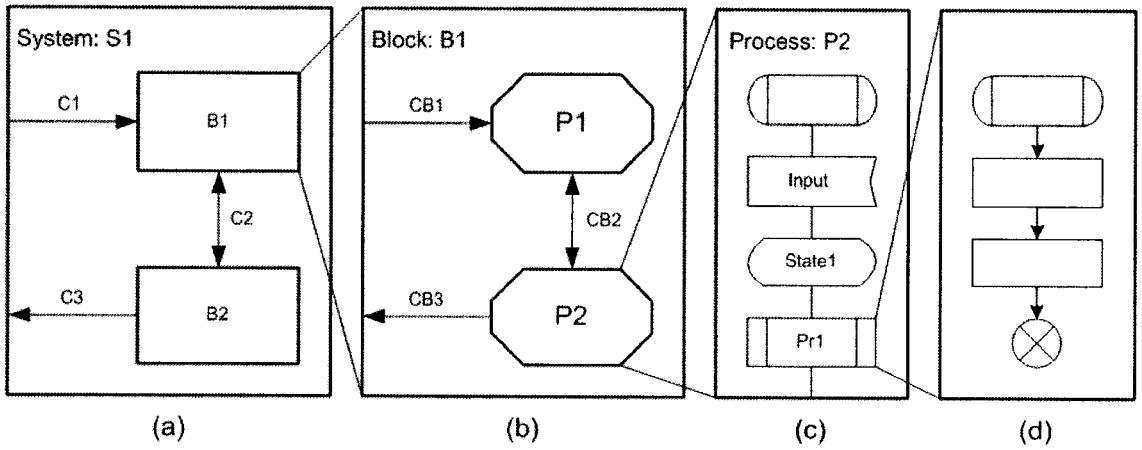


Figure 4-5: Structure d'une modélisation SDL

Une modélisation SDL est réalisée de façon hiérarchique avec un système, des blocs, des processus et des procédures. Au plus haut niveau de la hiérarchie, il y a un système qui représente le modèle. Dans un système, nous partitionnons le modèle en différents blocs et nous définissons des canaux de communication entre les éléments contenus dans le système. À la Figure 4-5 (a), nous avons la représentation d'un système contenant 2 blocs : B1 et B2. Trois canaux de communication (C1, C2 et C3) ont été définis afin d'échanger des signaux entre les éléments contenus dans le système. Il est à remarquer que le canal C2 permet une communication entre les 2 blocs contenus dans le système et que les deux autres canaux permettent une communication entre les blocs et l'environnement extérieur du système. Un bloc, présenté à la Figure 4-5 (b), contient à la base des processus et des canaux de communication. À ce niveau, les canaux servent aux communications inter–processus, ainsi qu'à la communication entre les processus et les éléments de niveaux supérieurs. Le processus, présenté à la Figure 4-5 (c), définit le

comportement du modèle. À l'intérieur des processus, on définit une machine à états étendue permettant d'exécuter les différentes actions en fonction des signaux reçus par les canaux de communication. De plus, c'est au niveau des processus que des signaux sont envoyés vers d'autres éléments du modèle. Finalement, la Figure 4-5 (d) présente une procédure. Les procédures servent à exécuter une série d'actions.

On peut représenter une modélisation SDL par un ensemble de machines à états étendues communicantes (CEFSM ou *Communicating Extended Finite State Machine*) [10]. Afin de considérer un système SDL sous la forme d'un ensemble de CEFSM, il est nécessaire de ramener tous les processus au niveau du système en éliminant toutes les structures de bloc. De plus, il faut exprimer les procédures directement dans les processus. De cette façon, on obtient un ensemble de processus SDL (une CEFSM) communiquant par des canaux de communications. Une CEFSM est donc une machine à états finis étendue avec un ensemble de variables locales. De plus, la CEFSM est synchronisée par des événements paramétrés provenant de canaux de communication. Les événements sont paramétrés, puisqu'ils peuvent transporter des valeurs qui seront affectées aux variables de la CEFSM. Une CEFSM est un 5-tuple : $CEFSM = (ET, et_0, EV, \delta, V)$, où ET est un ensemble d'états, et_0 est l'état initial, EV est un ensemble d'événement entrant ou sortant de la CEFSM avec leurs paramètres, δ est la fonction de transition d'état et V est l'ensemble des variables locales de la CEFSM. Pour définir la fonction de transition d'états, il faut savoir que le SDL possède plusieurs types de transitions. Une transition est à la base déclenchée par un événement entrant dans la CEFSM. Cependant, la transition peut être conditionnelle à un prédicat défini avec les variables inclues dans V. De plus, il y a les transitions continues qui sont seulement basées sur un prédicat sans impliquer d'événement entrant dans la CEFSM. Ainsi, on peut formaliser la fonction de transition d'état par la formule suivante : $\delta(État\_Courant, Événement, Prédicat) \rightarrow (État\_suivant, \{liste\ d'actions\})$. En fonction d'un état courant, d'un événement entrant facultatif et d'un prédicat qui peut toujours être vrai, on obtient l'état suivant et une séquence d'actions effectuée lors de la transition d'état. Les actions permettent de modifier les variables locales contenues dans V et d'émettre des

événements, ce qui constitue les sorties de la CEFSM. Pour illustrer l'équivalence entre cette notation et la notation SDL, l'exemple *Exemple_CSFSM* est présenté au Tableau 4-1.

Tableau 4-1: Un exemple d'une CEFSM

Exemple_CEFSM = (	
L'ensemble d'états	***{E0, E1, E2, E3},***
L'état initial	***E0,***
	{
Les événements avec leurs paramètres	***ev0_entree,***
	ev1_entree(boolean),
	ev3_sortie(integer)
	},
	{
La fonction de transition d'états	***δ(E0,-,-) → (E1, { Ø }))***
	δ(E1, ev0_entree,-) → (E2, {v0 := v0 +1})
	δ(E2, ev1_entree(v1),-) → (E3, { Ø })
	δ(E3, -, v1 = FALSE) → (E1, { v0 := v0 +1})
	δ(E3,- , v1 = TRUE) → (E1, { ev3_sortie(v0), v0 := 0})
	},
	{
L'ensemble de variables	***v0(integer),***
	v1(boolean)
	}
)	

À l'aide de la notation SDL, le processus implémentant la CEFSM du Tableau 4-1 est illustré à la Figure 4-6 (a).

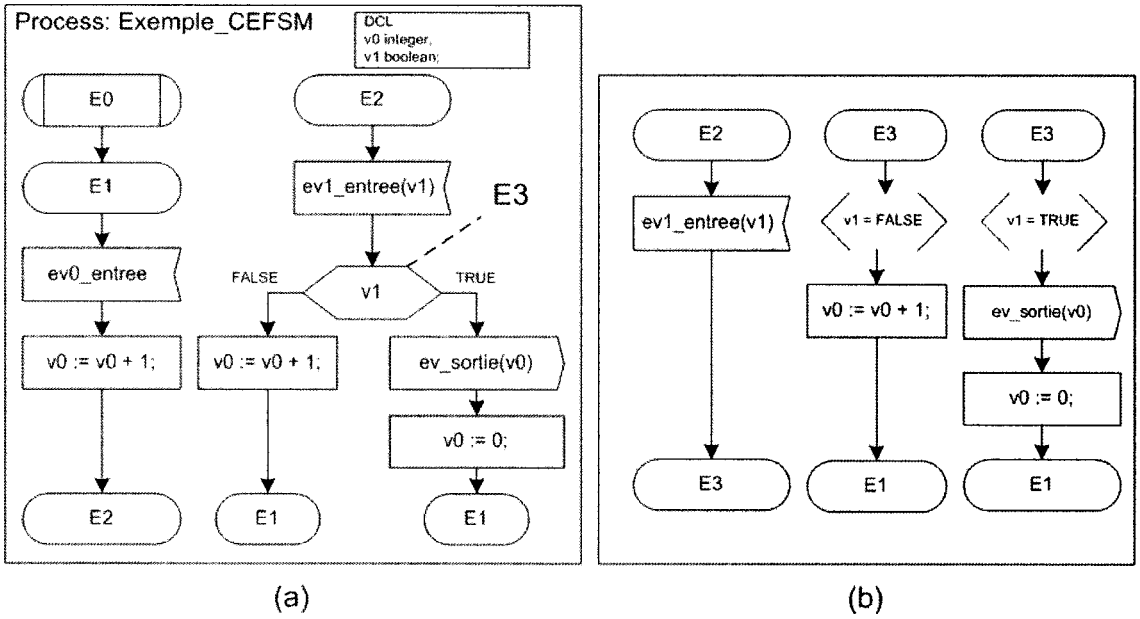


Figure 4-6: Un exemple de CEFSM en version SDL

Dans ce processus, l'état *E3* est implicitement représenté par une décision basée sur la variable *v1*. On peut représenter les définitions du Tableau 4-1 de cette façon, puisque l'équivalent de la transition, incluant l'élément décisionnel, est représenté en SDL à la Figure 4-6 (b). Dans cette dernière, nous avons trois transitions qui représentent fidèlement certaines définitions de la fonction de transition d'état. L'équivalence de la décision est définie avec deux transitions continues qui sont basées sur les prédicats définis dans la fonction de transition d'états.

La description d'un système de traitement de transactions avec SDL est réalisable en considérant chaque unité de traitement du système transactionnel avec un processus SDL. La possibilité de modéliser des décisions ou des transitions continues basées sur un prédicat permet de modéliser les processus dans un style semblable à un graphe de flot de contrôle. Pour ce qui est du cheminement des transactions dans le système, elles sont transmises entre les différentes CEFSM par les canaux de communication. Plus précisément, les transactions sont transférées par les paramètres des signaux s'échangeant entre les CEFSM.

La prochaine section formule la métrique des flots transactionnels en fonction d'une spécification décrite en SDL.

4.2.3 Formulation de la métrique

Le patron de vérification permet de mesurer la couverture des flots transactionnels sur une spécification définie en SDL. Il est donc nécessaire de formuler la métrique en fonction d'une description SDL. Ainsi, cette section décrit la logique développée pour obtenir la mesure. L'explication de la métrique sera faite avec un exemple théorique qui permet de visualiser le concept. Cet exemple n'est en rien un exemple concret, mais il permet d'illustrer de manière simple ce qui est mesuré par la métrique. La métrique se construit en suivant les 5 étapes suivantes :

1. ***Détermination du type de transaction couverte***

Plusieurs types de transaction peuvent être impliqués dans la représentation du modèle sous forme d'un système de traitement de transactions. La métrique que nous utilisons prend en charge un seul type de transaction. Donc, il est nécessaire de définir la transaction qui sera couverte par le MAC. Ainsi, ce sont les instances des transactions appliquées sur le modèle qui exerceront les flots transactionnels définis dans le modèle. D'autre part, afin d'identifier quelles transactions ont été exercées dans les parties du modèle, un jeton unique sera associé à chaque instance des transactions couvertes.

2. ***Détermination des processus impliqués dans le modèle de couverture***

La Figure 4-7 présente le modèle *M* auquel on appliquera la métrique des flots transactionnels.

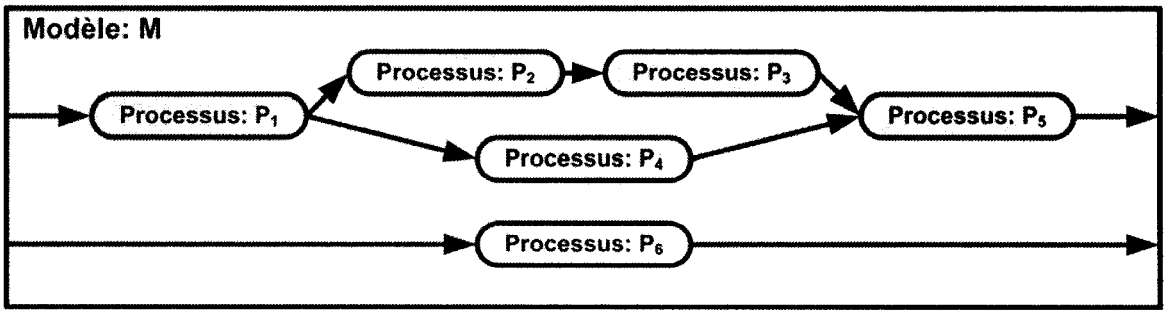


Figure 4-7: Modèle théorique

Un ensemble de processus P (P_1, P_2, P_3, P_4, P_5 et P_6) est utilisé afin de définir le comportement du modèle SDL *M*. Un sous-ensemble PC_i de P est l'ensemble des processus qui définissent le système de traitement de la transaction couverte. L'index i

sert à accéder à un processus de PC_i. L'ensemble PC_i inclut les processus P_1, P_2, P_3, P_4, *et* P_5. Il est à noter que le processus P_6 n'est pas inclus dans PC_i, car il n'est pas pertinent pour la couverture des flots transactionnels du type de transaction d'intérêt.

3. *Énumération des chemins dans tous les processus contenus dans* PC_i

Un flot transactionnel inclut un chemin exercé par la transaction couverte dans tous les processus inclus dans PC_i. Ainsi, il est nécessaire d'énumérer les chemins des processus inclus dans PC_i. Pour chaque processus dans PC_i, nous avons un ensemble de chemins C_j, j étant l'index d'un chemin particulier. Un processus théorique basé sur une représentation d'une CEFSM est présenté à la Figure 4-8.

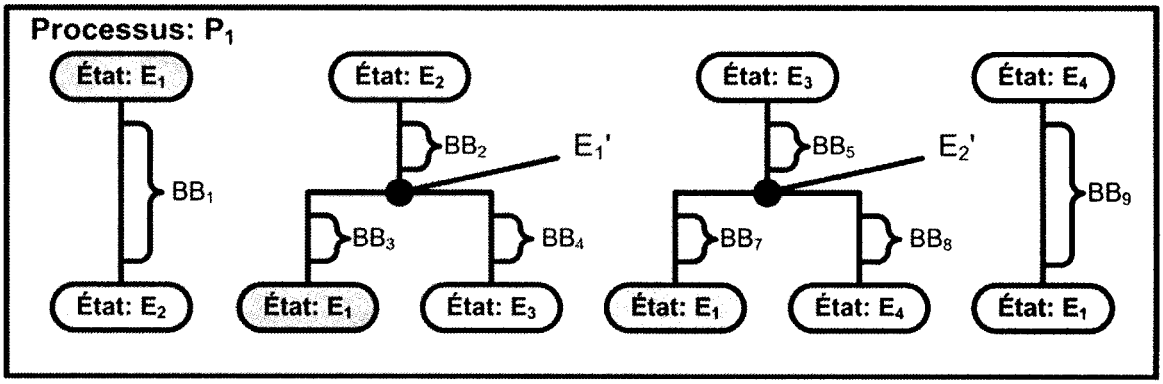


Figure 4-8: Processus théorique

Afin d'énumérer tous les chemins possibles d'un processus, nous allons considérer le processus avec des Branches de Base (BB) et des états. Tel qu'exposé à la section précédente, une transition incluant des éléments décisionnels est une composition de plusieurs transitions définies par la fonction de transition d'états. Ainsi chaque BB, définie à la Figure 4-8, possède une correspondance pour une transition définie dans la fonction de transition d'états. Puisque nous considérons des BB, nous devons aussi considérer des états implicites tel que l'état E_1' et E_2'. Puisqu'une transaction entre dans le processus comme un paramètre d'un signal d'entrée du processus et que la transaction sortira du processus comme un paramètre d'un signal de sortie, il est nécessaire de connaître ces points d'entrée et de sortie afin d'énumérer les chemins du processus. Un chemin est constitué d'une séquence de BB. Par exemple, considérons une transaction qui arrive dans le processus par un signal d'entrée déclenchant l'exécution de BB_1. Une

transaction peut ressortir du processus par une action des BB : BB_3, BB_7 et BB_9. En utilisant ces hypothèses, nous obtenons l'ensemble des éléments qui composent le chemin C_1. Le Tableau 4-2 résume l'ensemble des chemins C_j définis dans le processus PC_1.

Tableau 4-2: Ensemble des chemins du processus PC_1

Chemins (C_j)	Séquence de branches de base
C_1	$BB_1 - BB_2 - BB_3$
C_2	$BB_1 - BB_2$ - $BB_4 - BB_5 - BB_7$
C_3	$BB_1 - BB_2$ - $BB_4 - BB_5 - BB_8$- BB_9

L'exemple exposé est un cas simple. Dans certains cas, il faut aussi considérer des contraintes. Par exemple, s'il y a une possibilité d'itérer, en ayant des boucles, dans l'énumération des chemins, il faut que le nombre d'itérations effectuées par la boucle soit fini. Ainsi, une contrainte doit être spécifiée afin de définir le nombre d'itérations permis. Autrement, il serait parfois impossible de créer une liste finie de chemins pour un processus. Cet attribut est essentiel afin d'obtenir une métrique finie.

4. Détermination des origines et des fins des flots transactionnels

Les flots transactionnels doivent être caractérisés avec une origine et une fin. En fait, cela correspond au point d'entrée et au point de sortie d'une transaction dans le système de traitement de transactions considéré dans son entier. Il est possible qu'il y ait plusieurs points d'entrées et de sorties. Évidemment, dans un système incluant plusieurs états de traitement ou plusieurs processus, les points d'entrée et de sortie sont définis dans des processus différents. En fait, on doit spécifier les BB spécifiques dans le système qui seront considérées comme les origines et les fins possibles des flots transactionnels.

En utilisant cette définition, nous sommes en mesure de formaliser le format d'un flot transactionnel. Un ensemble de flots transactionnels légaux F_k définit ce qu'on doit couvrir. L'index k permet d'accéder à un flot transactionnel particulier. Pour tous les flots transactionnels spécifiques de FT_k, nous avons un ensemble de nom de chemin FP_i associé à chaque processus inclus dans PC_i. Ainsi, un flot transactionnel est un ensemble de chemins couverts par une transaction dans les processus PC_i du modèle M. Les chemins couverts par une transaction sont appariés en utilisant le jeton unique associé à chaque instance d'une transaction. D'autre part, un chemin est une séquence de BB.

Donc, on peut généraliser qu'un flot transactionnel est un ensemble de BB exercés par une transaction de façon séquentielle ou concurrentielle. L'aspect séquentiel vient des séquences dans les chemins couverts et aussi des séquences de processus exercées. Tandis que l'aspect concurrentiel vient de la stimulation de plusieurs processus en parallèle. Toutefois, un flot transactionnel n'implique pas obligatoirement qu'un chemin ait été exercé dans chaque processus inclus dans PC_i. Donc, pour le cas où un processus ne serait pas impliqué dans un flot transactionnel, le nom de chemin spécial *VIDE* est défini. Le nom de chemin spécial *VIDE* est affecté au nom de chemin dans FP_i associé au processus qui n'a pas été exercé. Le Tableau 4-3 présente quelques exemples de flots transactionnels.

Tableau 4-3: Exemples de flots transactionnels

Flot Transactionnel (F_k)	FP_1	FP_2	FP_3	FP_4	FP_5
F_1	C_1	C_8	C_1	C_4	C_2
F_{23}	C_2	*VIDE*	*VIDE*	C_4	C_2
F_{35}	C_1	C_4	C_2	*VIDE*	C_1

Les lignes du tableau représentent un flot transactionnel spécifique de F_k. Les colonnes représentent les noms de chemins inclus dans FP_i. Il faut aussi comprendre que les chemins ayant les mêmes indices dans les différentes colonnes ne représentent pas les mêmes chemins. Par exemple, pour le flot transactionnel F_{23}, le chemin C_1 de FP_1 est un chemin du processus PC_1 et le chemin C_1 de FP_5 de ce même flot transactionnel, est un chemin du processus PC_5.

5. Détermination des contraintes des flots transactionnels

En suivant les quatre étapes précédentes, nous sommes en mesure de déterminer les flots transactionnels exercés dans le modèle *M*. Cependant, le produit cartésien des noms de chemins de chaque processus inclus dans PC_i ne forme pas l'ensemble légal des flots transactionnels F_k. Il y a des combinaisons de chemin inaccessibles. Par conséquent, pour obtenir F_k il est nécessaire de définir des contraintes. Sans la définition de contraintes, nous ne pouvons pas obtenir une mesure exacte sur la couverture des flots transactionnels puisque la référence d'une couverture complète est incalculable.

Les contraintes nécessaires pour la création de l'ensemble F_k dérivent de diverses sources. Premièrement, il y a les contraintes qui dérivent de la structure du modèle. Par exemple, un nom de chemin d'un processus qui est *VIDE*. Dépendamment de la structure du modèle, cela peut impliquer qu'un autre processus a aussi obligatoirement le nom de chemin *VIDE* ou qu'il ne peut avoir le nom de chemin *VIDE*. D'autre part, il y a des contraintes qui dérivent directement des requis fonctionnels provenant de la spécification. Un exemple correspond au cas des bornes d'une boucle dans un processus.

Un algorithme devant calculer l'ensemble F_k peut être développé et doit considérer tous les éléments dans cette section. De plus, un ensemble de contraintes *CTS* définies dans un format non ambigu est aussi nécessaire afin d'effectuer ce calcul. Cet algorithme n'a pas été développé, puisque ce calcul est effectué par Specman Elite™ avec l'aide de son module de couverture fonctionnelle. Cependant les contraintes doivent être fournies dans un format précis à Specman Elite™ dans le but d'effectuer ce calcul. Le Tableau 4-4 expose quelques exemples de contraintes définis pour obtenir l'ensemble F_k.

Tableau 4-4: Exemples de contraintes définies pour la métrique

Contraintes	Description
Bornes des boucles	Dans l'énumération des chemins d'un processus, il est possible qu'un même BB soit inclut dans le chemin. Cela signifie qu'il y a une boucle. Donc, dans ce cas, il faut déterminer le nombre minimum et maximum d'itérations possibles de la boucle.
Implication Chemin - Chemin	Le passage dans un chemin d'un processus force le passage dans un chemin précis d'un autre processus.
Implication État - Chemin	Le passage dans un certain état d'un processus force le passage dans un chemin précis d'un autre processus.

Les détails sur le format des contraintes et sur le passage des contraintes à Specman Elite™ seront détaillés à la section 4.3.

4.2.4 Abstraction de la métrique

Il est évident que l'utilisation de cette métrique fonctionnelle sur un modèle d'une grande complexité peut mener à un ensemble légal de flots transactionnels très grand. Dans ce cas, une simulation pseudo-aléatoire ne pourra pas atteindre un taux de couverture satisfaisant dans un temps raisonnable. D'autre part, le fait de tenter d'exercer

l'univers complet des possibilités légales du modèle ne permet pas de cibler certaines fonctionnalités représentées par un sous-ensemble de F_k.

La solution proposée est d'appliquer le principe *diviser-pour-régner* sur notre formulation de la métrique des flots transactionnels. Concrètement, nous effectuons une hiérarchisation du système de traitement de transactions en permettant l'abstraction de chaque élément de la hiérarchie. L'abstraction de la métrique représente en fait l'abstraction de certains états de traitement d'une transaction. Dans le but d'illustrer le concept, nous utilisons toujours le modèle théorique présenté à la Figure 4-7. La hiérarchisation de la métrique est effectuée en utilisant trois éléments hiérarchiques : *l'élément terminal*, *l'élément séquentiel* et *l'élément concurrentiel*. Ensuite, tous les éléments de la hiérarchie peuvent être abstraits afin de réduire l'ensemble T_k.

Un élément terminal représente une structure qu'on retrouve directement dans le modèle. Ainsi, les processus inclus dans PC_i sont des éléments terminaux. Un élément séquentiel regroupe d'autres éléments hiérarchiques en séquence dans le modèle. Il est évident que tous les processus s'exécutent en parallèle, mais dans ce cas, le terme séquence est utilisé en fonction du passage d'une transaction dans un processus. La Figure 4-9 illustre le modèle théorique M avec un élément séquentiel ES_1 incluant les processus P_2 et P_3. En fait, les processus P_2 et P_3 sont considérés comme des éléments terminaux.

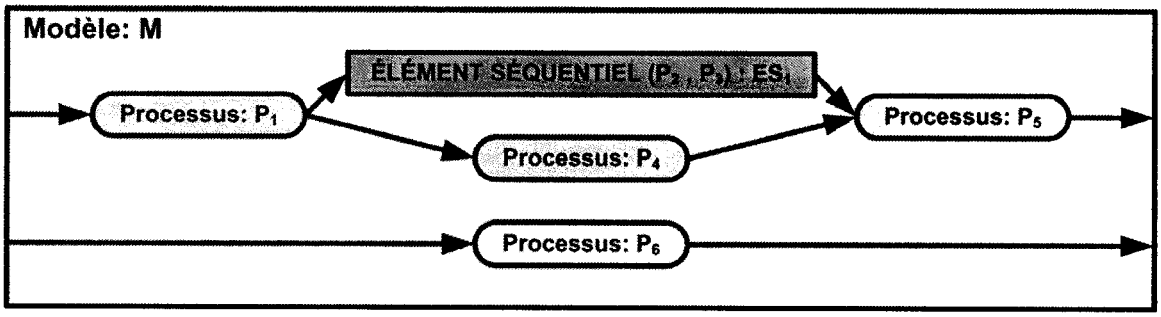


Figure 4-9: Éléments hiérarchiques séquentiels

Un élément concurrentiel regroupe d'autres éléments hiérarchiques en concurrence dans le modèle. La Figure 4-10 illustre le modèle théorique M avec un élément concurrentiel EC_1 incluant l'élément séquentiel ES_1 et le processus P_4.

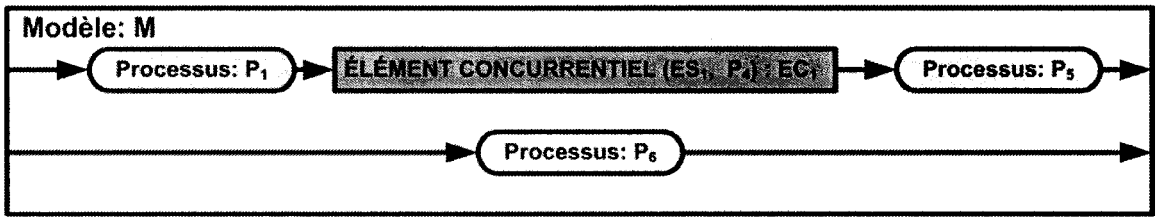


Figure 4-10: Éléments hiérarchiques concurrentiels

En utilisant les éléments hiérarchiques, nous sommes en mesure de définir une hiérarchisation complète du modèle. Puisque tous les éléments hiérarchiques peuvent être abstraits, un ensemble d'interrupteurs d'abstraction *H_ABS* est défini pour un modèle *M*. Chaque élément hiérarchique est associé à un interrupteur d'abstraction inclus dans *H_ABS*. Un interrupteur d'abstraction est un élément binaire qui peut être *OUVERT* ou *FERMÉ*. La valeur *OUVERT* signifie que l'élément hiérarchique sera inclus dans le calcul de la métrique tandis que la valeur *FERMÉ* signifie que l'élément hiérarchique sera abstrait et donc pas inclus dans le calcul de le métrique. L'abstraction d'un élément terminal correspond à l'abstraction du nom du chemin couvert dans un processus. Ainsi, lorsqu'on abstrait un élément terminal, la valeur du nom du chemin du processus peut être *VIDE* ou *ABSTRAIT*. La valeur spéciale *ABSTRAIT* signifie qu'un chemin a été couvert dans le processus en question, mais le nom du chemin spécifiquement couvert n'est pas détaillé. Pour ce qui est des éléments séquentiels ou concurrentiels, une indication est fournie sur sa couverture avec les valeurs *VIDE* et *ABSTRAIT*. La valeur *VIDE* signifie qu'aucun processus inclus dans l'élément hiérarchique n'a été exercé dans le flot transactionnel. La valeur *ABSTRAIT* implique qu'au moins un processus inclus dans l'élément hiérarchique a été exercé dans le flot transactionnel. Le Tableau 4-5 expose quelques exemples de flots transactionnels incluant l'abstraction de différents éléments hiérarchiques.

Tableau 4-5: Flots transactionnels avec abstraction

Abstraction d'éléments terminaux: P_2 et P_4 sont abstraits					
Flot Transactionnel (F_k)	FP_1	FP_2	FP_3	FP_4	FP_5
F_1	C_1	*ABSTRAIT*	C_1	*ABSTRAIT*	C_2
F_{23}	C_2	*VIDE*	*VIDE*	*ABSTRAIT*	C_2
F_{35}	C_1	*ABSTRAIT*	C_2	*VIDE*	C_3

Abstraction d'un élément séquentiel: ES_1 est abstrait				
Flot Transactionnel (F_k)	***FP1***	***FP2***	***FP3***	***FP4***
F_1	C_1	*ABSTRAIT*	C_4	C_2
F_{23}	C_2	*VIDE*	C_4	C_2
F_{35}	C_1	*ABSTRAIT*	*VIDE*	C_3
Abstraction d'un élément concurrentiel: EC_1 est abstrait				
Flot Transactionnel (F_k)	***FP1***	***FP2***	***FP3***	
F_1	C_1	*ABSTRAIT*	C_2	
F_{23}	C_2	*ABSTRAIT*	C_2	
F_{35}	C_1	*ABSTRAIT*	C_3	

On remarque dans le Tableau 4-5 qu'il y a une modification à la définition d'un flot transactionnel. Premièrement dans le cas où il n'y aurait pas d'abstraction, la hiérarchie définie n'est pas considérée. D'autre part, lorsqu'un élément hiérarchique est abstrait, cela implique que tous les autres éléments qui incluent l'élément abstrait ne seront pas considérés. Ainsi, on peut réviser la définition d'un flot transactionnel inclus dans F_k en ajoutant le concept d'abstraction. Sans abstraction, un flot transactionnel inclut dans F_k est un ensemble de noms de chemin FP_i associé à chaque processus de PC_i. Cependant lorsqu'un élément hiérarchique est abstrait, à la place d'avoir un nom de chemin associé pour l'élément, ce sont les valeurs spéciales *VIDE* ou *ABSTRAIT* qui y seront associées.

4.3 Patron de vérification

Le patron de vérification est le gabarit générique utilisé pour créer le MAC des flots transactionnels. Le patron doit être configuré pour chaque utilisation en fonction de la spécification exécutable et des paramètres définis à la section précédente. L'implémentation est faite en utilisant le langage de vérification *e*.

Tel qu'exposé à la section précédente, il est possible d'abstraire la métrique en nous basant sur une hiérarchisation du modèle. Il faut donc que le MAC soit en mesure de supporter l'abstraction. Un utilisateur du MAC pourra fournir la liste des interrupteurs d'abstraction au module afin de le configurer. Un MAC doit donc être en mesure de se configurer automatiquement en fonction de l'abstraction choisie (une liste d'interrupteurs).

Dans cette section, le fonctionnement du système implémenté par le patron sera exposé en tenant compte des éléments nécessaires afin de supporter n'importe quelle abstraction choisie par l'utilisateur du module. Cependant, certaines parties du MAC ne peuvent pas être implémentée en ne sachant pas l'abstraction choisie. Cela signifie que si le développement du MAC se butait à ce problème, nous produirions un MAC incomplet qui devrait être complété, en ajoutant du code, en fonction de l'abstraction choisie. Cependant, ce problème est résolu en proposant la création de méthodes spéciales, en langage *e*, qui généreront automatiquement le code *e* nécessaire pour compléter le MAC. Donc, dans ce qui suit, la reconfiguration automatique du MAC sera aussi détaillée.

4.3.1 Fonctionnement du module

Le MAC est composé de plusieurs éléments. Chaque élément est exploité afin d'obtenir le taux de couverture des flots transactionnels exercés suite à l'application de transactions sur la spécification exécutable. La Figure 4-11 présente une vue statique d'un module d'analyse de la couverture. Les résultats produits par une implantation du MAC sont le taux de couverture des flots transactionnels et aussi le taux de couverture des chemins couverts par chaque processus du modèle. Ces résultats sont générés en utilisant les fonctionnalités de couverture de Specman Elite™ décrite dans l'annexe A.

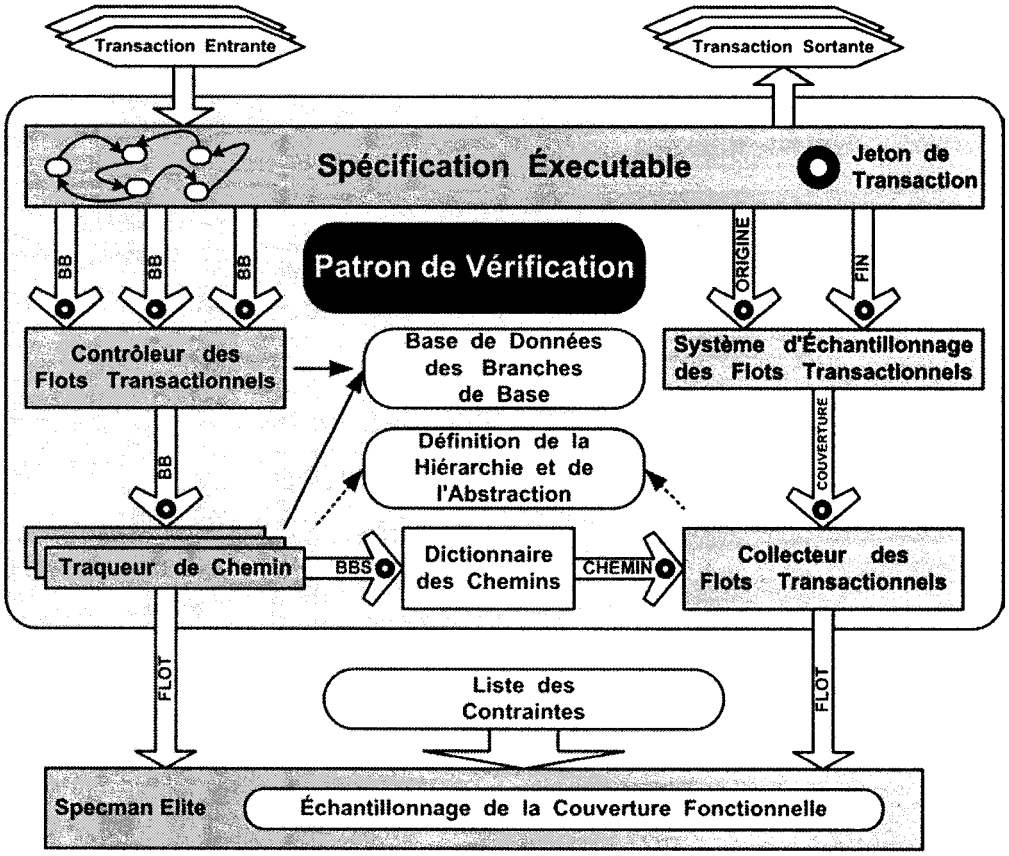


Figure 4-11: Patron de Vérification

Nous allons commencer par décrire la fonctionnalité de tous les éléments de la Figure 4-11. De plus, les interactions entre les différents éléments sont détaillées. Dans ces descriptions, le mot concepteur sera utilisé afin de représenter une personne qui implanterait un MAC avec ce patron de vérification. Il n'est pas possible d'exposer les sources en langage *e* de ces éléments dû aux limitations d'espace de ce document. Cependant, l'annexe D inclut les sources d'un exemple de module créé à partir du patron de vérification. Cela dit, la description de chaque élément du patron restera au niveau conceptuel.

Base de Donnée des Branches de Base

Le dictionnaire des BB contient des caractéristiques pertinentes sur toutes les BB utilisés dans le modèle. Rappelons qu'une BB représente une transition de la fonction de transition d'état d'une CEFSM. Ainsi, les informations pour chaque BB sont :

- Le nom du processus auquel la BB est associée;
- Son rôle dans les chemins du processus. (Début de chemin, Partie interne d'un chemin, Fin de Chemin);
- Un nom unique formé du nom du processus, du nom de l'état de départ et de fin de la BB.

Le concepteur doit fournir ces informations dans un format prédéfini en créant une table associative. La valeur utilisée pour accéder aux informations d'une BB est le nom unique de la BB. La base de données des BB est utilisée pour connaître le nom du processus auquel est associé la BB par le contrôleur des flots transactionnels. D'autre part, la base de donnée des BB est utilisée par les traqueurs de chemin pour connaître le rôle de la BB dans un chemin d'un processus.

Dictionnaire des Chemins

Le dictionnaire des chemins sert à transformer une liste de BB en un nom de chemin. Ainsi, le concepteur doit définir un nom de chemin pour chaque séquence de BB possibles. Le dictionnaire des chemins doit contenir des noms de chemins uniques qui sont composés du nom du processus où est défini le chemin et d'un numéro unique. Le dictionnaire des chemins sert d'intermédiaire entre les traqueurs de chemins et le collecteur de flots transactionnels.

Définition de la Hiérarchie et de l'Abstraction

Dans le module, le concepteur doit définir la hiérarchie du modèle en utilisant un format prédéfini. C'est-à-dire des éléments terminaux, séquentiels et concurrents. C'est aussi dans ce module qu'on retrouve les interrupteurs d'abstraction qui indiquent quels sont les éléments hiérarchiques qu'on veut abstraire. La configuration des interrupteurs

peut être modifiée suite à la création du module par son utilisateur. Ces informations sont utilisées par les traqueurs de chemins et le collecteur des flots transactionnels.

Jeton de Transaction

Le jeton de transaction est utilisé afin de déterminer les parties de la spécification exécutable exercées par une transaction. Ainsi, lorsqu'une transaction est appliquée sur le modèle, un jeton unique (en fait, un numéro unique) est associé à la transaction. Ce jeton sera utilisé pour identifier les BB exercées par une transaction. De plus, le jeton sert à associer les chemins exercés par une transaction.

Spécification Exécutable

La spécification exécutable est en fait une version équivalente en langage *e* du modèle SDL utilisée pour définir la métrique. La spécification exécutable prend en entrée les transactions appliquées sur le modèle, traite les transactions et produit des transactions en sortie. Un utilitaire a été développé afin de transformer une description SDL textuelle en modèle *e*. Cependant, le concepteur doit ajouter certains éléments afin d'échantillonner les flots transactionnels. Ces éléments ne modifient pas la fonctionnalité du modèle. Ces modifications sont :

1. *Création du jeton unique.*

Lorsqu'une transaction entre dans le système, un jeton est créé. Le jeton suivra la transaction à travers tous les processus du modèle. En fait, le jeton est seulement un entier incrémenté à chaque application d'une transaction.

2. *Ajout du jeton unique dans les canaux de communications.*

Les transactions sont transférées vers d'autres processus par l'entremise des canaux de communication reliant les processus. Puisque le jeton unique doit suivre la transaction, il est nécessaire d'augmenter les canaux du modèle en ajoutant le jeton aux canaux de communication. De cette façon, les jetons suivent les transactions lorsque celles-ci seront transmises vers les autres processus du modèle.

3. *Ajout d'émetteurs afin de connaître les BB exercées.*

Dans les processus du modèle, un chemin sera exercé suite à l'arrivée d'une transaction dans le processus. Afin de déterminer le chemin exercé, des émetteurs sont placés dans toutes les BB du processus. Ces émetteurs servent en fait à envoyer un événement au contrôleur des flots transactionnels avec le nom unique de la BB exercée et le jeton unique de la transaction.

4. *Ajout d'émetteur pour les origines et les fins des flots transactionnels.*

Tel qu'énoncé à la section précédente, il est nécessaire de connaître les origines et les fins possibles des flots transactionnels. Donc, aux points d'entrée d'une transaction dans le modèle, des émetteurs doivent être ajoutés afin d'envoyer, au système d'échantillonnage des flots transactionnels, un événement d'origine du flot transactionnel avec le jeton unique de la transaction. D'autre part, aux points de sortie possibles d'une transaction dans le modèle, des émetteurs doivent aussi être ajoutés afin d'envoyer, au système d'échantillonnage des flots transactionnels, un événement de fin de flot transactionnel avec le jeton unique de la transaction.

Contrôleur des Flots Transactionnels

Le contrôleur des flots transactionnels reçoit des noms de BB avec un jeton unique en provenance de la spécification exécutable. Lorsqu'un événement d'une BB arrive, le contrôleur connaît le nom unique de la BB et le jeton associé. En utilisant le nom unique de la BB, il consulte la base de donnée des BB afin de connaître le nom du processus d'où provient la BB exercée. Ensuite, il envoie le nom de la BB et le jeton au traqueur de chemin associé au processus.

Traqueur de Chemin

Le concepteur doit définir un traqueur de chemin pour chaque processus où une transaction peut être traitée. Un traqueur de chemin sert à accumuler des noms de BB afin

de composer un chemin. Les traqueurs de chemins reçoivent, selon l'arbitrage du contrôleur des flots transactionnels, les noms des BB exercées, avec le jeton, en provenance du processus auquel le traqueur est associé. Cependant, le fonctionnement d'un traqueur de chemins dépend de l'abstraction de la métrique. On considère deux modes : les modes normal et abstrait.

Dans le mode normal, le processus n'est pas abstrait, un traqueur de chemin a initialement une liste de nom de BB vide. Suite à l'arrivée d'un nom de BB, il consulte la base de donnée des BB afin d'assurer que la BB reçue correspond bien à un début de chemin. Ensuite, il accumule les noms de BB dans une liste jusqu'à l'arrivée d'une BB caractérisée comme étant une fin de chemin. Il est à noter que durant l'accumulation des BB, le jeton unique doit toujours être identique. Ensuite, ayant en sa possession une liste de BB correspondant à un chemin exercé par une transaction, le traqueur de chemins envoie cette liste au dictionnaire des chemins avec le jeton unique. Le dictionnaire des chemins transformera cette liste en un nom de chemin et il passera ce nom, avec le jeton, au collecteur de flots transactionnels.

Un traqueur de chemins fonctionne dans le mode abstrait, si le processus, auquel le traqueur est associé, est abstrait ou ce dernier fait partie d'un élément hiérarchique abstrait. Dans ce cas, le traqueur fonctionne de façon simplifiée. Plus précisément, lorsqu'il reçoit une BB, il vérifie avec la base de donnée des BB si la BB correspond à un début de chemin. Dans le cas où ce ne serait pas un début de chemin, il ignore la BB. Autrement, si c'est un début de chemin, il envoie la valeur spéciale *ABSTRAIT* au dictionnaire des chemins avec le jeton unique. Le dictionnaire des chemins passera directement cette valeur spéciale au collecteur des flots transactionnels.

D'autre part, les informations cumulées par les traqueurs de chemins, les chemins couverts, sont envoyées au système d'échantillonnage de la couverture de Specman Elite™. De cette façon, on obtient un premier résultat de couverture : la couverture des chemins pour tous les processus du modèle. Il est à noter que lorsqu'un processus est abstrait, la seule valeur possible pour la couverture d'un chemin est *ABSTRAIT*.

Système d'échantillonnage des Flots Transactionnels

Le système d'échantillonnage des flots transactionnels sert à avertir qu'une transaction qui était en activité dans le modèle est ressortie du modèle. Le système reçoit des événements de début et de fin de flot transactionnel en provenance de la spécification exécutable. De plus, ces événements sont accompagnés du jeton unique de la transaction ayant provoquée ces événements. Lorsque le système reçoit un événement de début, il attend un événement de fin de flot ayant un jeton identique. Lorsque cet appariement est effectué, le système envoie un événement de couverture, avec le jeton unique, au collecteur des flots transactionnels. Il est à noter que le système doit tenir compte qu'il peut y avoir plusieurs transactions en traitement dans le système. De plus, il n'est pas assuré que le principe de la première entrée, première sortie sera respecté. Ainsi, le système doit tenir compte de tous ces facteurs.

Collecteur des Flots Transactionnels

Le collecteur des flots transactionnels analyse la couverture des flots transactionnels. Dans ce module, il y a des listes associatives qui sont définies pour chaque traqueur de chemin dans le système. Ce module gère aussi une liste associative définie pour chaque processus contenu dans le modèle. Lorsque le collecteur reçoit un nom de chemin avec un jeton unique de la part d'un traqueur de chemin, il les place dans la liste associative associée au traqueur. Le nom de chemin peut être la valeur *ABSTRAIT* dans le cas où il y aurait abstraction du processus associé au traqueur de chemin. La valeur associative est la valeur du jeton unique et la valeur associée est le nom du chemin. Lorsque le collecteur reçoit un événement de couverture avec un jeton de la part du système d'échantillonnage des flots transactionnels, cela indique qu'il faut construire le flot transactionnel avec tous les bouts de chemins exercés dans le modèle. Ainsi, le collecteur parcourt toutes les listes associatives en tentant de trouver des chemins avec le jeton reçu du système d'échantillonnage. Le module qui gère les listes associatives va retourner la valeur du chemin exercé ou la valeur *VIDE* dans le cas où le processus n'aurait pas été inclus dans le flot transactionnel courant.

Ensuite le collecteur doit consulter la définition de la hiérarchie afin de savoir s'il y a une abstraction de la métrique. Dans le cas où il n'y a pas d'abstraction, le flot transactionnel sera le croisement des valeurs des chemins collectées. Dans le cas, où il y a abstraction, le collecteur doit considérer la hiérarchie définie. Pour un élément terminal abstrait, il n'y a qu'une liste associative associée à l'élément terminal. Donc pour un élément terminal, la valeur, apparié avec un jeton, est soit *ABSTRAIT* ou *VIDE*. Cependant, pour un élément séquentiel ou concurrentiel, il a toujours au moins 2 listes associatives associées à l'élément hiérarchique. Ainsi, le collecteur vérifie pour toutes les listes de l'élément hiérarchique s'il y a au moins une liste contenant la valeur *ABSTRAIT*. Si la valeur *ABSTRAIT* est retournée par une liste, alors la valeur d'élément hiérarchique est *ABSTRAIT*. Autrement, la valeur de l'élément hiérarchique sera VIDE.

Les valeurs échantillonnées des flots transactionnels sont recueillies avec l'aide des fonctionnalités de couverture de Specman Elite™ afin d'obtenir le taux de couverture.

Specman Elite™ et les Contraintes

Le MAC produit deux résultats. En premier lieu, il produit une mesure sur la couverture des chemins dans tous les processus du modèle. Deuxièmement, il produit une autre mesure sur la couverture des flots transactionnels du modèle. Dans la description des différents modules du système de couverture fonctionnelle, nous avons discuté comment les informations que nous voulons couvrir sont capturées et aussi comment nous faisons pour les échantillonner. Cependant pour produire des résultats concrets avec Specman Elite™, nous utilisons les fonctionnalités de couverture du langage *e* décrit dans l'annexe A. Cela implique l'utilisation des groupes de couverture du langage *e*. La création de groupes de couverture permet de créer automatiquement des rapports de simulation, par Specman Elite™, en fonction des éléments définis dans un ou des groupes de couverture. D'autre part, la création des groupes de couverture est aussi dépendante de la hiérarchie du modèle et de l'abstraction.

Premièrement, pour la couverture des chemins dans chaque processus du modèle, nous produisons un groupe de couverture pour chaque processus qui n'est pas abstrait. Il

est inutile de créer un groupe de couverture pour un processus dont la seule valeur possible est *ABSTRAIT*. Nous utiliserons le processus théorique présenté à la Figure 4-8 afin d'exposer comment les groupes de couverture sont produits. Les chemins pouvant être exercés dans ce processus sont définis au Tableau 4-2. Le groupe de couverture, en langage *e*, devant être produit pour couvrir les chemins du processus P_1 est exposé à la Figure 4-12.

```
1: cover path_coverage is {
2:    item B1 using illegal = not(B1 in [BB1]);
3:    item B2 using illegal = not(B2 in [BB2]);
4:    item B3 using illegal = not(B3 in [BB3, BB4]);
5:    item B4 using illegal = not(B4 in [BB5, EMPTY]);
6:    item B5 using illegal = not(B5 in [BB7, BB8,EMPTY]);
7:    item B6 using illegal = not(B6 in [BB9, EMPTY]);
8:
9:    cross B1, B2, B3, B4, B6, B5 using illegal =
10:      not(
11:           (B1 in [BB1])                          and
12:           (B1 == BB1    => B2 in [BB2])          and
13:           (B2 == BB2    => B3 in [BB3, BB4]) and
14:           (B3 == BB3    => B4 in [EMPTY])     and
15:           (B3 == BB4    => B4 in [BB5])          and
16:           (B4 == BB5    => B5 in [BB7, BB8]) and
17:           (B4 == EMPTY => B5 in [EMPTY])     and
18:           (B5 == BB7    => B6 in [EMPTY])     and
19:           (B5 == BB8    => B6 in [BB9])          and
20:           (B5 == EMPTY => B6 in [EMPTY])     and
21:          );
22: };
```

Figure 4-12: Exemple d'un groupe de couverture pour la couverture de chemin

Il faut savoir que le groupe de couverture de la Figure 4-12 n'est pas complet, mais les informations omises n'apportent rien à la compréhension du groupe. Le groupe de couverture est échantillonné pour l'émission de l'événement *path_coverage* par le traqueur de chemin associé à chaque processus. Nous avons défini 6 *items* dans ce groupe de couverture. Ces *items* sont associés à la liste de nom de BB, maintenue dans le traqueur de chemin associé à un processus, qui représente en fait un chemin couvert dans un processus. Nous définissons des nombres d'*items* égaux aux chemins les plus longs dans le processus. Cela est nécessaire puisque la définition d'un groupe de couverture est statique et elle ne peut être modifiée. Dans le cas où la longueur d'un chemin n'atteint pas la longueur du chemin le plus long du processus, la liste des noms de BB du chemin sera complétée avec des valeurs *VIDE* (*EMPTY*) pour obtenir une liste de longueur égale au chemin le plus long. Ensuite, nous avons défini un *cross* qui effectue un croisement

entre tous les *items* préalablement définis. Cet élément du groupe de couverture représente la mesure de couverture des chemins dans le processus. Si on laissait le groupe de couverture seulement avec ce qu'on vient de décrire, nous pourrions savoir quel chemin a été couvert, mais par rapport à l'ensemble complet des croisements possibles. Cependant, nous savons qu'avec cet exemple simple, il n'y a que trois chemins possibles. Il faut donc ajouter des contraintes au groupe de couverture afin d'obtenir la liste des chemins légaux du processus. Le résultat que nous avons pour le groupe de couverture de la Figure 4-12 est que pour chaque item, nous définissons les valeurs légales pouvant être prises pour la première BB du chemin, pour la deuxième et ainsi de suite. Pour ce qui est des contraintes pour le *cross*, une série d'implications doivent être définies. Par exemple, si la deuxième BB du chemin est *BB2*, cela implique que la troisième BB du chemin est *BB3* ou *BB4*. Ainsi avec la définition de ces contraintes, nous obtenons la liste des chemins légaux du processus. D'autre part, lorsque nous avons des chemins impliquant une boucle, la création d'un groupe est influencée. En effet, le chemin le plus long est augmenté, ce qui implique que nous devons avoir plus d'*items* définis. D'autre part, les implications définies pour le *cross* doivent tenir compte des contraintes pouvant être appliquées sur les chemins tel les limites d'une boucle.

Les groupes de couverture pour les flots transactionnels sont construits avec la même approche. Cependant dans le cas où il y aurait abstraction, il faut considérer la hiérarchie d'abstraction définie. Nous allons reprendre le modèle *M* présenté à la Figure 4-7 pour expliquer la création d'un groupe de couverture pour les flots transactionnels. L'exemple de la Figure 4-13 présente un groupe de couverture en ne considérant aucune abstraction.

```
1: cover flow_coverage is {
2:    item path_ps1 using illegal = not(path_ps1 in [PS1_PATH_1, PS1_PATH_2, PS1_PATH_3]);
5:    item path_ps2 using illegal = not(path_ps2 in [...]);
7:    item path_ps3 using illegal = not(path_ps3 in [...]);
8:    item path_ps4 using illegal = not(path_ps4 in [PS4_PATH1, PS4_PATH_2,
9:                                                   PS4_PATH_3, EMPTY]);
10:   item path_ps5 using illegal = not(path_ps5 in [...]);
11:
12:   cross path_ps1, path_ps2, path_ps3, path_ps4, path_ps5
13:      using illegal = not(
14:         (path_ps1 == PS1_PATH_1) => (path_ps2 != PS2_PATH_4)
15:      and ...);
16: };
```

Figure 4-13: Exemple d'un groupe de couverture pour la couverture des flots

Dans le cas du groupe de couverture permettant la couverture des flots transactionnels, les *items* sont maintenant les chemins des processus et le *cross* est le croisement entre les chemins empruntés dans le système de traitement des transactions. L'échantillonnage des informations définies dans le groupe de couverture se fait à l'émission de l'événement *flow_coverage*, qui est produit par le collecteur des flots transactionnels. Les contraintes appliquées sur le *cross* sont encore définies avec une série d'implications. L'exemple exposé à la ligne 14 de la Figure 4-13 signifie que lorsque le chemin du processus P_1 est *PS1_PATH_1*, il est illégal que le chemin du processus P_2 soit *PS_PATH_4*.

Pour la création d'un groupe de couverture incluant une abstraction, nous réutiliserons l'exemple de la Figure 4-9 qui inclut l'abstraction d'un élément hiérarchique séquentiel. La construction d'un groupe de couverture pour le modèle *M* incluant une abstraction est semblable au cas d'un groupe sans abstraction. La Figure 4-14 illustre la différence.

```
1: cover flow_coverage is {
2:    item path_ps1 using illegal = not(path_ps1 in [PS1_PATH_1, PS1_PATH_2, PS1_PATH_3]);
5:    item abs_se1  using illegal = not(path_ps2 in [ABSTRACT, EMPTY]);
8:    item path_ps4 using illegal = not(path_ps4 in [PS4_PATH_1, PS4_PATH_2,
9:                                                   PS4_PATH_3, EMPTY]);
10:   item path_ps5 using illegal = not(path_ps5 in [...]);
11:
12:   cross path_ps1, path_ps1, abs_se2, path_ps4, path_ps5
13:      using illegal = not(...);
14: };
```

Figure 4-14: Exemple d'un groupe de couverture pour la couverture des flots avec abstraction

Premièrement nous n'avons plus un *item* pour chaque processus du module, mais un *item* pour chaque élément hiérarchique du modèle. Donc, l'*item abs_se1* représente la couverture de l'élément hiérarchique séquentiel. Cet *item* peut seulement prendre la valeur *ABSTRACT* ou *EMPTY*. D'autre part, le *cross*, qui représente les flots transactionnels, inclut aussi l'*item* abstrait. Finalement, les contraintes permettant de restreindre l'ensemble des possibilités des flots transactionnels doit aussi être révisées afin d'inclure les valeurs abstraites.

Bien que les groupes de couverture doivent être construits de façon systématique, il y a un problème qui découle de l'abstraction. En fait, l'utilisateur du MAC doit être en

mesure de configurer l'abstraction qu'il désire obtenir. En sachant que les groupes de couverture nécessaires dans le module d'analyse sont différents pour chaque combinaison d'interrupteurs d'abstractions, il faudrait que le concepteur définisse des groupes de couverture pour toutes les combinaisons d'abstraction. Contrairement aux traqueurs de chemins et au collecteur des flots transactionnels qui travaillent seulement dans deux modes (normal et abstrait), l'implémentation d'un groupe de couverture n'est pas vraiment flexible.

La solution proposée est de générer automatiquement les groupes de couverture à chaque utilisation du MAC en fonction des paramètres d'abstraction. Cela est possible en utilisant les macros spéciales du langage *e*. Donc, le concepteur ne doit pas créer les groupes de couvertures pour la couverture des chemins et des flots transactionnels. Il a seulement à créer deux macros qui permettront une reconfiguration du MAC à chaque utilisation. Le détail de ces macros spéciales sera présenté à la prochaine section.

Récapitulation du traitement d'une transaction par le MAC

Le module a été conçu de façon à traiter l'application de plusieurs transactions en même temps. La Figure 4-15 résume le traitement d'une transaction par le système.

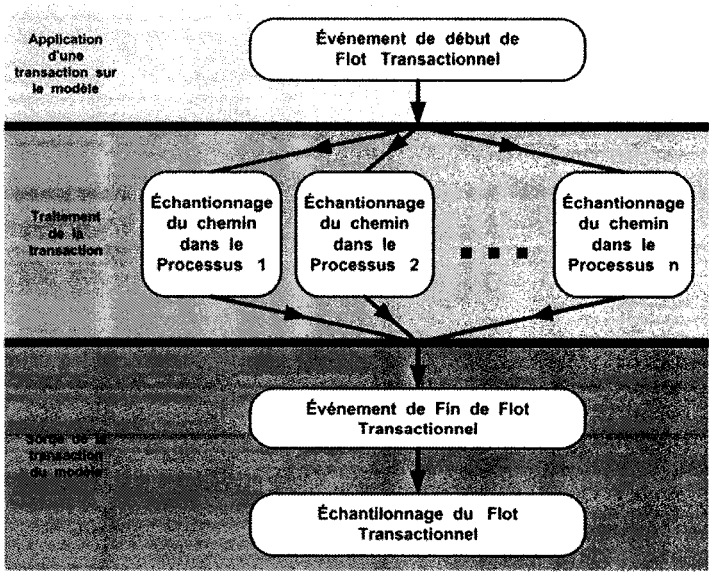


Figure 4-15: Séquence de traitement d'une transaction par le système de couverture

Donc, l'application d'une transaction sur le modèle implique l'émission d'un événement de début de flot transactionnel, ce qui démarre le mécanisme de couverture de la transaction. Ensuite, la transaction est traitée par la spécification exécutable, ce qui implique l'échantillonnage des chemins exercés par la transaction. Les chemins exercés sont envoyés au module de couverture de Specman Elite™, par l'intermédiaire des groupes de couverture, pour obtenir le premier résultat de couverture : la couverture des chemins. Lorsque le traitement de la transaction par la spécification exécutable est terminé, un événement de fin de flot transactionnel est émis. Ce qui produit l'échantillonnage du flot transactionnel et l'envoi du flot au module de couverture de Specman Elite ™. Specman Elite™ fournit des rapports textuels sur la couverture en fonction des éléments décrits dans les groupes de couverture. Ces résultats sont aussi visualisables avec une interface graphique fournie par Specman Elite™. Le schéma présenté à la Figure 4-15 représente le traitement d'une instance d'une transaction. Cependant la Figure 4-16 présente le traitement de plusieurs transactions par le système de couverture.

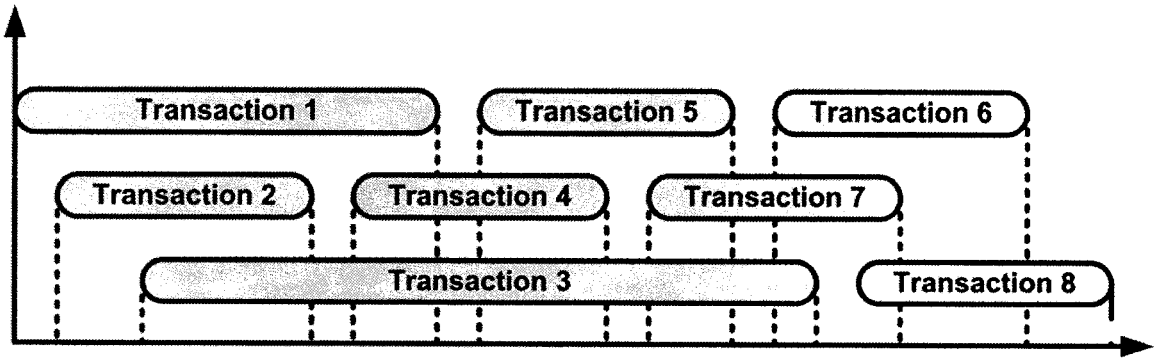


Figure 4-16: Traitement de plusieurs transactions

À la Figure 4-16, on peut considérer chaque bulle comme le diagramme présenté à la Figure 4-15. Le numéro de transaction peut être considéré comme le jeton de la transaction. L'axe des X ne représente pas le temps, car la spécification exécutable est définie de façon intemporelle. L'axe X représente plutôt l'ordre d'arrivée et de sortie des transactions du système, en quelque sorte leur durée de vie relativement aux autres

transactions. En fait, le schéma de traitement de plusieurs transactions, tel qu'illustré à la Figure 4-16, dépend du modèle auquel la métrique est appliquée.

4.3.2 Reconfiguration Automatique

À la section précédente, le fonctionnement des éléments du MAC a été détaillé. Il est cependant ressorti qu'il n'est pas possible de créer les groupes de couverture nécessaires à l'échantillonnage des mesures de couverture pour chaque mode d'abstraction. Ainsi dans cette section, une méthode permettant une reconfiguration automatique du MAC est exposée. Cette méthode passe par une génération automatique des groupes de couverture nécessaires en fonction de l'abstraction choisie. La réalisation de cette génération automatique utilise les macros spéciales du langage *e*. Ainsi, un résumé des types de macros du langage *e* permettra de consolider l'explication de la méthode. Ensuite, les algorithmes utilisés pour la création automatique des groupes seront discutés.

4.3.2.1 Les macros du langage *e*

Une macro, au sens de la programmation, sert à effectuer des remplacements dans un code de manière plus ou moins simple. On définit une macro en lui définissant un nom et un remplacement. Lorsque le nom de la macro sera utilisé dans le code, ce nom sera remplacé par le code approprié lors de la compilation du code. De plus, le nom d'une macro peut généralement être paramétré afin que le code de remplacement soit conditionnel aux paramètres. Le langage de vérification *e* supporte ce type de macros en les définissant comme des macros de remplacement simples. Le langage *e* possède un autre type de macros qui appelle les macros de remplacement complexes. L'avantage de ces macros est qu'elles sont beaucoup plus flexibles au niveau des remplacements possibles. Pour une macro simple, on définit le remplacement en spécifiant le code au travers les paramètres de la macro. Cependant, pour une macro complexe, le remplacement est défini par une fonction qui retourne une chaîne de caractères. Donc, les macros complexes sont en fait définies en générant une chaîne de caractères qui servira de remplacement dans le code. Le code généré peut être n'importe quel élément

syntaxique du langage *e*. De plus, la génération du remplacement peut être basée sur n'importe quelle structure de donnée définie de façon globale dans le langage *e*.

4.3.2.2 Algorithmes de génération des groupes de couverture

Les macros de remplacement complexes sont utilisées pour implanter des algorithmes qui permettent la génération des groupes de couverture pour la mesure de la couverture des chemins et des flots transactionnels. Ces algorithmes ne doivent pas être implémentées par le concepteur puisqu'ils sont complètement génériques. Plus précisément, la version développée, qui est en annexe, peut être directement utilisée par la création de n'importe quel MAC. Ces algorithmes génériques utilisent en fait les structures de données précédemment discutées dans ce chapitre. Ces données sont :

- Le dictionnaire des chemins;
- La base de données des BB;
- La liste de contraintes;
- La définition de la hiérarchie de l'abstraction.

Ces éléments ont des formats qui n'ont pas été formellement expliqués. Ces formats influencent le fonctionnement de la méthode. D'autre part, puisque les algorithmes sont implémentés avec des macros de remplacement complexes, il est nécessaire que les structures de données soient définies de façon globale. Une description conceptuelle des algorithmes développés est fournie en mettant en contexte les structures de données utilisées.

Génération des groupes de couverture pour les chemins des processus

Premièrement, il y aura seulement la création d'un groupe de couverture pour les chemins d'un processus qui ne sont pas abstraits. Donc, la définition de l'abstraction doit être consultée afin de connaître si le processus est abstrait. Dans le cas, où on doit créer un groupe, il faut savoir combien il y a d'items dans le groupe. Ce nombre correspond à la longueur du chemin le plus long. Cette information est disponible dans le dictionnaire des chemins qui inclut la définition de tous les chemins du modèle de couverture. Ensuite pour tous les *items*, il faut savoir leurs valeurs légales. On retire cette information du

dictionnaire des chemins en inspectant les chemins définis. C'est le même procédé pour la définition de l'élément *cross* pour l'échantillonnage des chemins. Le dictionnaire des chemins est utilisé pour construire les implications booléennes qui permettront d'échantillonner correctement les chemins. De plus, l'algorithme tient compte des contraintes qui auraient pu être ajoutées dans la liste de contraintes.

Génération du groupe de couverture pour les flots transactionnels

La génération du groupe de couverture pour les flots transactionnels est un ordre de grandeur plus complexe. En fait, le nombre d'*item* de groupe de couverture dépend de ce qui est abstrait et de ce qui ne l'est pas. La définition de la hiérarchie est donc utilisée afin de trouver les éléments abstraits du modèle. Donc, en explorant la définition de la hiérarchie, il est possible de trouver les éléments hiérarchiques abstraits et les processus qui ne le sont pas. Dans le cas où nous avons un élément hiérarchique abstrait, alors un *item* est créé et les valeurs légales de cet *item* sont soit *ABSTRAIT* ou *VIDE*. Autrement, nous devons définir un *item* pour un processus et les valeurs légales des chemins sont trouvées dans le dictionnaire des chemins. Pour la définition de l'élément *cross*, on redéfinit tous les *items* énumérés soit pour les éléments hiérarchiques ou pour les processus. On utilise encore la définition de la hiérarchie afin de définir les implications booléennes qui permettent de restreindre l'ensemble des flots transactionnels légaux. De plus, l'algorithme tient compte des contraintes qui auraient pu être ajoutées dans la liste de contraintes

4.4 Discussion

Dans cette section, nous allons discuter quelques points concernant la méthodologie permettant de créer un MAC. En premier lieu, le MAC est intégrable dans un banc d'essais en utilisant la méthodologie de conception présentée au chapitre 3. Ainsi, nous allons expliquer comment un MAC est intégrable avec la méthode de conception de banc d'essais. Ensuite, la création de façon systématique d'un MAC laisse présager des possibilités d'automatisation de ce processus. De ce fait, une ébauche du fonctionnement

d'un outil permettant la génération automatique d'un MAC sera exposée. Ensuite, quelques extensions possibles de la méthode de couverture seront présentées. Finalement, il sera question de l'efficacité de la vérification en utilisant la méthode de vérification fonctionnelle présentée dans ce chapitre.

4.4.1 Intégration du module avec la méthodologie de conception

Un MAC peut être utilisé sans avoir le modèle sous vérification. Cela permet d'élaborer des suites de tests efficaces avant que l'implémentation RTL soit prête à être vérifiée. Cependant, il faut avoir le banc d'essais dédié à l'implémentation RTL. C'est-à-dire, un générateur de transaction qui sera basé sur une approche de génération déterministe ou aléatoire. La Figure 4-17 (a) présente l'utilisation d'un MAC sans l'implémentation RTL.

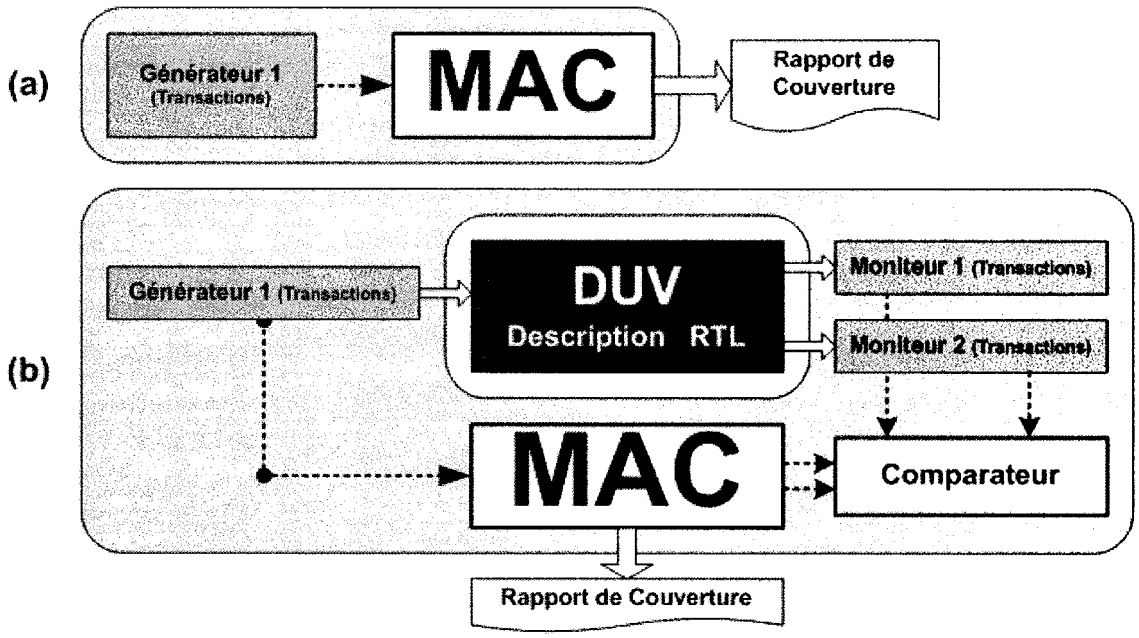


Figure 4-17: Configuration d'un MAC dans un banc d'essais

Ainsi, la Figure 4-17 (a) illustre un générateur de transactions à haut-niveau stimulant le MAC. Le générateur et le MAC sont définis indépendamment et ils sont connectés par un aspect de Couverture selon la méthode conception. Le résultat produit, suite à la stimulation du MAC à l'aide du générateur, est le rapport de couverture fournissant le

taux de couverture obtenu suite à l'application des transactions. La Figure 4-17 (b) illustre l'utilisation d'un MAC avec l'implémentation RTL. Si on connaît l'efficacité de la suite de test, il n'est pas vraiment nécessaire d'utiliser le MAC. Cependant, il y a une autre utilité du MAC que nous n'avons pas encore exposé. Elle consiste à utiliser le MAC pour la vérification automatique des réponses de l'implémentation RTL. Puisque le MAC inclut une version exécutable du modèle au niveau transactionnel, il est possible d'utiliser les réponses du MAC pour les comparer à celle de l'implémentation. Donc à la Figure 4-17 (b), nous avons toujours le même générateur de transactions et deux moniteurs de transactions pour recueillir les réponses du DUV. Un aspect HDL est utilisé pour complémenter le générateur et les moniteurs dans le but de définir comment appliquer ou recueillir les transactions sur le DUV. Une classe de vérification, le comparateur, est utilisée pour comparer les transactions provenant du DUV et du MAC. Les moniteurs sont connectés au comparateur avec un aspect de connexion. Finalement, le générateur et le comparateur sont reliés au MAC par un aspect de couverture. De plus, le rapport de couverture est toujours disponible suite à une simulation.

4.4.2 Automatisation

L'implémentation d'un MAC peut être une tâche fastidieuse, puisqu'il y a beaucoup d'information à retirer de la spécification exécutable et aussi beaucoup de paramètres à fixer. Cependant, en ayant défini toutes les étapes permettant de créer un MAC de façon systématique, il est possible d'automatiser sa création. La Figure 4-18 présente le fonctionnement d'un outil qui permettrait cette automatisation.

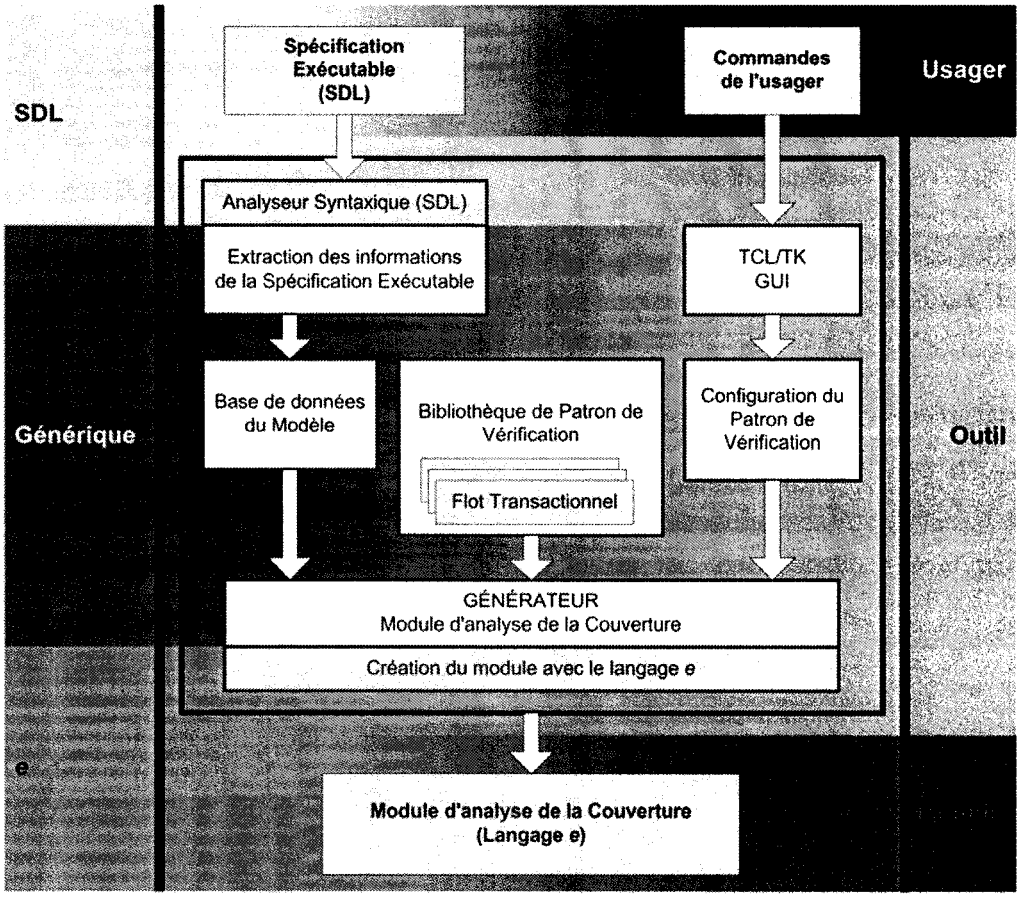


Figure 4-18: Automatisation de la création d'un MAC

Les entrées d'un tel outil seraient composées de la spécification exécutable, décrite en SDL, et des commandes de l'usager. Une interface graphique pourrait assister l'utilisateur dans la création du MAC. L'outil devrait inclure un analyseur syntaxique qui permettrait d'abstraire les informations pertinentes de la spécification exécutable et de placer ces informations dans une base de donnée interne à l'outil. Ensuite, une bibliothèque de patrons de vérification serait disponible pour définir comment implanter un MAC pour une métrique précise. Évidemment, la méthode de couverture ne supporte qu'un patron de vérification. Cependant en généralisant, il pourrait y avoir plusieurs patrons de vérification. L'interface graphique serait alors utile afin de configurer le patron de vérification. Des exemples de configuration seraient la création de contrainte pour les flots transactionnels ou encore la définition de la hiérarchie d'abstraction. Le MAC serait généré en fonction des informations de la spécification exécutable, du patron

de configuration et de commandes de l'usager. Finalement, en ayant un tel outil, il serait facilement imaginable de créer le MAC en utilisant différents langages de vérification pour l'implémentation.

4.4.3 Extension de la méthode de couverture

La première extension possible à la méthode de couverture serait d'utiliser d'autres métriques de couverture fonctionnelle. Cela signifie qu'il faudrait d'autres patrons de vérification qui permettraient d'échantillonner la mesure de la couverture de ces autres métriques. Une métrique qui pourrait être définie est un modèle de faute fonctionnelle au niveau transactionnel. Un ensemble de fautes fonctionnelles pourrait être défini sur les éléments décisionnels d'une spécification exécutable. Ensuite, le MAC fournirait un taux de couverture en fonction du nombre de fautes contrôlées par rapport au nombre de fautes totales. Cela permettrait de constituer une bibliothèque de patrons de vérification.

D'autre part, l'utilisation de SystemC comme langage de spécification est aussi envisageable. Pour l'instant, Specman Elite™ est utilisé pour effectuer une co-simulation du banc d'essais en *e* avec le modèle HDL. Il n'est pas possible d'inclure le SDL dans cette co-simulation, c'est pourquoi il est nécessaire de créer une version en langage *e* de la spécification exécutable dans le MAC. Cependant il est possible d'ajouter du C++ en co-simulation avec un banc d'essais en langage *e* et un modèle HDL. La spécification exécutable décrite en SystemC pourrait être directement utilisée dans le MAC. Il y aurait seulement quelques modifications à faire dans le but d'extraire les événements pertinents du modèle SystemC afin d'échantillonner les flots transactionnels.

Finalement, l'extension qui permettrait d'obtenir une méthode de couverture fonctionnelle complète est de biaiser la génération des transactions en fonction du taux de couverture actuel de la métrique fonctionnelle. Présentement, la génération des transactions peut se faire de façon pseudo-aléatoire ou déterministe. En fonction, des résultats de simulation, l'utilisateur du MAC va manuellement biaiser la génération pseudo-aléatoire ou définir de nouveaux tests déterministes. Cette extension permettrait de rendre la génération de transactions adaptatives en fonction des résultats de simulation obtenus.

4.4.4 L'efficacité de la vérification avec cette méthode

L'utilisation d'un MAC produit en utilisant cette méthode permet d'obtenir une mesure d'avancement en fonction de la métrique utilisée. L'obtention d'un bon taux de couverture permet d'augmenter le degré de confiance du concepteur en regard de l'implémentation du circuit. Cependant cela n'est pas suffisant pour affirmer que nous avons une vérification fonctionnelle complète. Il y a d'autres aspects de la fonctionnalité d'un modèle qu'il faut vérifier. Par exemple, en travaillant au niveau transactionnel, on ne tient pas compte des éléments de bas niveau tel que les protocoles de communications aux interfaces du modèle. Ainsi, l'utilisation de la méthode de couverture décrite dans ce chapitre est un élément faisant partie d'une vérification complète d'un modèle.

D'autre part, cette métrique n'est pas adéquate pour tous les types de modèle. Afin d'appliquer cette métrique, il faut être en mesure de représenter le modèle sous vérification sous la forme d'un système de traitement de transactions. Autrement, il ne sera pas possible d'appliquer le patron de vérification défini.

Chapitre 5

APPLICATION DES MÉTHODES

Les méthodes de conception de bancs d'essais et de couverture fonctionnelle proposées dans ce mémoire ont été appliquées sur différents modèles. Le but de la méthode de conception de banc d'essais étant de faciliter la réutilisation de certaines parties d'un banc d'essais, nous démontrerons comment notre méthode de conception permet d'atteindre cet objectif. Pour ce qui est de la méthode de couverture, le fait de créer un module d'analyse de la couverture de façon systématique est déjà un résultat en soi. De plus, ce chapitre présente les résultats de l'utilisation de ce module d'analyse de la couverture en regard de la possibilité d'améliorer la couverture fonctionnelle des circuits numériques. Trois modèles ont été utilisés pour l'application des méthodes de vérification. Cette section sera organisée en fonction des modèles utilisés pour l'application des méthodes. En premier lieu, il y a un système de conversion où la méthode de conception de bancs d'essais a été appliquée. Ensuite, un commutateur ATM simple a été utilisé pour appliquer la méthode de conception et la méthode de couverture. Finalement, la méthode de couverture a été appliquée sur un modèle industriel fourni par la société PMC-Sierra.

5.1 Convertisseur de protocoles

Le convertisseur de protocoles est un projet développé au GRM. Ce projet implique des étudiants au niveau de la maîtrise, ainsi que des étudiants au baccalauréat réalisant leur projet de fin d'études. Le but de ce projet est de définir une architecture générique permettant la transmission d'un flot de données à haut débit à travers des réseaux hétérogènes. Plus précisément, il s'agit de permettre le passage d'un flot de données sur un réseau utilisant le protocole A vers un réseau utilisant le protocole B. L'architecture sur laquelle nous avons appliqué la méthode de conception de banc d'essais est la version 2 du projet. Cette version consiste à permettre la conversion d'un flot de données

formatées comme un ensemble de paquets FireWire en un flot de paquets Ethernet. Le but de créer cette version de l'architecture est d'explorer les aléas de la conversion de protocoles, afin de généraliser au cas d'une plate-forme générique.

5.1.1 Présentation du modèle

La Figure 5-1 présente l'architecture simplifiée de la version 2 du convertisseur de protocoles. Cette version simplifiée ne contient pas exactement tous les éléments du système, mais il y a suffisamment d'information pour en expliquer le fonctionnement.

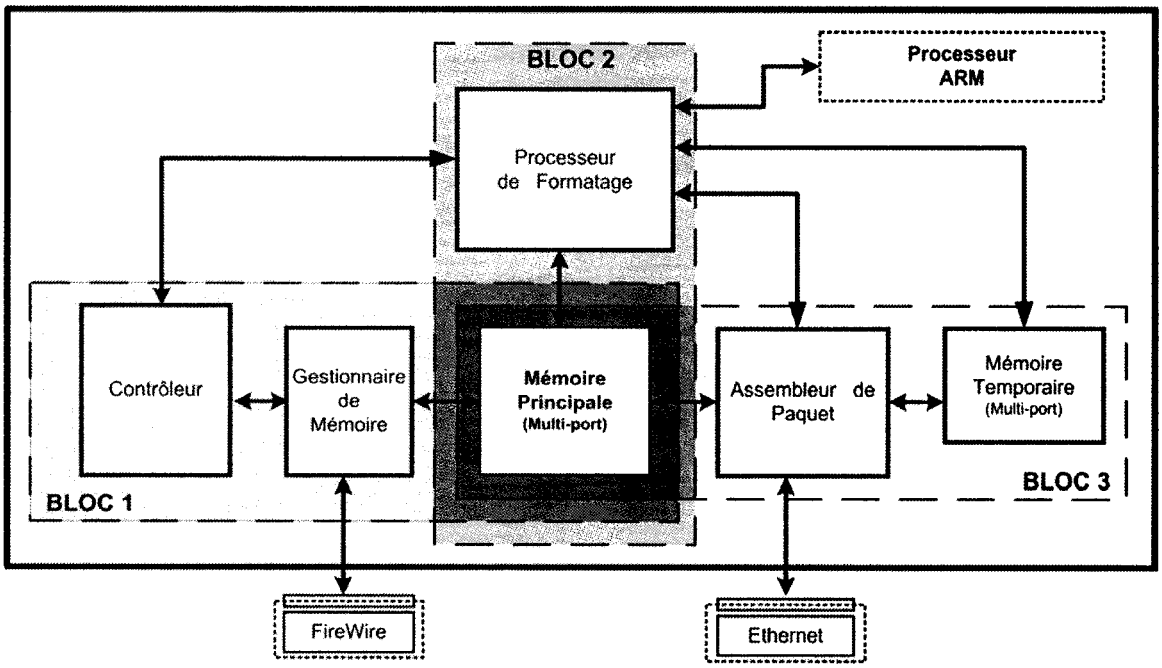


Figure 5-1: Architecture simplifiée du convertisseur de protocoles

Comme mentionné ci-haut, les entrées du système de conversion de protocoles sont donc des paquets FireWire et les sorties sont des paquets reformatés selon la norme Ethernet. L'architecture est constituée de trois blocs : Le bloc 1 s'occupe de la réception des paquets FireWire dans le système, le bloc 2 reformate le paquet à la norme Ethernet et le bloc 3 s'occupe de la transmission du paquet reformaté. De plus, un processeur ARM est utilisé pour configurer le système et gérer les exceptions.

Le bloc 1 est constitué de 3 modules. Premièrement, il y a la mémoire principale qui est utilisée par les trois modules de l'architecture. Cette mémoire a une capacité de 5

paquets. Le gestionnaire de mémoire reçoit les paquets FireWire. Il accepte des paquets tant qu'il y a de la place dans la mémoire principale. Lors de la réception d'un paquet, il place le paquet dans un espace mémoire disponible. De plus, il va compter le nombre d'octets composant le paquet. Lorsque la réception d'un paquet est complétée, le gestionnaire de mémoire envoie une étiquette au module contrôleur qui contient l'adresse où est placée en mémoire le paquet et la taille du paquet. Le gestionnaire de mémoire a aussi la responsabilité de supprimer les paquets de la mémoire lorsqu'il en reçoit l'ordre du module contrôleur. Le module contrôleur sert à gérer le traitement des paquets par le système. Ainsi, il communique avec le gestionnaire de mémoire et le processeur de formatage du bloc 2. Lorsque le contrôleur reçoit une étiquette du gestionnaire de mémoire, il place cette étiquette dans un *First-In First-Out* (FIFO). Ensuite, si le processeur de formatage est prêt, c'est-à-dire qu'il n'est pas en cours de traitement d'un paquet, une étiquette est envoyée au processeur de formatage. Le processeur de formatage traitera le paquet et informera le contrôleur lorsqu'il a terminé le traitement. Conséquemment, le contrôleur enverra un ordre de suppression pour le paquet traité au gestionnaire de mémoire et s'il y a d'autres paquets en attente de traitement, il transmettra une autre étiquette au processeur de formatage.

Le bloc 2 inclut essentiellement le processeur de formatage et la mémoire principale. Le processeur de formatage est un processeur *Reduced Instruction Set Computer* (RISC) dédié à la manipulation de données. Le processeur possède une mémoire d'instructions et 16 registres. La mémoire d'instructions contient l'algorithme de conversion des entêtes FireWire à Ethernet. Cet algorithme est placé dans la mémoire d'instructions par l'intermédiaire du processeur ARM. Lorsque le processeur de formatage reçoit, en provenance du contrôleur, l'étiquette d'un paquet, l'algorithme de conversion démarre. Cet algorithme va chercher, en fonction de l'étiquette du paquet traité, l'entête du paquet FireWire en mémoire principale et produire une entête Ethernet. L'entête reformatée sera placée dans la mémoire temporaire du bloc 3. Ces étiquettes servent à indiquer à l'assembleur de paquets comment rassembler le paquet dans le format Ethernet. Ainsi, une étiquette inclut l'adresse et la taille de l'entête Ethernet dans la mémoire temporaire,

alors que l'autre inclut l'adresse et la taille de la charge utile du paquet en mémoire principale. Il est à noter que le bloc 2 inclut aussi des coprocesseurs pour assister le processeur de formatage dans la conversion de l'entête FireWire. Ces coprocesseurs n'étaient pas encore développés lors de l'élaboration de la version 2 du système et ils ne sont pas nécessaires à la compréhension de l'architecture.

Le bloc 3 sert à la retransmission du paquet sur le réseau Ethernet. Il est constitué de la mémoire principale, de la mémoire temporaire et de l'assembleur de paquets. Donc, lorsque l'assembleur de paquets reçoit des étiquettes en provenance du processeur de formatage, il commence la reconstruction du paquet. Premièrement, il va chercher l'entête Ethernet dans la mémoire temporaire et il transmet les octets de l'entête. Ensuite, il continue la transmission en passant, à la suite de l'entête, les octets de la charge utile du paquet qui se trouvent dans la mémoire principale. De plus, puisqu'un paquet Ethernet nécessite un CRC, calculé en fonction de tous les octets du paquet, à la fin du paquet, l'assembleur de paquets génère ce CRC et le transmet à la suite de la charge utile.

Cela résume les fonctionnalités de la version du convertisseur de protocoles auxquelles la méthode de conception de banc d'essais a été appliquée.

5.1.2 Vérification du convertisseur de protocoles

Les bancs d'essais dédiés à la vérification du système de conversion de protocoles ont été développés en parallèle avec l'implémentation RTL. Une approche de conception et de vérification ascendante a été adoptée dans ce projet. Les modules de l'architecture ont été conçus et vérifiés de façon unitaire et ensuite, une intégration graduelle a été faite jusqu'à l'obtention du système. La Figure 5-2 présente l'ordre de vérification des différents modules du système.

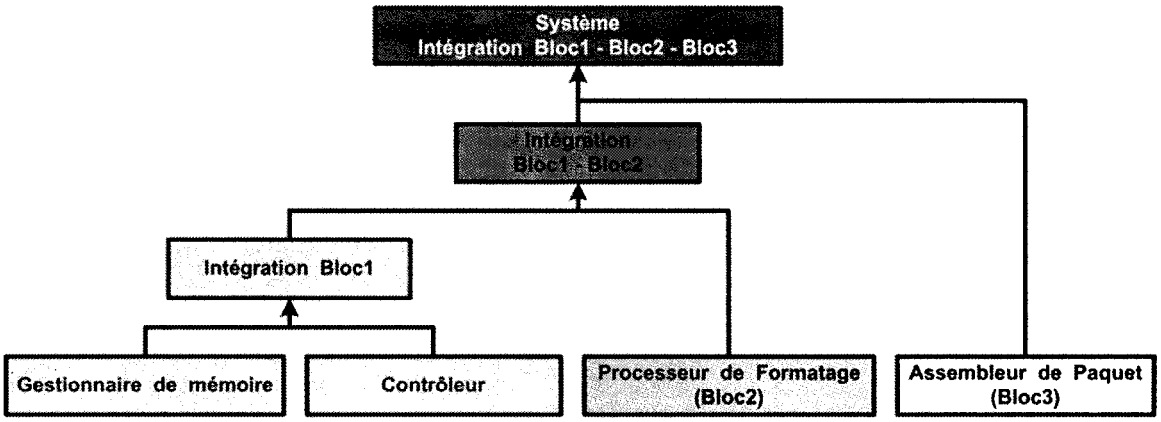


Figure 5-2: Approche de vérification ascendante

Sept bancs d'essais ont été développés en utilisant la méthode de conception de bancs d'essais. Le développement des bancs d'essais a été effectué dans le cadre d'un projet du cours ELE6305 et d'un projet de fin d'études au niveau du baccalauréat. Les bancs d'essais développés pour les modules unitaires ont été créés à partir de rien. Chaque banc d'essais unitaire développé a permis de vérifier le module comme s'il était dans son environnement d'opération réel, en créant une classe d'émulation pour chaque module connecté avec le DUV. De plus, tous les bancs d'essais permettent de vérifier automatiquement les réponses des DUV. Ainsi, la création de classes de vérification a permis d'adopter une stratégie de vérification automatique des réponses. En fait, les classes d'émulation et de vérification sont celles qui constituent le plus grand potentiel de réutilisation, puisque dans les bancs d'essais d'intégration, certaines de ces classes sont nécessaires. Certains facteurs pourraient empêcher la réutilisation directe de ces composants de banc d'essais. Par exemple, on peut citer les connexions avec d'autres modules de banc d'essais qui ne vont pas être réutilisées, ou encore les connexions avec le DUV. Le partitionnement par aspects utilisé pour la conception des bancs d'essais a permis d'éviter la création de classes de base contaminées. Une classe de base contaminée est une classe qui ne peut pas être directement réutilisée puisqu'elle contient des propriétés propres au banc d'essai pour laquelle elle a été créée. Donc, lors de l'intégration de deux modules, on a fusionné les bancs d'essais en réutilisant presque toutes les classes des deux bancs d'essais. Seules les classes émulant le comportement

des DUV intégrés n'ont pas été gardées au niveau des classes de base. Donc, pour les bancs d'essais d'intégration, les seuls éléments que nous devions définir ont été les aspects non réutilisés. Citons à titre d'exemple un aspect HDL pour définir les nouvelles connexions et les interactions du modèle sous vérification, un aspect de connexion pour définir les interactions entre les classes de base et un ou des aspects couverture et configuration pour créer les tests dans les environnements de vérification développés.

Pour les bancs d'essais servant à l'intégration des modules, on a observé que plus de 50 % du code source des bancs d'essais unitaires ont été réutilisés pour la création des bancs d'essais d'intégration. D'autre part, le développement des bancs d'essais pour la version courante du convertisseur de protocoles est encore basé sur cette méthode de conception de banc d'essais, et il réutilise en partie les bancs d'essais développés pour la vérification de cette itération du système de conversion de protocoles.

5.2 Commutateur ATM

Un commutateur ATM simplifié [7] a été utilisé pour appliquer la méthode de conception de bancs d'essais et la méthode de couverture. Dans cette section, un résumé des caractéristiques de ce modèle sera présenté. Ensuite, le banc d'essais développé avec la méthode de conception de bancs d'essais sera exposé. Finalement, l'application de la méthode de couverture et les résultats de simulation seront présentés.

5.2.1 Le fonctionnement du commutateur ATM

Le commutateur ATM sert de pont entre une interface usager et une interface réseau d'un réseau ATM. Pour résumer la spécification fonctionnelle du modèle, nous pouvons dire que ce modèle prend en entrée des cellules ATM de type *User Network Interface* (UNI) sur 4 ports d'entrées et qu'il reformate les cellules au format *Network to Network Interface* (NNI). Ensuite, les cellules reformatées sont routées vers les 4 ports de sorties. Le reformatage et le routage des cellules sont effectués en fonction d'un fichier de registres qui est introduit dans le modèle par une interface de contrôle. Toutes les interfaces impliquées dans ce modèle sont détaillées dans un document de la norme ATM

[2]. La Figure 5-3, tiré de la spécification fonctionnelle du modèle, présente la configuration des interfaces de ce modèle.

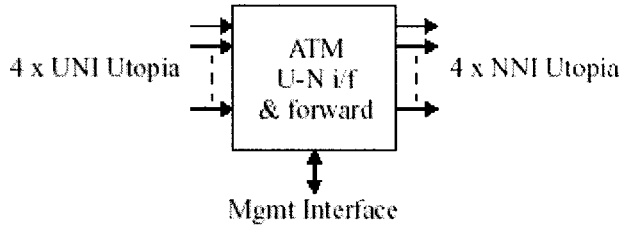


Figure 5-3: Interfaces du design ATM

Maintenant, pour ce qui est du reformatage des cellules de format UNI vers le format NNI, la transformation implique seulement une légère modification dans l'entête de la cellule UNI et le chargement de la cellule ATM est préservé. La Figure 5-4 montre les formats des deux types de cellule ATM impliqués dans ce design.

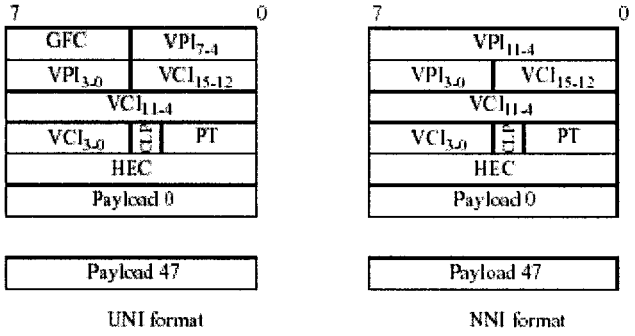


Figure 5-4: Format des cellules ATM

Cependant, pour qu'une cellule soit reformatée et routée, il faut que la cellule ATM soit valide. La validité d'une cellule est déterminée par son champ *Header Error Control* (HEC) : un CRC de 8 bits sur les 4 premiers octets de l'entête. Ainsi, lorsque le HEC d'une cellule est erroné, la cellule est éliminée. Autrement, lorsque la cellule est intègre, le reformatage s'effectue de la façon suivante:

- Remplacement des champs GFC (4 bits) et VPI (8 bits) dans l'entête UNI par un nouveau VPI (12 bits) dans l'entête NNI;
- Remplacement du HEC par un nouveau HEC qui est calculé en fonction des nouveaux octets de l'entête.

La manière utilisée afin de déterminer le nouveau champ VPI de l'entête NNI est d'utiliser le fichier de registres actualisé par l'interface de contrôle. Ce fichier de registres contient 256 vecteurs de 16 bits. La configuration des vecteurs contenus dans ce fichier de registres est présentée à la Figure 5-5.

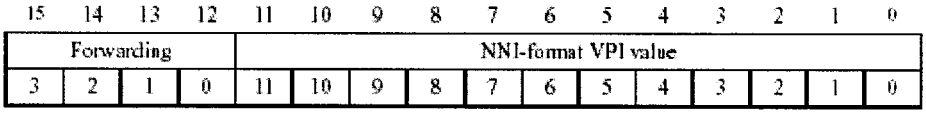


Figure 5-5: Vecteur de réécriture et de routage

Ainsi, ce vecteur contient le nouveau champ VPI sur 12 bits et un vecteur de routage codé sur 4 bits. Le modèle utilise le VPI (8 bits) de la cellule UNI comme index pour accéder au bon vecteur de reformatage et de routage contenu dans le fichier de registres. Finalement, pour ce qui est du routage, chaque bit du vecteur de 4 bits destiné à cette fin est associé à un port de sortie. Ainsi, la cellule reformatée sera transmise vers les ports de sortie dont le bit, dans le vecteur de routage, est actif.

Ce modèle est très simple et il ne représente en rien un équipement pouvant fonctionner réellement dans un réseau ATM. Cependant, sa simplicité a été très utile pour le développement de la méthode de couverture.

5.2.2 Le banc d'essais

Un banc d'essais a été réalisé pour la cellule ATM. Ce banc d'essais a été réalisé en respectant la spécification du modèle et les normes ATM indiqués. Le banc d'essais a été modélisé au niveau transactionnel afin d'être utilisé en conjonction avec la méthode de couverture. L'utilisation de la méthode de conception permet une intégration facile du MAC qui a été produit.

En premier lieu, nous avons modélisé les transactions interagissant avec le design. Voici la liste des transactions :

- Une cellule ATM de format UNI;
- Une cellule ATM de format NNI;
- Une écriture sur l'interface de contrôle;

- Une lecture sur l'interface de contrôle.

Ensuite, nous avons modélisé l'architecture du banc d'essais qui est présentée à la Figure 5-6.

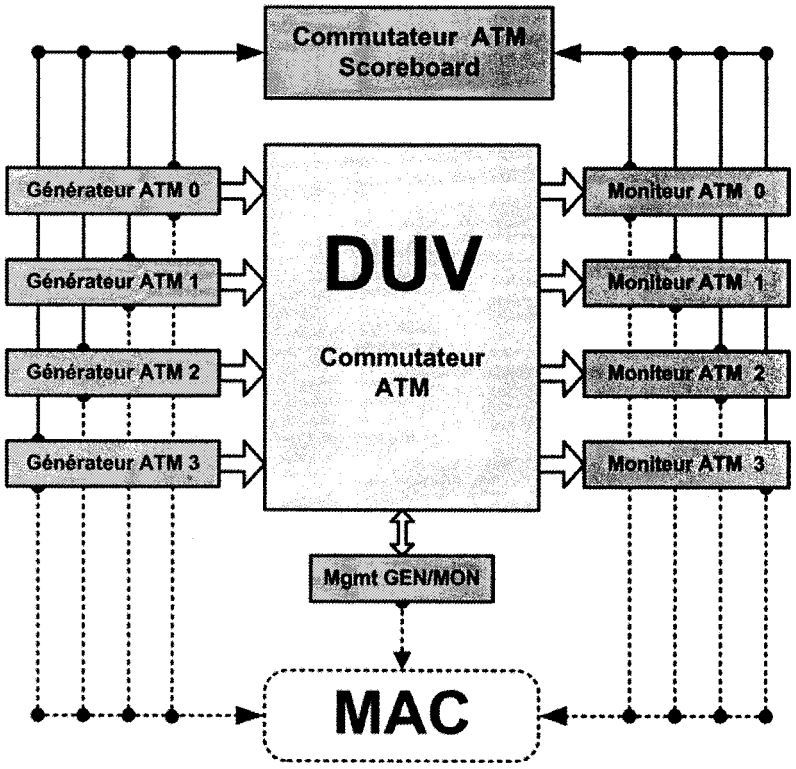


Figure 5-6: Architecture du banc d'essais du design ATM

Tel qu'illustré à la Figure 5-6, nous avons produit un générateur de transactions pour chaque port d'entrée et un moniteur de transactions pour chaque port de sortie. Ces éléments permettent d'injecter ou de recueillir les transactions appropriées sur les interfaces du design. De plus, ces éléments implémentent, dans l'aspect HDL, le protocole de bas niveau associé à chacune des interfaces. Les générateurs de transactions sont tous configurables en appliquant des aspects de configuration qui ajouteront des contraintes de génération au banc d'essais. Pour ce qui est de la vérification des résultats produits par le design, nous avons produit un *Scoreboard* qui vérifie les éléments suivants :

- Lorsqu'une cellule UNI valide est injectée, une ou des cellules NNI correspondantes et correctement reformatées doivent être recueillies sur les ports de sortie spécifiés par le vecteur de routage;
- Il ne doit pas y avoir de cellules NNI recueillies par les ports de sortie qui n'ont pas de source. La source est évidemment une cellule UNI.

Toutes les parties du banc d'essais sont connectées ensemble avec un aspect de connexion. Ainsi, on peut remarquer que tous les générateurs et moniteurs de transactions sont connectés au *Scoreboard*. Les interactions avec le MAC sont aussi exposées.

5.2.3 Application de la méthode de couverture

La méthode de couverture a été appliquée sur le commutateur ATM. L'application de la méthode sera décrite en trois étapes. Le développement de la spécification exécutable du commutateur ATM sera d'abord détaillé. Ensuite, il sera question de la création du MAC. Finalement, les résultats de l'utilisation du module seront exposés.

5.2.3.1 Développement de la spécification exécutable

Le modèle SDL du commutateur ATM a été développé avec l'aide du logiciel Telelogic Tau™. Ce logiciel permet une capture graphique d'un modèle SDL et il permet aussi sa simulation. Dans le but de ne pas surcharger ce document, seulement deux diagrammes de la modélisation SDL seront présentés. Cependant, toute la description SDL du commutateur ATM se trouve en annexe C.

La Figure 5-7 présente le diagramme du bloc principal du modèle SDL.

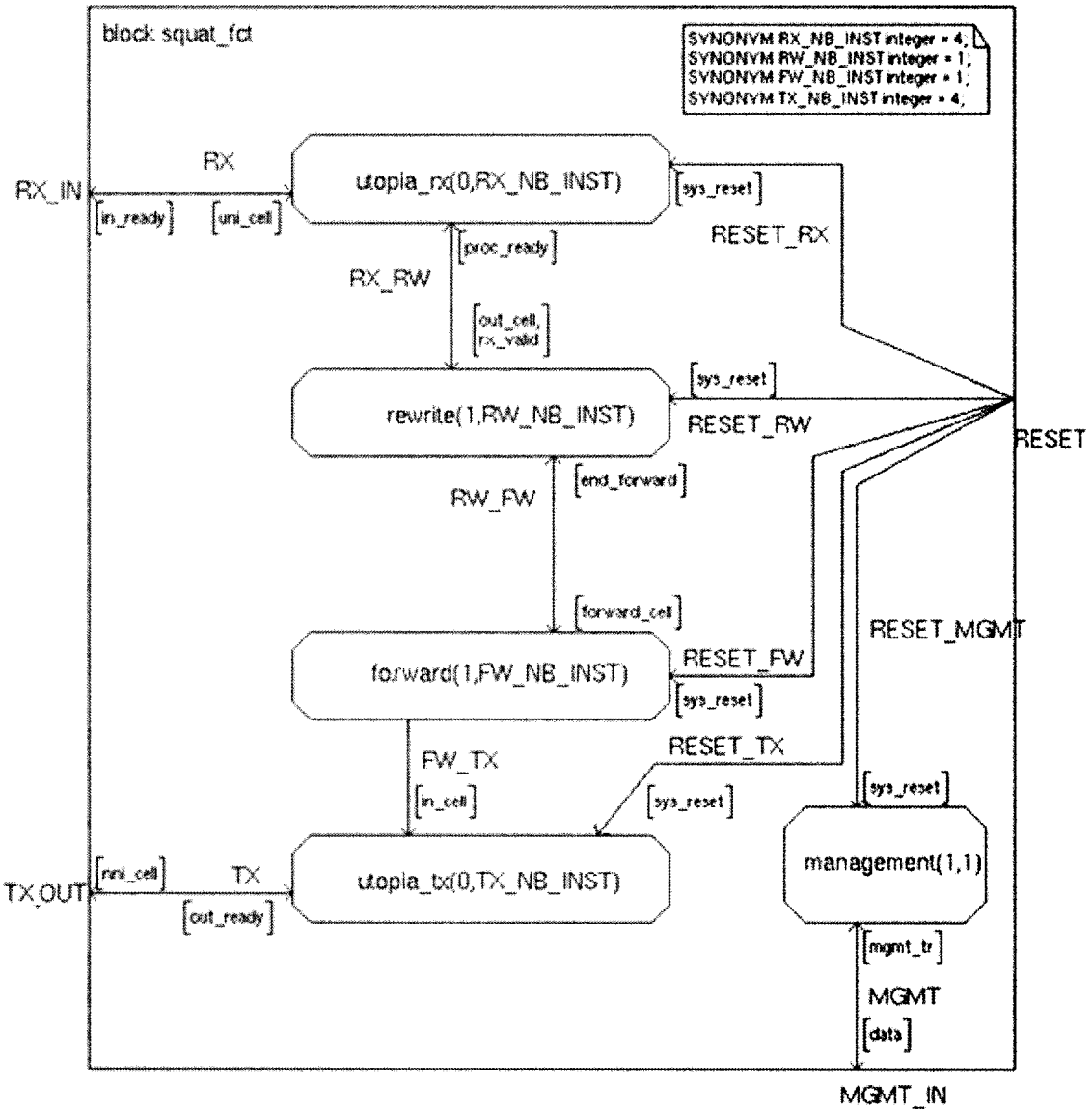


Figure 5-7: Description SDL Structurelle

La Figure 5-7 décrit les interactions entre les différents processus utilisés pour définir le comportement du modèle. Les interactions sont définies avec les canaux de communication et les signaux transmis par ces canaux. Les quatre processus *utopia_rx* servent à recevoir les cellules ATM de format UNI dans le modèle. Ensuite, le processus *rewrite* traite les cellules en reformatant les cellules valides et en éliminant les cellules erronées. Suite au reformatage, les cellules sont envoyées au processus *forward* pour

qu'elles soient transmises vers les bons ports de transmissions. Il y a quatre processus *utopia_tx* définissant la fonctionnalité des ports de transmission. Le processus *management* sert à actualiser la table de routage et de reformatage du modèle.

La Figure 5-8 est un exemple de processus : le processus *rewrite*.

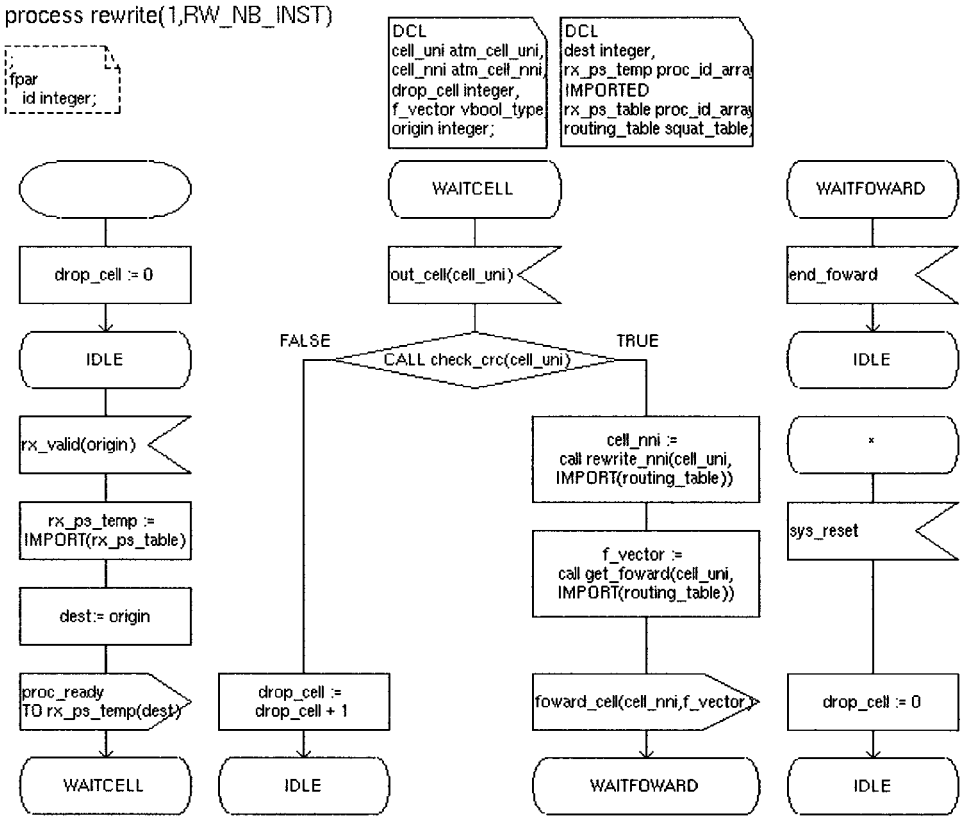


Figure 5-8: Description SDL comportementale

Ce processus est décrit en utilisant une CEFSM. Les transactions du modèle passeront par ce processus. En fait, ils arrivent par le signal *out_cell*. Ensuite, il y a une décision qui détermine si la cellule est valide. Dans le cas où la cellule est invalide, le flot transactionnel se terminera dans ce processus. Autrement, la transaction (cellule) sera reformatée et envoyée vers un autre processus, le processus *forward*, par le signal *forward_cell*.

5.2.3.2 Création du MAC

Le MAC a été créé manuellement avec le langage *e*. L'annexe D inclut les sources du MAC dédié au commutateur ATM. En fait, le module produit est une version configurée du patron de vérification décrit au chapitre 4. Cette configuration dépend de plusieurs points que nous allons réviser en fournissant quelques exemples :

Détermination du type de transaction couverte

Les transactions qui peuvent être appliquées sont des cellules ATM ou des opérations (lecture et écriture) sur l'interface de contrôle. Les transactions couvertes seront les cellules ATM.

Détermination des processus impliqués dans le modèle de couverture

Tous les processus du modèle sont impliqués dans le modèle de couverture à l'exception du processus *management*. Ce processus n'influence pas directement le traitement des transactions couvertes dans ce système.

Énumération des chemins dans tous les processus contenus dans P_C

Si nous prenons le processus *rewrite* présenté à la Figure 5-8, il y a seulement deux chemins possibles : la cellule est reformatée ou elle est éliminée. Cependant, dans le cas du processus *forward*, il y a plusieurs chemins dus à une boucle créée pour l'envoi de la cellule sur plusieurs ports de transmission.

Détermination des origines et des fins des flots transactionnels

Les flots transactionnels commencent par l'entrée d'une cellule ATM dans un des processus *utopia_rx*. Ils peuvent se terminer de différentes façons. Dans un premier temps, ils peuvent se terminer dans le processus *rewrite* dans le cas où la cellule est erronée. Ensuite, dans le cas où la cellule serait valide, la cellule reformatée peut être dupliquée vers 1 ou plusieurs processus de transmission (*utopia_tx*). Dans ces derniers cas, les flots transactionnels se termineront lors de la dernière transmission de la cellule dupliquée.

Détermination des contraintes des flots transactionnels

Il y a plusieurs contraintes dans ce modèle. Un premier exemple est la boucle dans le processus *forward*. Cette boucle représente en fait l'envoi de la cellule reformatée sur les

ports de transmission (*utopia_tx*), donc ses limites sont au minimum de une itération et au maximum de quatre itérations. Un autre cas de contrainte surgit lorsque la cellule est erronée. Ce cas implique que la valeur du chemin du processus *forward* et des processus *utopia_tx* seront *VIDE* puisque le flot transactionnel se terminera dans le processus *rewrite*.

Abstraction de la métrique

Pour utiliser l'abstraction de la métrique, il faut hiérarchiser les éléments composant le modèle. La Figure 5-9 présente la hiérarchie définie pour ce modèle.

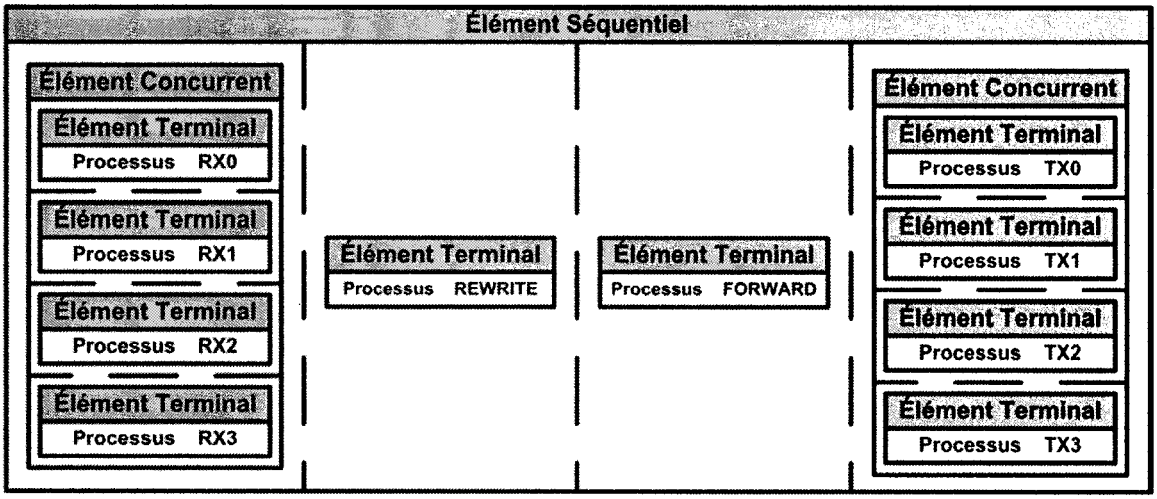


Figure 5-9: Hiérarchie d'abstraction du commutateur ATM

En ayant cette hiérarchie d'abstraction, nous sommes en mesure d'abstraire la métrique en abstrayant n'importe quel partie de la hiérarchie.

5.2.3.3 Création d'une suite de tests

L'utilisation de la métrique des flots transactionnels sur le commutateur ATM produit un ensemble de 64 flots transactionnels légaux. Cet ensemble est petit, car l'exemple est très simple. Cependant, le MAC a été utilisé pour créer une suite de tests permettant d'atteindre un taux de couverture de 100%. La génération des transactions est effectuée en utilisant le générateur pseudo-aléatoire de Specman Elite™. En utilisant les générateurs de cellules ATM définis dans le banc d'essais, des cellules ATM formatées correctement sont produites. Les cellules ATM correctement formatées incluent des

cellules ayant des erreurs dans leur CRC. D'autre part, à partir de l'interface de contrôle, une table de reformatage et de routage uniforme est introduite dans le modèle.

La Figure 5-10 présente les résultats obtenus en utilisant le module d'analyse de la couverture.

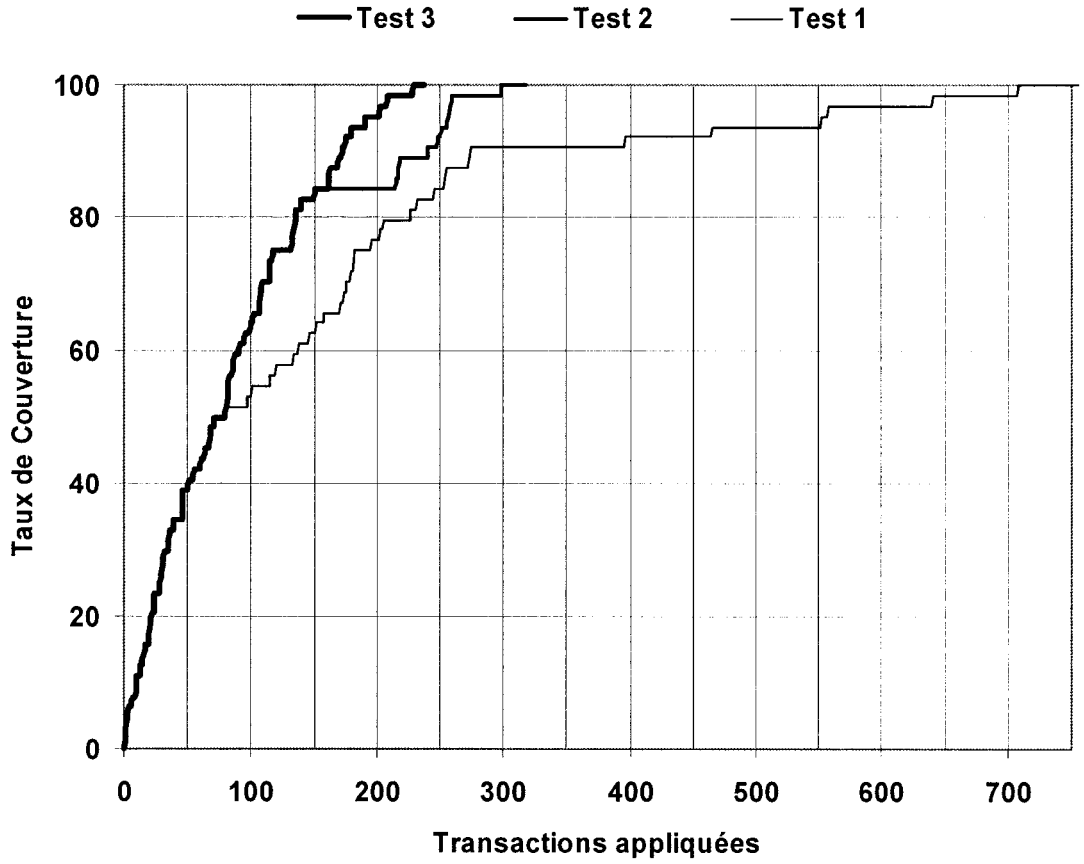


Figure 5-10: Résultats de couverture

Nous avons développé trois tests permettant d'obtenir un taux de couverture de 100%. Nous avons débuté l'expérimentation en déterminant le nombre de transactions appliquées de façon complètement pseudo-aléatoire (Test1) qui permettait d'obtenir un taux de couverture de 100%. Ainsi, nous avons obtenu un taux de couverture de 100% avec l'application de 708 transactions. Donc, pour accélérer la convergence de la couverture, un second test a été développé (Test2). Ce second test est en fait un nouvel

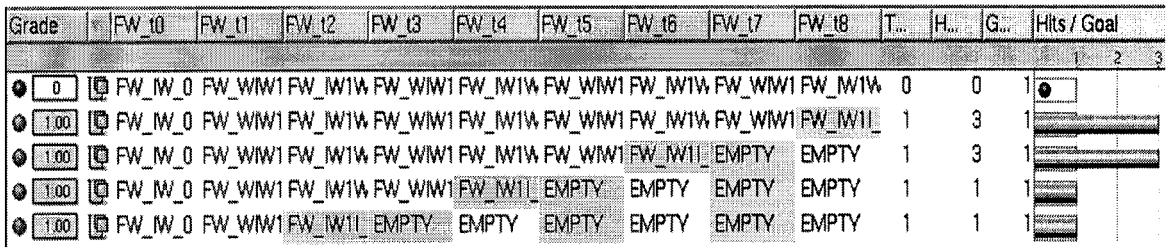
aspect de configuration définissant des contraintes de génération permettant de viser la couverture des flots transactionnels qui n'ont pas été couverts. En utilisant ce second test, nous avons obtenu 100% avec l'application de 299 transactions. Finalement, la création d'un troisième test (Test3) permet de réduire le nombre de transactions à 229. Donc, l'utilisation du MAC produit pour le commutateur ATM permet le développement de suites de tests qui utilisent moins de transactions qu'une génération complètement pseudo-aléatoire. Cela permet de réduire le temps de simulation tout en n'affectant pas la puissance de détection d'erreurs.

Le MAC produit des rapports de couverture textuels qui peuvent être visualisés avec l'interface graphique de Specman Elite™. Ces rapports dérivent des définitions faites dans les groupes de couverture du MAC. Voici, quelques résultats obtenus du MAC. La Figure 5-11 présente les flots transactionnels couverts.

Grade	L2_sq0...	L2_sq1...	L2_sq1...	L2_sq2...	L2_sq2...	L2_sq2...	L2_sq2...	Te...	Hits	Go...	Hits / Goal
100	RX_P_0_0	RW_P_0_0	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	3	1	
0	RX_P_0_0	RW_P_1_0	FW_P_0_0	–	–	–	–	0	0	1	
0	RX_P_0_0	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	EMPTY	–	0	0	1	
100	RX_P_0_0	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	EMPTY	1	1	1	
100	RX_P_0_0	RW_P_1_0	FW_P_3_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	TX_P_0_3	1	1	1	
100	RX_P_0_1	RW_P_0_0	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	3	1	
0	RX_P_0_1	RW_P_1_0	FW_P_0_0	–	–	–	–	0	0	1	
0	RX_P_0_1	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	EMPTY	–	0	0	1	
100	RX_P_0_1	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	EMPTY	1	1	1	
100	RX_P_0_1	RW_P_1_0	FW_P_3_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	TX_P_0_3	1	1	1	
100	RX_P_0_2	RW_P_0_0	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	3	1	
0	RX_P_0_2	RW_P_1_0	FW_P_0_0	–	–	–	–	0	0	1	
100	RX_P_0_2	RW_P_1_0	FW_P_1_0	EMPTY	TX_P_0_1	EMPTY	TX_P_0_3	1	1	1	
0	RX_P_0_2	RW_P_1_0	FW_P_1_0	EMPTY	TX_P_0_1	TX_P_0_2	–	0	0	1	
0	RX_P_0_2	RW_P_1_0	FW_P_1_0	TX_P_0_0	–	–	–	0	0	1	
0	RX_P_0_2	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	EMPTY	–	0	0	1	
100	RX_P_0_2	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	EMPTY	1	1	1	
100	RX_P_0_3	RW_P_0_0	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	3	1	
0	RX_P_0_3	RW_P_1_0	FW_P_0_0	EMPTY	EMPTY	–	–	0	0	1	
100	RX_P_0_3	RW_P_1_0	FW_P_0_0	EMPTY	TX_P_0_1	EMPTY	EMPTY	1	1	1	
0	RX_P_0_3	RW_P_1_0	FW_P_0_0	TX_P_0_0	–	–	–	0	0	1	
100	RX_P_0_3	RW_P_1_0	FW_P_3_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	TX_P_0_3	1	1	1	

Figure 5-11: Résultats de couverture (flot transactionnel)

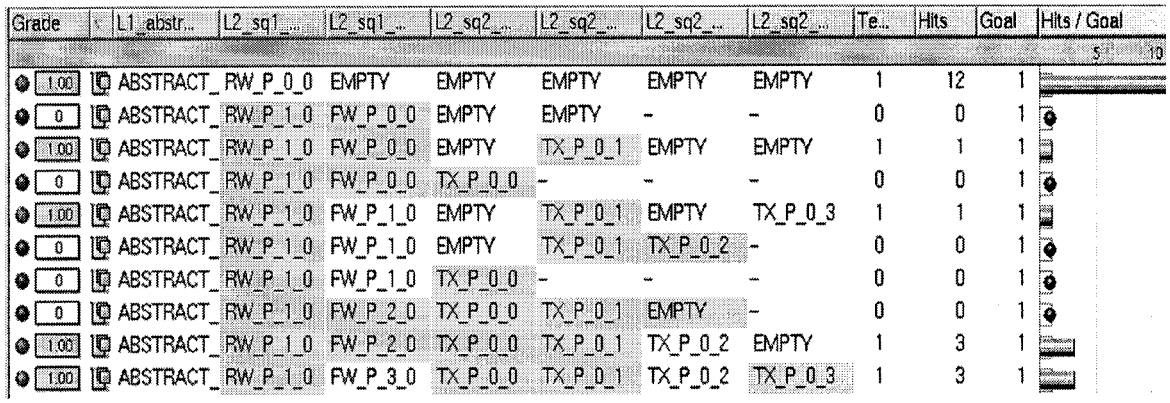
Ainsi, à la Figure 5-11, les chemins couverts dans certains flots transactionnels peuvent être observés. La Figure 5-12 présente la couverture des chemins dans le processus *forward*.



Grade	FW_t0	FW_t1	FW_t2	FW_t3	FW_t4	FW_t5	FW_t6	FW_t7	FW_t8	T...	H...	G...	Hits / Goal
0	FW_IW_0	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	0	0	1	
100	FW_IW_0	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1I_	1	3	1	
100	FW_IW_0	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1I_	EMPTY	EMPTY	1	3	1	
100	FW_IW_0	FW_WIW1	FW_IW1W	FW_WIW1	FW_IW1I_	EMPTY	EMPTY	EMPTY	EMPTY	1	1	1	
100	FW_IW_0	FW_WIW1	FW_IW1I_	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	1	1	

Figure 5-12: Résultats de couverture (chemin d'un processus)

Dans ce cas-ci, c'est la liste des noms des BB constituant les chemins qui est présentée. Finalement, la Figure 5-13 présente la couverture des flots transactionnels avec une abstraction.



Grade	L1_abstr...	L2_sq1_...	L2_sq1_...	L2_sq2_...	L2_sq2_...	L2_sq2_...	L2_sq2_...	Te...	Hits	Goal	Hits / Goal
100	ABSTRACT_	RW_P_0_0	EMPTY	EMPTY	EMPTY	EMPTY	EMPTY	1	12	1	
0	ABSTRACT_	RW_P_1_0	FW_P_0_0	EMPTY	EMPTY	-	-	0	0	1	
100	ABSTRACT_	RW_P_1_0	FW_P_0_0	EMPTY	TX_P_0_1	EMPTY	EMPTY	1	1	1	
0	ABSTRACT_	RW_P_1_0	FW_P_0_0	TX_P_0_0	-	-	-	0	0	1	
100	ABSTRACT_	RW_P_1_0	FW_P_1_0	EMPTY	TX_P_0_1	EMPTY	TX_P_0_3	1	1	1	
0	ABSTRACT_	RW_P_1_0	FW_P_1_0	EMPTY	TX_P_0_1	TX_P_0_2	-	0	0	1	
0	ABSTRACT_	RW_P_1_0	FW_P_1_0	TX_P_0_0	-	-	-	0	0	1	
0	ABSTRACT_	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	EMPTY	-	0	0	1	
100	ABSTRACT_	RW_P_1_0	FW_P_2_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	EMPTY	1	3	1	
100	ABSTRACT_	RW_P_1_0	FW_P_3_0	TX_P_0_0	TX_P_0_1	TX_P_0_2	TX_P_0_3	1	3	1	

Figure 5-13: Résultat de couverture (flot transactionnel avec abstraction)

Dans ce cas-ci, l'élément hiérarchique concurrentiel incluant les processus *utopia_rx* est abstrait. Ceci est illustré à la Figure 5-13, où nous n'avons pas le détail des chemins couverts pour ces processus, mais seulement la valeur spéciale *ABSTRACT*.

5.3 Module PMC-Sierra

La méthode de couverture a été appliquée avec succès sur un modèle industriel fourni par PMC-Sierra. Sans donner de détail sur la nature de ce modèle et de ses fonctionnalités, certains résultats de l'application de la méthode sont présentés.

La spécification exécutable du modèle a été faite en utilisant SDL. Cette description a été produite par un ingénieur de vérification de PMC-Sierra responsable de la vérification de ce module. Donc, en appliquant la méthode, un MAC a été produit. Dans ce cas-ci, l'ensemble des flots transactionnels contient 4030 flots transactionnels. Ensuite, nous

avons utilisé une génération pseudo-aléatoire de transactions comme point de départ à la création d'une suite de tests. Contrairement à l'exemple du commutateur ATM, il n'a pas été possible d'atteindre un taux de couverture de 100% avec seulement une génération pseudo-aléatoire. En fait, le taux de couverture a plafonné à 10%. Donc, les capacités d'abstraction du module d'analyse de la couverture ont été utilisées afin de réduire la complexité de la métrique. En d'autres mots, nous avons concentré la création d'une suite de tests sur une partie du modèle en ne tenant pas compte des parties abstraites. En sachant que le modèle SDL est composé des processus A, B, C et D, le Tableau 5-1 expose le nombre de flots transactionnels en fonction de l'abstraction de différents processus du modèle.

Tableau 5-1: Nombre de flots transactionnels en fonction de l'abstraction

Abstraction	Aucune	A	B	C	A - B	A - C	B - C	A - B- C
Nombre de Flots Transactionnels	4030	1955	2187	1987	1095	1107	897	73

On a travaillé avec l'abstraction A-B-C, qui produit 73 flots transactionnels. Il a été facile de créer une suite de tests permettant d'atteindre un taux de couverture de 100% avec une génération de transactions pseudo-aléatoire dirigée. Ensuite, en partant de cette suite de tests et en éliminant graduellement l'abstraction faite, la suite de tests a pu être augmentée afin de couvrir les éléments qui étaient abstraits. Cependant, il faut dire que cette expérimentation a permis de mettre en évidence que la création manuelle de suites de tests efficaces est très complexe.

Chapitre 6

CONCLUSION

Dans l'introduction de ce mémoire, il a été mentionné que la vérification fonctionnelle est sur le chemin critique du processus de développement de n'importe quel circuit numérique. En conséquence, il est nécessaire de travailler aux développements de méthodes et outils qui permettront de réduire le retard accumulé de la vérification par rapport aux techniques de conception des circuits numériques. La vérification peut être améliorée en travaillant sur l'accélération de la conception de la vérification et sur l'augmentation de la qualité de la vérification. L'objectif du travail présenté dans ce mémoire est d'apporter une contribution au niveau des méthodes utilisées pour la vérification fonctionnelle des circuits numériques. Deux méthodes de vérification ont été présentées dans ce mémoire. La présentation de ces méthodes a été précédée par une revue de littérature sur le sujet de la vérification. Cette revue de littérature a permis d'introduire des définitions et d'exposer plusieurs techniques de vérification actuelle. D'autre part, il a été question des méthodes et outils de conception de banc d'essais qui permettent d'obtenir des bancs d'essais évolués. Les langages de vérification sont de nouveaux outils qui facilitent l'application de méthodes de vérification évoluées. Parmi les méthodes de conception qui permettent la création de bancs d'essais évolués, on retrouve les méthodes qui permettent de générer automatiquement les vecteurs de tests dans le banc d'essais, de vérifier automatiquement les réponses du modèle sous vérification et de fournir une mesure sur l'avancement de la vérification fonctionnelle.

Les méthodes présentées dans ce mémoire sont des contributions originales permettant d'augmenter l'ensemble des méthodes de vérification utilisées pour créer des bancs d'essais évolués. La méthode de conception de banc d'essais axée propose un partitionnement dédié à la création de bancs d'essais en utilisant une des dernières technologies de la conception logicielle : la programmation orientée aspects. Le partitionnement favorise la réutilisation des modules de banc d'essais en retirant tous les

éléments pouvant nuire à la réutilisation dans des aspects. La méthode de couverture fonctionnelle proposée est un pas vers l'application systématique de métriques fonctionnelles sur une spécification exécutable d'un modèle. En fait, la création de modules d'analyse de la couverture de façon systématique en fonction d'un patron de vérification générique.

Trois cas ont servi pour l'évaluation des méthodes de vérification. Le système de conversion de protocoles a servi de véhicule pour l'évaluation de la méthode de conception axée sur la réutilisation. La conclusion est que la méthode est valable puisqu'elle a permis d'accélérer la création des bancs d'essais dédiés au système de conversion de protocoles. Cette affirmation est appuyée par le fait que suite à la création des bancs d'essais unitaires du système, plus de 50% des autres bancs d'essais (intégration et niveau système) sont basés sur la réutilisation directe de modules définis dans les bancs d'essais unitaires. Cette méthode pour la production des bancs d'essais est présentement utilisée pour la vérification de la version la plus récente du système de conversion de protocoles. Cette nouvelle application permettra de retirer d'autres informations relatives à la validité de la méthode. Pour ce qui est de la méthode de couverture fonctionnelle, l'exemple du commutateur ATM a permis de prouver l'applicabilité de la méthode. Cela a permis, à l'aide du MAC produit, de créer une suite de test efficace servant à couvrir la métrique des flots transactionnels. En réduisant la longueur de la suite de tests sans diminuer sa puissance de détecteur d'erreurs, il est possible d'accélérer le temps consacré à la simulation du DUV. Cependant, l'exemple du commutateur ATM étant très simple, cela ne prouve pas l'applicabilité de la méthode sur des modèles de complexité réelle. L'application de la méthode sur le module fourni par PMC-Sierra a permis de démontrer que la méthode est applicable sur des modèles réels. Il est cependant ressorti que même si le MAC fournit une mesure sur l'avancement de la vérification et une indication sur comment optimiser une suite de tests, le processus de création manuel d'une suite de tests peut devenir très complexe. Il faudrait donc travailler à une technique facilitant la création des suites de tests. D'autre part, le mécanisme d'abstraction du MAC permet de réduire la complexité de la métrique. Toutefois, la

question qui peut se poser est de savoir si cette simplification implique que les suites de tests créées avec l'abstraction vont manquer la couverture de certaines fonctionnalités qu'on n'aurait pas dû abstraire. La réponse est sûrement affirmative dans certains cas. Cependant, l'abstraction peut être utilisée dans le cas où ce qu'on abstrait est fonctionnellement indépendant de ce qu'on n'abstrait pas. Supposons, un modèle constitué d'une partie A et d'une partie B. Si A et B sont fonctionnellement indépendants et qu'on considère A comme étant fonctionnellement correct. Donc, si la métrique est appliquée sur un modèle constitué de A et de B, il serait possible d'abstraire A sans possibilité de manquer la couverture de fonctionnalités. Toutefois, le problème à résoudre reste de savoir comment affirmer que A est fonctionnement indépendant de B. Finalement, l'expérimentation de la méthode de couverture sur d'autres modèles permettrait de généraliser son application.

Plusieurs travaux peuvent découler des méthodes présentées dans ce mémoire. Pour la méthode de conception, puisque le but est d'isoler les éléments ne pouvant pas être réutilisés, il serait intéressant de définir des techniques pour assister la création de ce qui n'est pas réutilisable. La méthode de couverture est d'ailleurs une technique permettant d'assister la création d'un module au niveau de la couverture fonctionnelle qui n'est pas réutilisable. Pour ce qui est de la méthode de couverture, il est évident que la métrique présentée dans ce mémoire n'est pas applicable sur tous les types de modèle. Donc, la définition de nouvelles métriques fonctionnelles et l'évaluation de l'applicabilité de ces métriques en fonction du type de modèle seraient des contributions pertinentes. L'évaluation et le raffinement du processus d'abstraction d'une métrique fonctionnelle sont des avenues qui pourraient aussi être approfondies. Cela pourrait mener à une abstraction conditionnelle d'une métrique fonctionnelle en fonction du type de modèle et des dépendances fonctionnelles. D'autre part, la création d'un outil générant les MAC est une continuité naturelle puisque la création des modules est systématique. Finalement, les patrons de vérification définis pour mesurer l'avancement de la couverture pourraient être augmentés afin de faciliter la création des suites de tests en fournissant plus d'information. Présentement, un MAC fournit le taux de couverture actuel ce qui sert à la

création manuelle de suites de tests. Cependant, cela pourrait bénéficier d'une automatisation plus poussée. Une idée serait de développer une approche de spécification pour la couverture. Une spécification exécutable doit être développée pour appliquer la métrique. Donc, si on incluait des informations facilitant la création de suite de tests dans la spécification, un outil pourrait prendre en charge ces informations. En fonction de ces informations et du taux de couverture actuel, il serait envisageable de produire la suite de tests permettant de faire converger la couverture de la métrique. La création automatique de suites de tests fonctionnels permettrait d'avoir une méthodologie de couverture fonctionnelle complète ou en d'autres mots, une solution de vérification dynamique totalement adaptative.

RÉFÉRENCES

[1] ARDITI, L. CLAVÉ, G. 2000. « A Semi-Formal Methodology For The Functional Validation of An Industrial DSP System ». Proceeding of ISCAS2000, IEEE. 4, 205-208.

[2] ATM FORUM. 1995. *UTOPIA Level 2. Version 1.0.* [En ligne]. ATM FORUM TECHNICAL COMMITTE. 70p. af-phy-0039.000. ftp://ftp.atmforum.com/pub/approved-specs/af-phy-0039.000.pdf. (Page consultée le 2 février)

[3] BEIZER, B, 1990. *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold. 550p.

[4] BEIZER, B. 1995. *Black-Box Testing: Techniques for Functional Testing of Softwares and Systems*, Toronto : Wiley. 294p.

[5] BENNING L., FOSTER H., 2001. *Principles of Verifiable RTL Design.* 2nd ed. Boston : Kluwer Academic Publishers. 312p.

[6] BERGERON, J. 2000. Writing Testbenches : Functional Verification of HDL Models, Boston : Kluwer Academic Publishers. 354p.

[7] BERGERON, J. 2001. Verification Guild Project. In. *Janick Bergeron's Home page.* [En ligne]. http://www.janick.bergeron.com/guild/project.html. (Page consultée le 2 février)

[8] BOOCH, G. RUMBAUGH J. JACOBSON, I. 1998. *The Unified Modeling Language User Guide.*, Addison Wesley Professional, 512p.

[9] BRAHME, D.S., COX, S., GALLO, J., GLASSER, M., GRUNDMANN, W., NORRIS IP, C., PAULSEN, W., PIERCE, J.L., ROSE, J., SHEA, D. AND WHITING, KARL. 2000. *The Transaction-Based Verification Methodology.* Berkeley: Cadence Berkeley Labs. 8p. CDNL-TR-2000-0825.

[10] BYUN, Y. BEVERLY, A. S. KEUM, C. 2001. « Design Patterns of Communicating Extended Finite State Machines in SDL ». Proceedings of PLop 2001.

[11] CADENCE DESIGN SYSTEM. 2003. TestBuilder User Guide. In. *Cadence Testbuilder.* [En ligne]. http://www.testbuilder.net. (Page consultée le 2 février).

[12] COHEN, B. 2000. *Component Design by Example : A Step-by-Step Process Using VHDL with UART as Vehicle.* VhdlCohen Publishing, 284p

[13] CZARNECKI K. EISENECKER, U. 2000. *Generative Programming: Methods, Techniques, and Applications.* Addison Wesley Professional. 864p.

[14] DIJKSTRA, E.W. 1976. *A Discipline of Programming.* Prentice Hall, 217p.

[15] DOLDI, L. 2001. *SDL Illustrated - Visually design executable models.* Publié par Laurent Doldi. 270p.

[16] GAMMA E., HELM R., JOHNSON R., VLISSIDES J. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley. 416p.

[17] GRÖTKER, T., LIAO, S., MARTIN, G. AND SWAN, S. 2002. System Design with SystemC. Kluwer Academic Publishers. 240p.

[18] HALFHILL, T. R.. 1995. The Truth Behind the Pentium Bug. In. Byte.com. [En ligne]. http://www.byte.com/art/9503/sec13/art1.htm. (Page consultée le 2 février)

[19] HAQUE, F. KHAN. MICHELSON, J. 2001. *The Art of Verification with VERA*, Verification Central, 452p.

[20] HOWDEN, W.E. 1987. *Functional Program Testing & Analysis.* McGraw-Hill, 171p.

[21] INTERNATIONAL TELECOMMUNICATION UNION. 1999. *Specification and description language (SDL).* [En ligne] Telecommunication Standardization Sector of ITU. 246p. ITU-T Recommendation Z.100. http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf . (Page consultée le 2 février)

[22] JAMES, P. 2000. The Five-Day Verification Plan. In. *The Qualis Library.* [En ligne]. http://www.qualis.com/cgi-bin/qualis/library.pl. (Page consultée le 2 février)

[23] Kiczales, G. Lamping, J. Mendhekar, A. Maeda, C. Lopes, C.V. Loingtier J. Irwin, J. 1997. « Aspect-Oriented Programming ». Proceeding of ECOOP97, 220-242.

[24] MANNA, Z. WALDINGER, R. 1978. « The Logic Of Computer Programming». IEEE Transaction on Software Engineering, 4:3. 199-229.

[25] MOUNDANOS, D.J. ABRAHAM, A. HOSKOTE, Y.V. 1998 « Abstraction Techniques for Validation Coverage Analysis and Test Generation ». IEEE Transactions on Computers. 47 :1. 2-14.

[26] MYERS, G.J. 1979. *The Art of Software Testing.* Wiley, John & Sons. 177p.

[27] PRESSMAN R.S. 1996. *Software Engineering: A Practitioner's Approach.* 4th ed. McGraw-Hill, 852p.

[28] RASHINKAR, P. PATERSON, P. SINGN, L. 2001. *System-on-a-chip Verification, Methodology and Techniques.* Boston : Kluwer Academic Publishers. 372p.

[29] REGIMBAL, S., LEMIRE, J.F., SAVARIA, Y., BOIS, G., ABOULHAMID, E.M. AND BARON, A. 2002. « Aspect partitioning for Hardware Verification Reuse». *2002 International Workshop on System-On-Chip.* 49-58.

[30] REGIMBAL, S., LEMIRE, J.F., SAVARIA, Y., BOIS, G., ABOULHAMID, E.M. AND BARON, A. 2002. « Applying Aspect-Oriented Programming to Hardware Verification with *e*». *The 2002 International HDL Conference.* 68-75.

[31] SHIMIZU, K. DILL, D. L. 2002. « Deriving Input Generator and a Coverage Metric From Formal Specification ». *Proceedings of the 39th conference on Design automation.* ACM, 801-806.

[32] SUZUKI, JUNICHI. YAMAMOTO, YOSHIKAZU. 1999. «Extending UML for Modelling Reflective Software Components ». *<<UML>>'99 - The Unified Modeling Language: Beyond the Standard*, Springer LNCS 1723, 220-235.

[33] SYNOPSYS. 2001. Language Reference Manuel, version 1.0. In. *Open-Vera Website*. [En ligne]. http://www.open-vera.com. (Page consultée le 2 février)

[34] TASIRAN, S. KEUTZER. K. 2001. « Coverage Metrics for Functional Validation of Hardware Designs », IEEE Design & Test of Computers. 18 :4. 36-45.

[35] TELELOGIC. 2003. Telelogic product – Telelogic Tau - Overview. In. Telelogic's Home page. [En ligne]. http://www.telelogic.com/products/tau. (Page consultée le 2 février)

[36] VERISITY. 2003. Verisity's On-Line Verification Portal. In. *Verification Vault.* [En ligne]. https://www.verificationvault.com. (Page consultée le 2 février)

[37] VSI ALLIANCE. 2001. *Taxonomy of Functional Verification for Virtual Component Development and Integration Standard, Version 1.1.2.* [En ligne]. [Los Gatos, CA] :Functional Verification Development Working Group. 38p. http://www.vsi.org/library/specs/ver111.pdf. (Page consultée le 2 février)

[38] WEINBERG, G.M.. 1998. *The Psychology of Computer Programming*. New York: Dorset House Publishing, 360p.

[39] WHITTEMORE, P. DEARTH, G. 1999. « Object-Oriented Approach to Verification ». *In Proceedings of 1999 SNUG*. Section FC-1.

[40] WILSON, RON. 2002. Solutions proposed for verification crisis, In. *EEDesign.* [En ligne]. http://www.eedesign.com/story/OEG20020930S0054. (Page consultée le 2 février)

[41] XANTHAKIS, S. RÉGNIER, P. KARAPOULIOS, C. 2000. *Le test des logiciels.* Paris : Éditions Hermes Science. 328p.

[42] ZHANG, Q. IAN, H. 2000. « Mutation Analysis for the Evaluation of Functional Fault Models ». *Proceeding of the HLDVT99.*

ANNEXES

Annexe A

ÉLÉMENTS SPÉCIAUX DU LANGAGE *e*

Pour que l'on puisse qualifier un langage comme étant un langage de vérification, cela dépend seulement de quelques caractéristiques spéciales. Ces caractéristiques sont ce qui permet de faciliter la tâche de la vérification. Dans cette annexe, il est question du langage *e* qui est dédié à la vérification. Il est à noter que les éléments présentés dans ce document se trouvent aussi dans les autres langages de vérification tel que *OpenVera* et *Testbuilder*.

Ce document contient un survol de trois sujets permettant l'élaboration de banc d'essai avec le langage *e* :

- Une brève introduction aux termes et aux éléments de bases que l'on retrouve dans ce langage;
- Les contraintes de génération et le générateur pseudo-aléatoire;
- La couverture fonctionnelle avec Specman Elite™.

A.1 Base du langage e

Un banc d'essai en *e* est construit avec des classes. Les mots utilisés pour représenter une classe sont *struct* et *unit*. *unit* est habituellement utilisé pour modéliser les éléments statiques d'un banc d'essai, et pour ce qui est dynamique le mot *struct* est utilisé. Cependant dans le cadre de cette discussion, cela n'a pas d'importance et nous nous contenterons de parler de *struct.* Ainsi, un programme en *e* est élaboré avec une hiérarchie de *structs*. Le tableau suivant présente les éléments que l'on peut retrouver dans les *structs*.

Tableau A-1 : Éléments contenus dans une *struct* en e

Nom	Description
Champ de donnée	Les champs de donnée permettent de définir les données dans les *structs*. Les champs de données sont par défaut pseudo-aléatoire et

	doivent donc être générés. Ils sont équivalents aux attributs membres d'une classe en C++.
Contrainte de génération	Les contraintes de génération viennent biaiser les valeurs aléatoires qui sont affectées sur les champs de donnée d'une *struct*.
Éléments temporels	Parmi les éléments temporels, l'on retrouve les définitions des événements et les définitions des actions se produisant sur ces événements. De plus, on retrouve les assertions qui sont construites à partir des événements.
Méthodes et TCM (*Time Consuming Method*)	Les méthodes sont l'équivalent des fonctions membres en C++. Néanmoins, il y a deux types de méthode en *e* : les méthodes et les TCMs. La différence se situe au niveau de l'aspect temporel. Une méthode est définie comme une séquence d'opérations qui se passe dans un temps instantané. Pour ce qui est d'une TCM, ce type de méthode est synchronisé sur un événement et plusieurs *threads* peuvent être engendrés dans ce type de méthode. On peut ainsi dire qu'une TCM consomme du temps
Groupe de couverture	Les groupes de couverture permettent de définir un modèle de couverture. Ainsi, un groupe de couverture contient des items qui seront échantillonné sur un certain événement prédéfini. La combinaison des items définis dans un groupe de couverture sert de métrique fonctionnelle à la vérification. On peut donc mesurer l'avancement de la vérification en basant sur les résultats de l'échantillonnage obtenu avec différents groupes de couverture.
Directives au Préprocesseur	Directives permettant une compilation conditionnelle.

A.2 Générateur pseudo-aléatoire

Initialement, tous les champs de données qu'on peut retrouver dans un programme en *e* sont aléatoires. Avant de procéder à l'exécution du programme, on doit donc faire appel au générateur pseudo-aléatoire de Specman Elite™ afin d'effectuer la génération de chaque champ de donnée du banc d'essai. De plus, cette génération peut être biaisée afin de réduire l'ensemble des possibilités qui peuvent être affectées à un champ de donnée ou encore complètement forcer une valeur sur un champ de donnée. La technique utilisée afin de biaiser un champ de donnée est de définir des contraintes de génération sur les champs de donnée. Donc, le générateur pseudo-aléatoire doit pouvoir déterminer une valeur pour chaque champ de donnée en tentant de satisfaire toutes les contraintes de génération formulées.

Dans cette section, il sera question des contraintes de génération pouvant s'appliquer aux champs de donnée. Ensuite, un bref aperçu du fonctionnement du générateur pseudo-aléatoire sera présenté.

A.2.1 Contraintes de génération

Lorsqu'un champ de donnée est défini dans un programme *e*, la valeur de ce champ sera soit générée par le générateur pseudo-aléatoire ou elle ne sera pas générée en le spécifiant avec l'opérateur !*(do not generate)*. Ainsi, lorsque l'on spécifie de ne pas générer de valeur pour un champ spécifique, le générateur affecte la valeur par défaut qui est définit selon le type du champ. Par exemple, si le champ est de type *uint* ou *int*, la valeur par défaut sera 0 ou encore si le champ de donnée est un vecteur binaire quelconque, il sera initié à 0. Par contre, lorsque le champ de donnée doit être généré, une valeur est choisie dans l'ensemble des possibilités du champ de donnée. À la base, cet ensemble des possibilités est encore défini par le type du champ de donnée. Cet ensemble des possibilités peut être ensuite biaisé avec l'aide de contraintes de génération. Finalement, le générateur choisit une valeur pour le champ de donnée, à la base selon une distribution uniforme, dans l'ensemble des possibilités restantes. Voici un tableau comprenant différente déclaration de champ de donnée et une valeur pouvant être générée.

Tableau A-2 : Déclarations de champ de donnée

Déclaration		Ensemble des possibilités	Exemple d'une valeur
A	`:uint;`	[0..4294967295]	239374
B	`:bit;`	{0,1}	1
!C	`:uint;`	[0] (valeur par défaut)	0 (valeur non générée)
D	`:packet_type;`	{ATM, IPv4, IPv6}	Ipv6

Maintenant, pour ce qui est des contraintes de génération permettant de biaiser l'ensemble des possibilités d'un champ de donnée, nous devons utiliser le mot réservé *keep*. Le mot *keep* permet de définir explicitement une contrainte de génération qui s'appliquera sur un ou plusieurs champs de donnée. Le tableau suivant présente les opérations pouvant être utilisées dans la définition d'une contrainte de génération.

Tableau A-3: Opération pouvant être effectuée dans une contrainte

Opérateur	Opération	Exemple
==	equal	**keep** x == y
<	less than	**keep** x < y

>	greater than	**keep** x > y
<=	less than or equal	**keep** x <= y
>=	greater than or equal	**keep** x >= y
=>	implication	**keep** x => y
or	Or	**keep** x == b1 or y == b2
and	And	**keep** x == b1 and y == b2
meth()	Method call	**keep** x == meth()
+, -, *, /,%	arithmetic operations	**keep** z == x + y, …
in	range constraint	**keep** x in list **keep** x in {1,3,5} **keep** x in [0..100]

Il est permis de définir plusieurs contraintes sur un même champ de donnée. Par contre, la définition de plusieurs contraintes contradictoires, entraînant l'ensemble vide pour l'ensemble des possibilités pour un certain champ de donnée, génère une erreur de la part du générateur. Il est donc important de maintenir une certaine logique lors de la définition des contraintes de génération. Pour cette raison, le générateur permet la définition de contrainte suggestive, c'est-à-dire que nous pouvons définir des contraintes que le générateur n'est pas tenu de respecter. S'il peut respecter une contrainte suggestive, il le fera, mais dans le cas contraire, la contrainte sera ignorée. Ce type de contrainte est défini en ajoutant le mot réservé *soft* après le mot keep. Ce qui permet d'introduire le concept inverse qui de contrainte *hard*. Une contrainte *hard* est une contrainte que le générateur est tenu de respecter. Ainsi, par défaut une contrainte est *hard*, et cela, sans qu'on soit obligé de le spécifier. Les exemples suivants présentent quelques cas d'application des contraintes sur des champs de donnée.

Tableau A-4 : Exemples de contraintes appliquées sur l'ensemble des possibilités

Déclaration	Ensemble des possibilités	Exemple d'une valeur
A : **uint** ; **keep** A <= 100; **keep soft** A >= 50; **keep soft** A <= 25;	[0..4294967295] [0..100] [50..100] [50..100] 67	Ensemble de base de A Réduction de l'ensemble de A Réduction de l'ensemble de A Contrainte soft contradictoire ignorée Valeur générée de A
A :**uint**; B :**uint**; **keep** A **in** [50..150]; **keep** B == A + 5;	[0..4294967295] [0..4294967295] [50..150] [55..155] 79 [84]	Ensemble de base de A Ensemble de base de B Réduction de l'ensemble de A Réduction de l'ensemble de B Valeur générée pour A Réduction implicite de l'ensemble de B

	84	Valeur générée (FORCÉ) pour B

A :uint;
keep A <= 100;
keep A >= 200;
	[0..4294967295]
	[0..100]
	[]
		Ensemble de base de A
		Réduction de l'ensemble de A
		Réduction de l'ensemble de A
		L'ensemble est vide
		(Erreur de génération)

Dans le premier exemple, il est question d'une contrainte *soft* qui est ignorée, car elle entre en contradiction avec une autre contrainte. Ensuite, le second exemple expose le concept de réduction implicite par une contrainte impliquant deux champs de donnée. Dans ce cas-ci, la valeur de *A* a été générée avant *B*, ce qui a influencé l'ensemble de *B* et forcé celle-ci à 84. Dans le cas où la valeur de *B* aurait été générée avant *A*, la réduction implicite aurait été effectuée sur l'ensemble des possibilités de A. Ainsi, on appelle ce type de contrainte une contrainte bidirectionnelle, car l'ordre de génération des champs impliqué dans la contrainte n'importe peu. Un exemple de contrainte unidirectionnelle sera présenté ultérieurement. Le dernier exemple présente des contraintes de génération (*hard*) contradictoire qui résulte en une erreur de génération.

Il existe trois autres mécanismes pouvant être utilisés pour définir une contrainte de génération. Ces trois mécanismes servent à effectuer les tâches suivantes :

- Influencer la distribution de la génération pseudo-aléatoire;
- Affecter des contraintes de génération sur les éléments d'une liste;
- Influencer l'ordre de génération des champs de donnée[1];

Le tableau suivant présente ces mécanismes.

Tableau A-5 : Mécanismes permettant de définir d'autres contraintes de génération

Opération	Opérateur	Exemple

Contrainte de contrôle de la distribution	keep soft expression == select
	{
	weight_1: values_1;
	...
	weight_n: values_n;
	};
		A : uint;
		keep A <= 100;
		keep soft A == select {
		1 : [0..10];
		5 : [11.99];
		10 : 100;
		};

Contraintes sur une liste	keep for each [...] in list {
	...
	};
		A_list : list of uint;
		keep A_list.size() == 10;
		keep for each (element_A) using

[1] Le générateur pseudo-aléatoire tient compte des contraintes de génération dans un ordre particulier qui sera discuté dans la prochaine section.

		index (iteration) **in** A_list {
		  element_A == iteration;
		};

Contrainte d'ordonnancement de la génération	**keep** [soft] **gen** (y) **before** (x);	A :uint; B :uint; C :uint; **keep gen** (C) **before** (B); **keep** B == A + C; **keep** A **in** [50..150]; **keep** C **in** [20..40];

Le premier exemple permet de contrôler la distribution des valeurs pseudo-aléatoires générées. Par défaut, le générateur utilise la distribution uniforme dans l'ensemble des possibilités d'un champ de donnée. Pour ce type de contrainte, nous devons définir un poids pour chaque plage de valeurs qui peut être affecté à A. Dans cet exemple, le total des poids est 16 et donc :

- 6.25% des valeurs de A seront entre 0 et 10 (1/16);
- 31.25% des valeurs de A seront entre 11 et 99 (5/16);
- 62.5% des valeurs de A seront 100 (10/16);

Le second exemple permet de définir une contrainte de génération sur les éléments d'une liste de données. Dans ce cas-ci, on applique une première contrainte pour spécifier la taille de la liste : il y aura 10 éléments dans la liste. Ensuite, pour tous les éléments de la liste, on affecte la valeur de l'index de la boucle itérative (1 à 10). Finalement, le troisième exemple expose comment imposer l'ordre de génération des contraintes. L'exemple présenté ici est le même que le celui présenté dans le Tableau A-4, où il était question d'une contrainte bidirectionnelle. Ainsi en ajoutant cette contrainte sur l'ordre de génération, nous nous assurons que la valeur de B sera générée avant la valeur A. Cela implique que la réduction implicite de l'ensemble des possibilités s'effectuera sur l'ensemble de A plutôt que sur l'ensemble de B.

A.2.2 Fonctionnement du générateur pseudo-aléatoire

Le générateur pseudo-aléatoire peut être biaisé par des contraintes de génération. Ainsi, l'algorithme de satisfaction de contraintes utilisée par le générateur de Specman Elite™ doit suivre certaines règles s'il veut être en mesure de satisfaire toutes les

contraintes de génération. L'algorithme n'est pas explicitement détaillé dans la documentation, mais tout de même certaines règles de base sont spécifiées afin de permettre une utilisation efficace du générateur.

Premièrement, le générateur utilise quatre niveaux pour l'ordonnancement des contraintes de génération. On parle ici de l'ordre dans lesquels seront évaluées les contraintes pour un champ de données et aussi dans quel ordre les champs de donnée seront générés. Voici un tableau présentant ces différents niveaux et ce qu'ils impliquent.

Tableau A-6 : Niveaux d'ordonnancement des contraintes

Ordre	Implication
Principal	L'ordre principal est la base de la génération. Il indique le point de départ de la génération. La génération commence par le premier champ de donnée que l'on retrouve dans la *struct* au plus haut niveau de la hiérarchie de *structs*. Ensuite, l'ordre principal implique que lorsque le générateur rencontre un champ de donnée composite, comme une liste ou une *struct*, ce champ devra être complètement généré, de façon récursive, avant de passer à la génération d'un autre champ. Cet ordre est toujours satisfait, sinon le générateur produit une erreur.
Hard	L'ordre *hard* permet d'appliquer les contraintes *hard* sur les champs de donnée. Cet ordre est toujours satisfait, sinon le générateur produit une erreur.
Soft	L'ordre *soft* permet d'appliquer les contraintes suggestives (*soft*) sur les champs de données. Cet ordre est satisfait que si seulement c'est possible.
Par défaut	Cet ordre est utilisé lorsque aucune contrainte n'est affectée à un champ de donnée. Ainsi, puisqu'il n'y pas de contrainte à satisfaire, le champ de donnée est généré par défaut par son ordre d'apparition dans le code *e*. Cet ordre est satisfait que si seulement c'est possible.

Le générateur produit des erreurs lorsqu'on fixe des contraintes contradictoires dans les ordres de génération devant être satisfaits. Il existe d'autres cas où le générateur produit une erreur. Par exemple, lorsqu'un champ de donnée, qu'on a spécifié *do not generate*, est impliqué dans une contrainte de génération. Un autre cas de problème de génération est celui du cycle récursif.

```
<'
struct temperature {
   celsius    : int;
   fahrenheit : int;

   keep celsius    == f_to_c(fahrenheit);
   keep fahrenheit == c_to_f(celsius);
   keep celsius    in [-50..50];

   f_to_c(f :int):int is {
```

```
      result = ((f-32)*100)/(212-32);
   };

   c_to_f(c :int):int is {
      result = ((c*(212-32))/100) + 32;
   };

};
'>
```

Figure A-1: Problème de récursivité

Dans cet exemple, le problème vient du fait que nous avons deux contraintes unidirectionnelles *hard*, c'est-à-dire qu'il y a une contrainte *hard* sur le champ de donnée *celsius* qui exige implicitement que le champ de donnée *fahrenheit* soit généré avant que *celsius* puisse être généré. L'inverse est appliqué aussi sur *fahrenheit*. Dans ce cas-ci, le générateur produit donc une erreur, car il n'est pas capable de résoudre le problème.

A.3 La couverture fonctionnelle avec Specman Elite™

Les mécanismes permettant d'effectuer une couverture fonctionnelle avec Specman Elite™ fournissent les éléments nécessaires à la création de modèles de couverture fonctionnelle. Ces modèles de couverture fonctionnelle servent à assurer qu'une fonctionnalité d'un design a été couverte. Si nous nous situons à un autre niveau d'abstraction, le modèle des fautes collées sert à assurer qu'un design, au niveau structurel, ne possède pas ce type de défaut : les collages. Similairement au niveau fonctionnel, un modèle de couverture fonctionnelle peut permettre d'assurer, par exemple, qu'un processeur produise bien les interruptions spécifiées lorsqu'on lui impose une certaine séquence d'instructions. Ces modèles de couverture sont définis par l'ingénieur de vérification et ils servent de métrique fonctionnelle adaptée à la vérification d'un modèle particulier. Néanmoins, à la différence des modèles de couverture conventionnelle, la qualité d'un modèle de couverture fonctionnelle dépend beaucoup de la méthode de conception utilisée afin de réaliser ce modèle.

Dans cette section, nous présenterons les groupes de couverture qui permettent de définir des modèles de couverture. Ensuite, le mécanisme de pondération permettant d'avoir une mesure quantitative sur l'avancement de la vérification sera présenté.

Finalement, l'API (Application Programmation Interface) permettant d'effectuer des extensions sur les mécanismes de couverture de Specman Elite™ sera introduite.

A.3.1 Groupe de couverture

Un groupe de couverture est un élément d'une *struct* au même titre qu'un champ de donnée ou qu'une contrainte. Un groupe de couverture sert à échantillonner des données lors d'une simulation. Ces données peuvent provenir de 2 sources :

- Les signaux échantillonnés sur le DUV (interne ou externe);
- Des champs de donnée de la *struct* auxquels le groupe de couverture appartient;

L'échantillonnage des groupes de couverture s'effectue sur un événement qui doit être défini dans la *struct*. La Figure A-2 présente un exemple afin d'illustrer un groupe de couverture.

```
<'
struct instruction {
   opcode: [ADD, SUB, AND, XOR];
   operand1 : byte;
   operand2 : byte;

   event stimulus is @sys.new_op;

   cover stimulus is {
      item opcode;
      item operand1 using
         ranges = {
                     range([0..15]);
                     range([16..0xff]);
                   };
      item operand2 using
         ranges = {
                     range([0..15]);
                     range([16..0xff]);
                   };
   transition opcode;
   cross opcode, operand1;
   };
};
'>
```

Figure A-2 : Exemple 1 d'un groupe de couverture

Dans cet exemple, il est question d'une *struct* modélisant une instruction. Dans le cadre d'un test, plusieurs instanciations de cette instruction seront générées par le générateur pseudo-aléatoire. L'instruction comporte un code OP et aussi deux opérandes. De plus, un groupe de couverture est défini et il échantillonné à chaque occurrence de l'événement *stimulus*. Cet événement sera produit à chaque nouvelle injection de

l'instruction dans le DUV. Dans le groupe de couverture, nous retrouvons plusieurs items. Les items sont des pointeurs sur les données qui seront échantillonnées. Par exemple, nous avons un item *opcode* qui échantillonnera les valeurs du champ de donnée *opcode*.

Afin de pouvoir compter le nombre d'occurrence de chaque valeur d'un item, Specman Elite™ a le concept de *bucket*. Un item est composé de plusieurs *buckets*. Les *buckets* définissent une valeur ou une plage de valeurs que peut prendre un item. Ainsi, l'ensemble des buckets d'un item doit couvrir toutes les valeurs possibles d'un item. Dans le cas de l'item *opcode*, il y a création de quatre buckets car cet item peut prendre quatre valeurs : *ADD*, *SUB*, *AND* et *OR*.

Si nous voulons effectuer l'échantillonnage des valeurs d'un signal interne de 1 bit, il aurait création, dans ce cas, de 2 buckets. Le premier bucket servira à compter le nombre de fois que l'item sera échantillonné à '1' et l'autre bucket servira à compter le nombre de fois où le signal sera échantillonné à '0'. Dans un autre cas, on peut avoir un nombre entier (32 bits) comme item à échantillonner. Dans ce cas-ci, $(2^{32}-1)$ buckets serait nécessaires afin d'avoir un bucket pour chaque valeur possible du vecteur ce qui est impossible à cause des limitations en mémoire. Conséquemment, par défaut, Specman Elite™ partitionne l'étendue des valeurs possibles de l'item par plage de valeurs avec 16 buckets. Néanmoins, il est possible de spécifier le partitionnement des plages de valeur et le nombre de buckets désirés. De plus, il est possible de spécifier des buckets où les valeurs ou les plages de valeurs sont illégales ou encore à ignorer. En retournant à l'exemple 1, l'item *operand1* utilisera donc 2 buckets étant donné que nous l'avons spécifié. Le premier bucket récupérera les valeurs de *operand1* plus petite que 15 et le reste des valeurs seront échantillonnées dans l'autre bucket.

L'exemple 1 illustre encore deux autres éléments dans le groupe de couverture : les items *transition* et les items *cross*. Ces deux éléments sont en fait deux autres types d'item : des composites des items de base. Le Tableau A-7 présente les trois types d'item :

Tableau A-7 : Les 3 types d'items d'un groupe de couverture

Item	Utilités
Base	Ils servent à échantillonner les champs de données des *struct* ou des signaux provenant du DUV.
Transition	Ils servent à échantillonner les transitions sur un item de base. Ce type d'item est spécifié en fonction d'un item de base.
Cross	Ils servent à échantillonner le croissement en deux ou plusieurs items de base. Ce type d'item est spécifié en fonction de deux ou plusieurs items de base.

L'exemple 1 présente un cas pour chacun de ces items composites. De la même façon qu'un item de base, les items composés doivent être partitionnés en buckets. Dans le cas de l'item *transition* sur l'item de base *opcode*, nous pouvons facilement déterminer le nombre de buckets engendrés par cet item. En sachant que l'item *opcode* nécessite 4 buckets (*ADD*, *SUB*, *AND* et *OR*), l'item *opcode* peut donc être dans 4 états à un échantillonnage au temps *i*. Ainsi à un échantillonnage au temps (i+1), l'item *opcode* peut encore être dans 4 états différents, donc il est nécessaire d'avoir 16 buckets afin d'avoir un bucket alloué à toutes les transitions possibles sur l'item *opcode* (ADD→SUB, ADD→ADD, AND→OR, ...). Dans le même ordre d'idée, l'item *cross* entre l'item de base *opcode* (4 buckets) et l'item de base *operand1* (2 buckets) nécessite 8 buckets.

En résumé, un modèle de couverture avec Specman Elite™ est composé d'un ensemble de groupe de couverture. Ces groupes de couverture sont composés d'un ensemble d'items et les items sont composés de buckets. Donc, en ayant un modèle de couverture fonctionnelle défini, nous avons accès aux informations de couverture recueillies suite à un test. Ces informations sont accessibles dans un fichier texte ou elles peuvent être visualisées via une interface graphique tel que présentée dans la Figure A-3.

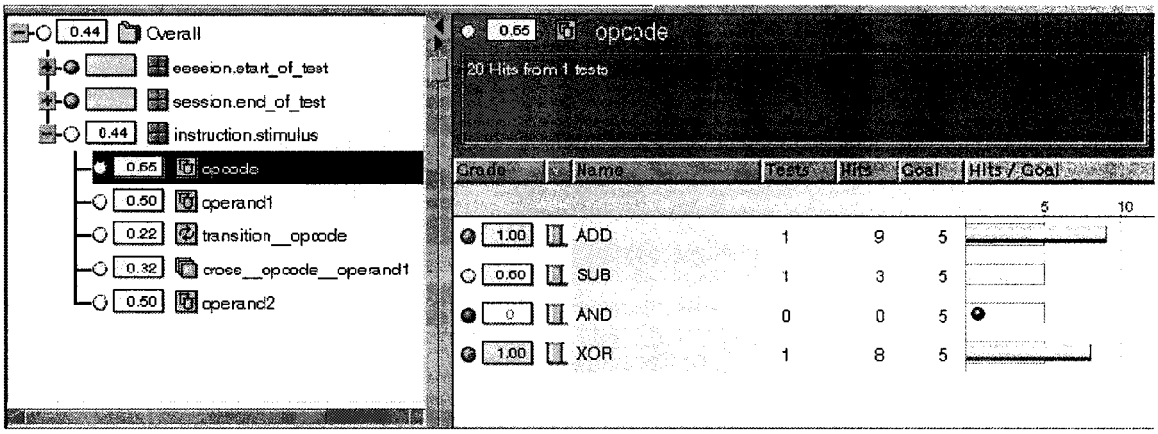


Figure A-3 : Résultats d'un test

Nous pouvons voir à la gauche de la Figure A-3, le groupe de couverture *instruction.stimulus* et les items compris dans ce groupe. Dans la partie de droite de la figure, nous avons les quatre buckets créés pour l'item de base *opcode*.

A.3.2 Mesure de la couverture

Le fait de seulement définir un modèle de couverture avec un ensemble de groupe de couverture ne permet pas d'avoir une mesure quantitative de l'avancement de la vérification. La solution proposée par Specman Elite™ est de pondérer tous les éléments du modèle de couverture afin d'obtenir une métrique fonctionnelle quantitative. La marche à suivre afin de calculer la pondération d'un modèle de couverture est de descendre à plus bas niveau, le bucket, et de remonter jusqu'au plus haut niveau, le modèle de couverture. Afin de réaliser la pondération, il est nécessaire de fixer un poids pour tous les éléments du modèle de couverture. Les poids des éléments de couvertures sont définis explicitement où la valeur par défaut est utilisée (un poids de 1). De plus, au plus bas niveau du modèle de couverture (le bucket), il est possible de fixer un but pour chaque bucket. Le but d'un bucket est le nombre d'échantillons devant être échantillonné avant que la valeur ou la plage de valeurs définies dans un bucket soit couverte. Par défaut, le but de tous bucket est d'atteindre 1 échantillon. Voici l'exemple 2 qui est en fait une révision de l'exemple 1, afin d'incorporer la notion de pondération :

```
<'
struct instruction {
   opcode: [ADD, SUB, AND, XOR];
   operand1 : byte;
   operand2 : byte;

   event stimulus is @sys.new_op;

   cover stimulus using weight = 3 is {
      item opcode using weight = 2;
      item operand1 using
         ranges = {
                    range([0..15]);
                    range([16..0xff]);
                  }, at_least = 10;
      item operand2 using
         ranges = {
                    range([0..15]);
                    range([16..0xff]);
                  };
      transition opcode using weight = 5;
      cross opcode, operand1 using weight = 10;
   };
};
'>
```

Figure A-4 : Exemple 2 d'un groupe de couverture

Dans cet exemple, le poids du groupe de couverture a été fixé à 3 et certains items ont des poids différents. Il est à noter que lorsqu'il n'y pas de poids spécifié, la valeur par défaut est utilisée. De plus, dans le cas de l'item *operand1*, un minimum de 10 échantillons a été fixé comme but pour les buckets de cet item.

Ensuite, les calculs de la pondération du modèle de couverture s'effectuent avec les formules de pondération de la couverture. Il y a deux types de pondération disponibles afin d'effectuer ces calculs : une pondération linéaire et une pondération *Root Mean Square*. Le Tableau A-8 présente les deux séries de formules utilisées dans le calcul de la pondération.

Tableau A-8: Formules pour le calcul de la pondération

Niveau	Formule utilisée Linéaire	Root Mean Square
Bucket	$grade(bucket)=\min\left(1.00,\frac{samples(bucket)}{goal(bucket)}\right)$	$grade(bucket)=\min\left(1.00,\sqrt{\frac{samples(bucket)}{goal(bucket)}}\right)$
Item	$grade(item)=\frac{\sum(grade(bucket)\times weight(bucket))}{\sum(weight(bucket))}$	$grade(item)=\sqrt{\frac{\sum(grade(bucket)^2\times weight(bucket))}{\sum(weight(bucket))}}$
Groupe de couverture	$grade(group)=\frac{\sum(grade(item)\times weight(item))}{\sum(weight(item))}$	$grade(group)=\sqrt{\frac{\sum(grade(item)^2\times weight(item))}{\sum(weight(item))}}$

Modèle de couverture	$grade(model)=\frac{\sum(grade(group)\times weight(group))}{\sum(weight(group))}$	$grade(model)=\sqrt{\frac{\sum(grade(group)^2\times weight(group))}{\sum(weight(group))}}$

Le calcul de la pondération du modèle de couverture, qui est la métrique fonctionnelle proposée par Specman Elite™, s'effectue en commençant par le calcul de la pondération des buckets. Ensuite, la pondération des items est calculée avec l'ensemble des pondérations des buckets, et ainsi de suite jusqu'à l'obtention de la pondération du modèle de couverture.

A.3.3 API de couverture

Une interface de programmation permet d'effectuer certaine opération sur les éléments de couverture de Specman Elite™. En fait, nous avons accès à cinq méthodes permettant d'avoir accès aux informations de couverture recueillies lors d'une simulation. Une courte description de ces méthodes est présentée dans le tableau suivant :

Tableau A-9 : Description de l'API de couverture fournit par Specman Elite™

Nom	Description
scan_cover()	Cette méthode sert à effectuer l'analyse complète de la couverture. En fait, cette méthode fait une tournée des groupes de couverture, items et buckets du modèle de couverture. L'information recueillie est la couverture actualisée au moment de l'appel de la méthode.
start_group()	Cette méthode est appelée par la méthode *scan_cover()* au début de l'observation d'un groupe de couverture. Cette méthode est appelée pour chaque groupe de couverture du modèle de couverture. Elle donne accès aux informations relatives au groupe de couverture courant.
start_item()	Cette méthode est appelée par la méthode *scan_cover()* au début de l'observation d'un item. Cette méthode est appelée pour chaque item du modèle de couverture. Elle donne accès aux informations relatives à l'item et au groupe de couverture courant.
scan_bucket()	Cette méthode est appelée par la méthode *scan_cover()* lors de l'observation de chaque buckets de l'item courant. Elle donne accès aux informations relatives au bucket, de l'item et au groupe de couverture courant.
end_item()	Cette méthode est appelée par la méthode *scan_cover()* à la fin de l'observation de tous les buckets d'un item. Cette méthode est appelée pour chaque item du modèle de couverture. Elle donne accès aux informations relatives à l'item et au groupe de couverture courant.
end_group()	Cette méthode est appelée par la méthode *scan_cover()* à la fin de l'observation de tous les items d'un groupe de couverture. Cette méthode est appelée pour chaque groupe de couverture du modèle de couverture. Elle donne accès aux informations relatives au groupe de couverture courant.

Le principe de l'utilisation de l'API de couvertures est que lors de l'appel de la méthode *scan_cover()*, une tournée itérative de tous les éléments du modèle de couverture est effectuée. La Figure A-5 ci-dessous présente l'algorithme de base utilisé par la méthode *scan_cover()*.

```
scan_cover()
  For all Groupe de couverture
        start_group()
        For all Item
                start_item()
                For all Bucket
                        scan_bucket()
                End For
                end_item()
        End For
        end_group()
  End For
```

Figure A-5 : Algorithme de la méthode *scan_cover()*

Le but de cet algorithme est de procurer une visibilité complète sur les informations de couverture. Cela est possible dû aux méthodes de l'API insérées dans l'algorithme de *scan_cover()*. Les méthodes de l'API, à l'exception de *scan_cover()*, ne peuvent pas être explicitement appelées, car elles le sont automatiquement par la méthode *scan_cover()*. De plus, ces méthodes sont vides à la base. Il faut donc les spécifier afin de préciser les éléments d'intérêt à observer.

Cette interface de programmation peut être utilisée afin de réaliser plusieurs tâches. Premièrement, elle peut être utile afin de réaliser des rapports personnalisés sur l'avancement de la couverture. D'autre part, elle peut être utile afin d'extraire les trous dans la couverture, pour ensuite tenter de trouver de meilleures contraintes de génération permettant de combler ces trous dans la couverture.

A.4 Références

Cette annexe est basée sur la documentation du langage *e* et de Specman Elite™. Les documents d'intérêt consultés sont :

- *e* Language Reference Version 3.3.3
- Usage and Concepts Guide for Specman Elite™ Version 3.3.3
- Specman Elite™: Generation Guide Version 3.3.3

Annexe B

MÉTHODOLOGIE DE CONCEPTION AVEC LE LANGAGE *e*

Applying Aspect-Oriented Programming to Hardware Verification with *e*

S. Regimbal[1], J.-F. Lemire[1], Y. Savaria[1], G. Bois[1], E.-M. Aboulhamid[2], A. Baron[3]

[1] *École Polytechnique de Montréal, {regimbal, lemire, savaria, bois}@grm.polymtl.ca*
[2] *Université de Montréal, aboulham@iro.umontreal.ca*
[3] *PMC-Sierra, andre_baron@pmc-sierra.com*

Abstract

Considering that hardware verification has become the main bottleneck of most major digital design effort, efficient methods are required for implementing verification environments. In a previous paper, we proposed an aspect partitioning dedicated to hardware verification reuse. The proposed partitioning uses the aspect-oriented paradigm to enhance the verification effort by allowing efficient software modularization, which facilitates reuse. An aspect-oriented analysis for hardware verification allows a concise separation of the verification environment, by defining layers of crosscutting features in each class of the verification environment. This paper shows how we have applied our aspect partitioning methodology to the development of a verification environment with the e language. We apply our method to a concrete example, which consists of verifying a subset of a SOC protocol converter platform.

1. INTRODUCTION

A difficult part of hardware system design is to ensure functional correctness. The generation of large numbers of high quality test stimuli, to uncover design errors with a minimum effort, is one of the key challenges of modern simulation based hardware verification. Another challenge of verification is the need to emulate accurately the system environment with a test bench. Considering that verification can consume over 70% of the overall hardware design effort [1], efficient methods are required for implementing verification environments.

The implementation of verification environments requires the use of specialized programming languages. HDL languages such as VHDL or Verilog have improved the hardware design by allowing high-level hardware descriptions. However, because hardware verification needs some sophisticated mechanisms that HDL languages do not possess, HDL languages are not ideal for verification tasks. Recognized languages, like C++ and Java, address the requirements for complex data and algorithmic structures needed for functional verification. However, the problem with these languages for hardware verification is their inefficient interface with HDL designs. Thus, some alternatives are to use an extended language like Superlog™ that is a superset of the Verilog language, or to use a library, like TestBuilder, that extends the C++ language. Other alternatives are the use of Hardware Verification Languages (HVL), such as Verisity's *e*, OpenVera™ or RAVE™.

However, no language can ensure the quality of a verification environment. Thus efficient methodologies are still required and can be supported by a variety of languages. Nevertheless, a

methodology may be more easily applicable with a specific language. Furthermore, software reuse is known to be the most important issue for improving the productivity of software development processes. Most reuse approaches are based on an object-oriented methodology. The techniques proposed in [3] are guidelines for the creation of object-oriented verification environments. Object oriented programming decreases development time, and increases the quality of verification environments, by providing a high level programming support. In [4], some issues have been identified with the object-oriented paradigm in modeling properties of a system that influence several objects. These authors propose new programming techniques called aspect oriented programming (AOP), which can capture the aspects of a system. Aspect-oriented programming allows efficient software modularization, which facilitates reuse. In [6], authors expose that the Specman's *e* language possesses constructs that can support AOP. We have applied in [7] an aspect partitioning approach dedicated to hardware verification reuse. The proposed partitioning uses the aspect-oriented paradigm to enhance the verification effort.

This paper shows how we have applied our aspect partitioning methodology to the development of a verification environment, through the use of the *e* language. The design under test (DUT) is a subset of a communication protocol converter SOC platform designed at the GRM (*Groupe de Recherche en Microélectronique*) of École Polytechnique de Montréal. The verification environment is a self-checking and configurable test bench developed in an aspect-oriented framework, to emulate the environment of the DUT. The implementation of this verification environment was done with the *e* language, and was executed in the Specman Elite™ environment. Thus, the contribution of this work is to present, based on our partitioning methodology, how to apply aspect partitioning with the *e* language. We will illustrate this partitioning method through concrete examples.

The remainder of the paper is organized as follows. In section 2, we review the previous work done with the aspect-oriented methodology. Section 3 describes the DUT, and we apply the partitioning methodology to build an environment to verify that DUT. In section 4, we review the aspect-oriented feature of the *e* language, and we expose with two examples how we have applied AOP with the *e* language. Section 5 discusses the lessons learned from applying AOP to that verification example. Finally, in section 6, we present a conclusion and suggest possible areas of future investigation.

2. PREVIOUS WORK

The complexity of today's test benches has brought them to the level of highly complex software designs. It is thus natural to apply methodologies from the software domain to build efficient verification environments. An example of a maintainability problem the complexity of test benches causes occurs when verification engineers have to change a feature, which affects several functional blocks of the verification environment. It is often difficult for verification engineers to find every instance of a feature to be modified in thousands lines of code. If these features are not well tracked and implemented, this can introduce bugs.

To address this issue, software researchers are developing methodologies based on a new programming artefact: the aspect. Aspects are two things: at the design level, they are concerns that crosscut functional modules boundaries, and at the implementation level, they are programming constructs [9]. Crosscutting concerns are issues or programmer concerns that are not local to the natural unit of modularity. At the code level, in AOP, an aspect is a modular unit crosscutting the implementation, just as classes are modular implementation units in object-oriented programming.

The complexity of a good object-oriented analysis is to choose the right set of abstractions (objects) for a concrete problem. In an aspect-oriented analysis, the problem is similar: we need to find the right set of abstractions for a concrete problem, but this time the abstractions are aspects. In our case, the concrete problem is to develop efficient and reusable verification environments. Thus, we have proposed in [7] an aspect partitioning dedicated to the implementation of verification environments. This partitioning is separated into five categories that encompass all aspects involved in the development of a specific verification environment. Table 1 summarizes the definition of each category.

Table 1— Aspect Definitions

Aspect	Definition
Base	Defines the core of each class of the verification environment.
HDL	Defines the connections and interactions of the classes with the DUT.
Glue Logic	Defines the connections and interactions between classes.
Coverage	Defines coverage models for the verification environment.
Configuration	Defines each test case necessary to achieve the verification plan of the DUT. Test cases are constraints added on top of the verification environment.

The purpose of using this proposed partitioning is to improve portability, maintainability and customizability of a verification environment. Consequently, the reuse potential is increased due to the improved modularity of the verification environment. Moreover, this code structure provides an efficient support to implement typical modifications, such as:

- Adapting the verification environment to DUT modifications (portability);
- Extending the verification environment to include new design blocks according to a bottom-up verification method (maintainability);
- Modifying the verification environment to increase coverage according to the verification plan (customizability);

Figure 1 shows a high level block diagram of the proposed aspect partitioning. The core of our aspect partitioning is the Base Aspects category. In an object-oriented analysis, a set of classes is defined as the environment core. The Base Aspects block encompasses all these classes. Aspects from other categories extend these base classes. We perform these extensions by adding features (aspects) to the base classes in other modules. These features have a very low or no reuse potential, and by defining them as new aspects, we separate them from the base aspects, which have a high reuse potential. This methodology produces another level of modularity inside the verification environment.

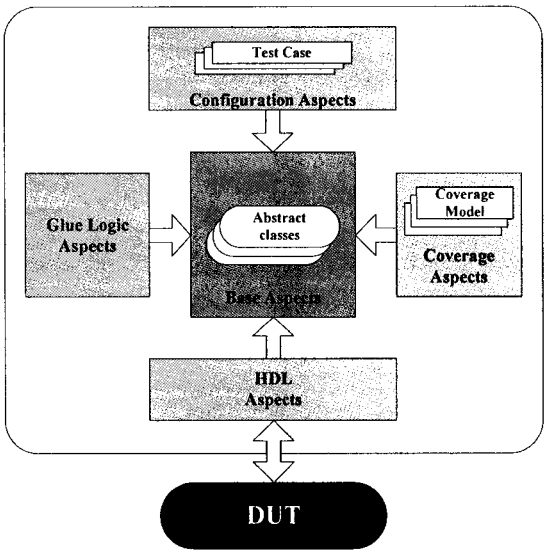


Figure 1— Aspects of a verification environment

3. THE STUDY

In this section, we explain how we applied the aspect partitioning methodology to build the verification environment for component verification. The discussion is supported by concrete examples from our environment. This environment takes the form of a test bench that can:

- Emulate the environment of the DUT;
- Self-check the response of the DUT to the stimuli generated by the environment;
- Provide a quantitative feedback on the progress of the functional coverage;
- Be easily configurable to allow test writing by users unfamiliar with the verification environment.

3.1 The DUT

The DUT is a module of a protocol converter system in development at the GRM of École Polytechnique de Montréal. The purpose of this protocol converter is the translation between pairs of network protocol stacks. The project focuses on protocols suitable for video streaming over telecommunication networks. This system is designed to support a variety of protocol stacks. The first version of the design is configured and programmed to convert Firewire (IEEE 1394.b) to Ethernet (IEEE 802.3).

The subsystem on which our verification efforts focus, as illustrated in Figure 2, is the block that

receives and memorizes the incoming packets. This module also creates packets related tags, identifies the packets' incoming protocols, and finally transmits the relevant information to a formatting engine.

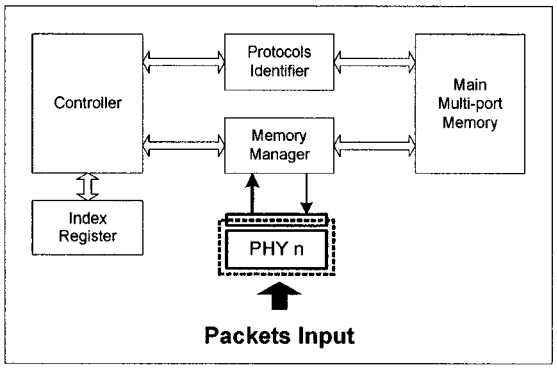


Figure 2— Part of the protocol converter

3.2. Verification Environment Functional Partitioning

Figure 3 presents a functional view of the verification environment. It shows the main functional verification components used to stimulate the DUT and to verify its behavior. This type of test bench functional partitioning is well presented in [1] and [8]. The main objective of this verification task is to check the proper packet processing of the *Memory Manager* through its interaction with the *Controller* and *Memory* modules. The *Packet Injector* sends data to the DUT in accordance with the *Verification Plan*. The *Controller* and *Memory* modules are not behavioral models of the DUT's functional blocks, but they instead emulate the interface behavior. The three Bus Functional Models (*BFM*) (controller-DUT, memory-DUT, and packet injector-DUT) are communication protocol abstractions between the DUT and each verification environment module interacting with it. Two scoreboards ensure a self-checking process that verifies if the data injected is transmitted adequately by the DUT to the *Controller* and the *Memory*. We use an *Assertion Checker* to verify internal and external protocol adherence, data integrity and value of signals at any stage of the simulation. Finally, the *Coverage Analysis* implements coverage models to evaluate the progress of the verification task.

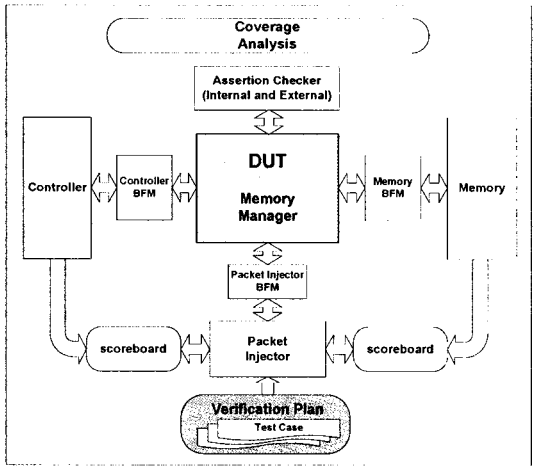


Figure 3— Verification Environment (Functional View)

3.3. Verification Environment Aspect Partitioning

Figure 4 shows an implementation view of the environment. It represents the environment functional partitioning and the different aspects within each functional block. Each block surrounding the DUT represents one of the main classes instantiated inside the Verification Ring, which is the top-level class of our environment. The HDL aspect connects the classes to the DUT while the Glue Logic aspect connects the classes between each other and defines the intercommunication mechanisms. It is remarkable that the scoreboard classes do not interact directly with the DUT, consequently the HDL aspect does not crosscut these classes.

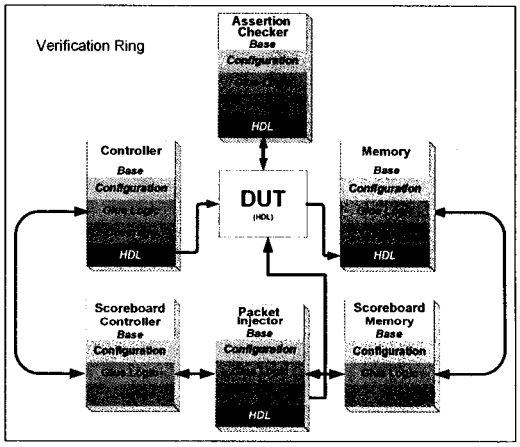


Figure 4— Verification Environment (Aspect View)

The aspects of Figure 4 are features that are commonly defined in the majority of the environment classes. It is beneficial to separate these common features and group them together in separate files, to create another level of modularity inside the environment, as shown in Figure 5. Each layer in Figure 5 represents a specific aspect dedicated to the Memory Manager Verification. These aspects are based on the aspect groups definitions introduced in table 1.

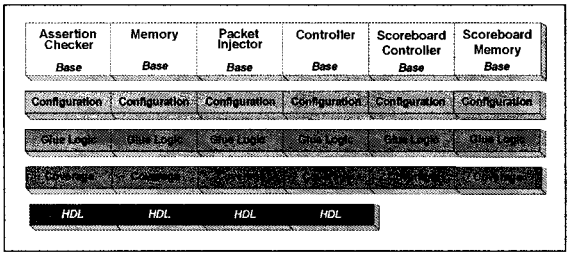


Figure 5— Aspect Partitioning

3.4. Concrete Benefits

Aspect partitioning provides concrete benefits when we want to integrate new HDL modules into the verification environment, according to a bottom-up verification approach. In this study, we concentrate our verification effort on the Memory Manager module. The reuse of various aspects will facilitate the integration of other modules as the system integration progresses. As mentioned at the end of section 2, the Base aspects have a high reuse potential, while the remaining environment aspects have a low reuse potential. Consequently, instead of building another verification environment from scratch for the integration of the other modules of figure 2, we reuse some Base aspects without any modification. Then, we only need to redefine parts of the other non-reusable aspects to address the verification of the extended DUT. We can thus facilitate the verification effort.

4. APPLYING AOP WITH THE E LANGUAGE

In this section, we expose how to achieve aspect-oriented programming with the e language. First, we review the aspect-oriented features initially presented in [6]. Then we present two examples of the use of these features for the realization of an aspect partitioning within a verification environment. The first example (section 4.2) suggests employing an aspect for the definition of intercommunications between objects of a verification environment. In the second example (section 4.3), we present the utilization of an aspect for the abstraction of a functional coverage model applied to the verification of the protocol converter system.

4.1 ASPECT-ORIENTED FEATURES

The *e* language provides two kinds of inheritance. The first kind is the *like* inheritance, which is a single inheritance mechanism equivalent to the inheritance mechanism of the C++ language. The other inheritance type is the *when* inheritance, which allows to extend a class based on a *determinant* field. Through the *when* inheritance, an object inherits only when the *determinant* condition of the class is met. Orthogonal extensions of a class by different aspects is an advantage of using *when* inheritance.

An aspect defines a behavior or a functionality of a system, which can affect several classes of this system. Therefore, aspects usually crosscut class boundaries. With the *e* language, an aspect can be defined with a *module* (concretely a file) that contains several extensions of the different classes affected by the aspect. These extensions are performed with the *when* inheritance mechanism without defining a *determinant* field.

To apply aspect-oriented programming, we also need mechanisms that allow the modification of methods affected by an aspect. These mechanisms are characterized by the use of is also, is first and is only keywords. The is first and is also keywords append code to the targeted method respectively at the beginning and at the end of its body. The is only construct overrides the previously defined method.

These mechanisms allow an aspect to crosscut the boundaries of classes by adding constraints or attributes to existing classes, or by allowing the extension or redefinition of methods. The amalgamation of these features provided by the e language allows an efficient implementation of an aspect-oriented methodology. The implementation of the aspect partitioning methodology proposed in [7] requires these specific e features.

4.2 Intercommunications abstraction by aspect

In our proposed methodology, an aspect that abstracts the intercommunication between objects is defined in the Glue Logic Aspect group. This abstraction allows developing classes in isolation, without being concerned about which data structure

will interact with the class, and which fields or methods might be connected to other classes. A Glue Logic aspect is specific to a particular verification environment. So, the example presented here is specific to the verification of the memory manager of the protocol converter system. Consequently, other verification environments that contain other modules of the protocol converter system can reuse the base classes but then, a new Glue Logic aspect must be defined. Figure 6 shows the essential structures of the code of the Glue Logic aspect dedicated to our verification environment.

```
aspect:glue_logic_mm
<'
extend verification_ring{
   pktI   : packet_injector is instance;
   ctr    : controller      is instance;
   mem    : memory          is instance;
   sb_ctr : score_phy_ctr   is instance;
   ...
   keep pktI.parent   == me;
   keep ctr.parent    == me;
   keep mem.parent    == me;
   keep sb_ctr.parent == me;
   ...
   event start_injection;
   on start_injection {
      start pktI.pkts_injection();
   };
   init_test() @sys.driving_clk is also {
      emit start_injection;
   };
};

extend packet_injector{
   parent: verification_ring;
   send_pkt(p: paquet)@sys.driving_clk is first {
       parent.sb_ctr.add(p);
   };
   ...
};

extend controller {
   parent: verification_ring;
   pkt_processing(p: packet)@sys.driving_clk is first {
        parent.sb_ctr.match(p);
      };
   ...
};

extend memory {
   ...
};

...
'>
```

Figure 6—A *Glue Logic* Aspect

The glue_logic_mm aspect crosscuts the boundaries of the base class definition of the verification_ring, packet_injector, controller and memory classes. Extensions are performed on these classes with the when inheritance. First, a packet_injector, a controller, a memory and a scoreboard are instantiated into the verification_ring class. Also, a parent field is declared in all classes aggregated in the verification_ring. This represents the connection part of the Glue Logic aspect.

For the interaction implementation, we use the extension mechanism to link the packet_injector, the scoreboard and the controller together. More precisely, we insert one line of code in methods of the packet_injector and the controller, with the is first keyword, to call a method from the scoreboard instantiated in the verification_ring. We also start the packet injection with an extension of the init_test method of the verification_ring with the is also keyword.

4.3. Functional coverage model by aspect

An important requirement to achieve the verification effort of a design is to reach the complete coverage of the verification plan. The verification plan contains the test cases defined to verify the DUT. The coverage of each test case of the verification plan during the verification increases the confidence in the DUT proper behavior. Applying several test cases on top of the verification environment requires the use of an aspect methodology. We can define a test case by defining a coverage model and by defining a set of constraints that will drive the pseudo-random generation of the verification environment. For each test case, we define a coverage model that can be abstracted by an aspect, and then we extend the verification environment with this aspect. The benefit of this approach is that it is possible to interchange the coverage models, without affecting the structure of the verification environment. Aspects that abstract functional coverage models are defined in the Coverage Aspect group.

The coverage model associated to a test case must be supported by metrics to allow the measurement of the completeness of a test program, and ultimately to allow the development of a more complete verification plan. The metric used in a coverage model can be for example the number of visited states or the number of transitions of a state machine, or simply the number of achieved coverage points extracted from the DUT. The portion of the coverage model presented in our example is a simple coverage model dedicated to the memory manager. The coverage model of our example is composed of two functional state machines. Figure 7 illustrates these state machines.

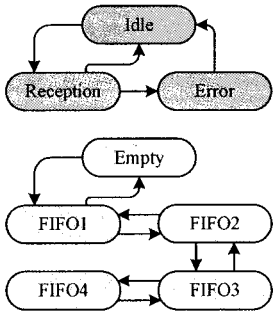


Figure 7—State Machine

The first state machine at the top of Figure 7 abstracts the action of packets processing in the DUT. In the Reception state, the DUT receives a packet normally. When packet processing or integrity errors are encountered, the state machine goes into the Error state. Otherwise, the state machine is in the Idle state. Concurrently, the second state machine abstracts the number of packets being processed by the protocol converter system. This second state machine represents a FIFO with the same size as the DUT's internal memory capacity. The following code, in Figure 8, implements the state machines of Figure 7 in the Coverage aspect.

```
aspect: coverage_model1
<'
extend packet_injector {...};

extend controller{
   !fifo_state: [EMPTY,FIFO1,FIFO2,FIFO3,FIFO4];
   !ctr_state: [IDLE,ERROR,RECEPTION];
   event fifo_event is {@add_pkt or @remove_pkt};
   event ctr_event is {@end_pkt or
                       @start_recept or @error_pkt };
   event start_reception;
   event add_pkt;
   event remove_pkt;
   ...
   ctr_machine()@sys.driving_clk is {
      all of
      {
         state machine ctr_state {
            *         => IDLE      {wait @end_pkt;};
            IDLE      => RECEPTION {wait @start_recept;};
            RECEPTION => ERROR     {wait @error_pkt;};
            ...
         };
         state machine fifo_state  {
           EMPTY => FIFO1  {wait @add_pkt;};
            ...
         };
      };
   };
   ...
   reset_controller() is also {
      fifo_state = EMPTY;
      ctr_state = IDLE;
      start ctr_machine();
   };
   receive_pkt()@sys.driving_clk is first {
      emit start_reception;
   };
   popPkt() is also{
      emit remove_pkt;
   };
   ...
   cover ctr_event is {
      item ctr_state;
      item fifo_state;
      cross ctr_state, fifo_state;
      transition ctr_state
         using illegal =(...);
   };
   cover fifo_event is {
      ...
   };
};

extend memory {...};
...
'>
```

Figure 8— A *Coverage* Aspect

The coverage_model1 aspect crosscuts the boundaries of the class packet injector, controller and memory. The code example shown here reveals the controller part of the complete coverage model. The first step in the definition of our coverage model is to define the state variables of the state machines. These variables are fifo_state and ctr_state. We then model the state machine with the state machine construct. The all of construct allows starting two threads that will execute these state machines concurrently. We also need to connect the events, which control the state machine, into the appropriate methods predefined by the base aspect of the controller. To collect the coverage information, we define two coverage groups with the cover construct provided by the e language. Inside the coverage group, we define coverage metrics used in our coverage model. For example, we cover in this example states and transitions of

each state machines and the crossing state of the two state machines.

5. DISCUSSION

New verification methodologies may target one of two possible areas of investigation: improvement of the verification environment design process or enhancement of the verification quality. AOP targets the verification environment design process improvement, but does not affect the quality of the verification task. By using AOP, we take another step towards increasing the kinds of verification concerns that can be captured cleanly within the source code. However, determining whether a new software development technique is useful and usable is a challenging task. Authors in [10] have attempted an initial assessment of AOP, by conducting exploratory experiments. The results of these experiments highlight the importance of the *aspect-core interface*. In our case, the *aspect-core interface* refers to the boundary between the core of our environment, which is the base aspect, and all other remaining aspects. This type of interface can be either narrow or wide. The interface is characterized as narrow when the effect of an aspect on the base aspect code has a well-defined scope. In other words, a narrow interface allows the designer to understand the aspect code without analyzing extensive parts of the base code. On the other hand, a wide interface means that it is necessary to look at both the aspect code and large chunks of the base code to understand the aspect code. In [10], the narrow interface helped the designers to complete assigned tasks. By contrast, wider interfaces seemed to hinder designers. By definition, an *HDL* aspect has a very narrow interface with its base definition, because it enables the connections with the DUT. The *Coverage* and *Glue Logic* aspects have also a narrow interface with their respective bases. The first uses coverage events defined specifically for its base aspect to define coverage models based on these events. The second defines intercommunication mechanism between classes. These mechanisms are specific to the classes affected by these aspects. Finally, the configuration aspects have a narrow interface with their corresponding base aspects, because they only apply constraints to the attributes declared in the classes of these base aspects. We believe that these narrow interfaces between each aspect of the environment and their corresponding base aspects help designers focus more easily on code related to a task, improving the designer's ability to reuse parts of the verification environment.

Another important issue would be to verify the AOP methodology impacts on execution speed and memory usage performances of the verification environment. Even if at the time of completing this paper we did not investigate these issues in detail, we believe that the AOP methodology has a minor impact on performances.

To conclude this discussion, we have also gathered four lessons learned from applying aspect-oriented programming for functional verification:

5.1. Follow the Principles

An aspect-oriented approach combined with an object-oriented approach allows a better modularity that eases the maintenance task. However, coding principles are necessary if we want to benefit from this approach. For example, a violation of the encapsulation principle of the object-oriented approach can damage the modularization established in the verification environment.

5.2 Orthogonal extension

Several people have implemented the verification environment in an orthogonal fashion. More specifically, after an initial definition of the base classes, which is the essence of the object-oriented analysis, the implementation of other aspects was accomplished by extending the base definitions. Authors in [6] expose how this can be accomplished with the *e* language. After experimenting with this technique, we found that the method proposed speed-up the implementation process by allowing an efficient parallel implementation through concurrent engineering.

5.3. Automation

An aspect-oriented separation of concerns has clearly demonstrated that some aspects have no reuse potential. Nevertheless, there could be an automation perspective for these aspects, by developing an automatic code generation process. For example, it could be possible to generate *e* code directly from the HDL description of the DUT to automate the HDL aspect generation.

5.4 Documentation

Even if the aspect-oriented approach allows a better separation of concerns, the code structure, on the other hand, becomes more difficult to understand than with a classical programming style. This stresses the requirement for keeping a solid documentation regarding the verification development process. We also suggest using UML notation to accomplish this task.

6.0 Conclusion

In this work, we have applied aspect-oriented programming to implement a hardware verification environment. We presented two aspect implementation examples: an intercommunication *Glue Logic* aspect and a *Coverage* aspect. An aspect-oriented design methodology offers an efficient separation of concerns by creating another level of modularity inside a test bench.

On the basis of the promising results obtained so far, we see three good directions to investigate. First, it would be very useful to have a method to code aspects such that they are more easily reusable as part of a reuse framework, using verification design patterns to support the implementation. Another direction worth investigating is the development of an automated or computer assisted method to create non-reusable aspects of a functional verification platform. Finally, the impact of the proposed method on performances should be characterized.

Acknowledgements

We would like to thank PMC-Sierra, Micronet, and the Natural Sciences and Engineering Research Council of Canada for their financial support. This project is made possible by the donation of a license of the Specman software by Verisity, and benefits from the technical guidance of PMC-Sierra. Finally, several Polytechnique's students, including Vincent Rocher, Gregory Donzel, and Maria Mbaye, developed the protocol converter code used in this research.

References

[1] J. Bergeron, Writing testbenches, *Functional verification of HDL models*, Kluwer Academic Publishers, 2000.

[2] R.S. Pressman, *Software Engineering: A practitioner's approach*, McGraw-Hill, 1997.

[3] F.I. Haque, K.A. Khan, J. Michelson, The Art of Verification with VERA, Verification Central, 2001.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming", *Proceeding of ECOOP*, 1997, pp. 220-242.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*, Addison-Wesley, 2000.

[6] Y. Hollander, M. Morley, and A. Noy, "The E Language: A Fresh Separation of Concerns", *Proceeding of TOOLS-38*, 2001, pp 42-50.

[7] S. Regimbal, J.F. Lemire, Y. Savaria, G. Bois, E.M. Aboulhamid, A. Baron, Aspect partitioning for Hardware Verification Reuse, accepted to IWSOC2002, Banff, July 2002.

[8] Cohen, B. *Component design by example... a step-by-step process using VHDL with UART as vehicle*, VhdlCohen Publishing, 2001.

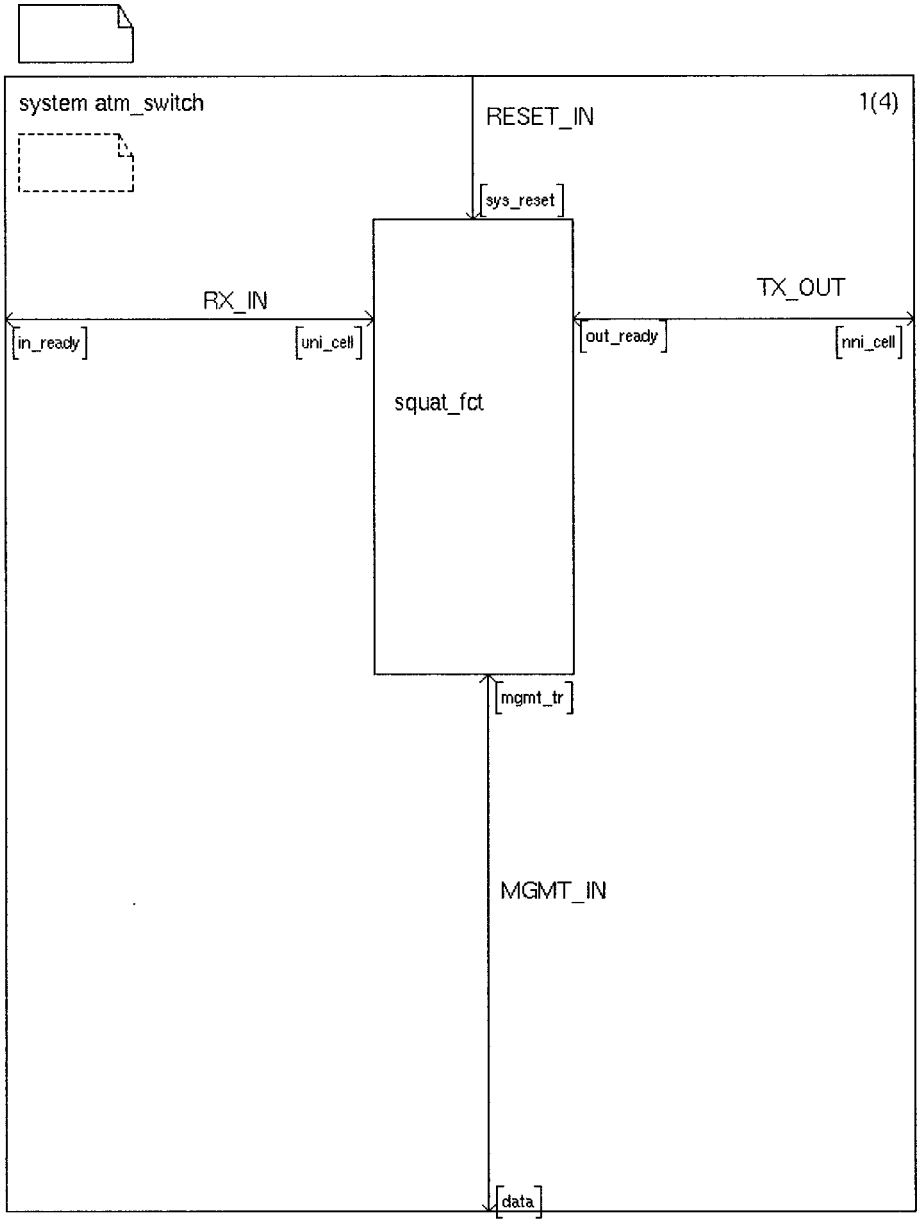
[9] XEROX Corporation. AspectJ - Aspect-Oriented Programming (AOP) for Java. Available at: http://aspectj.org.2001.

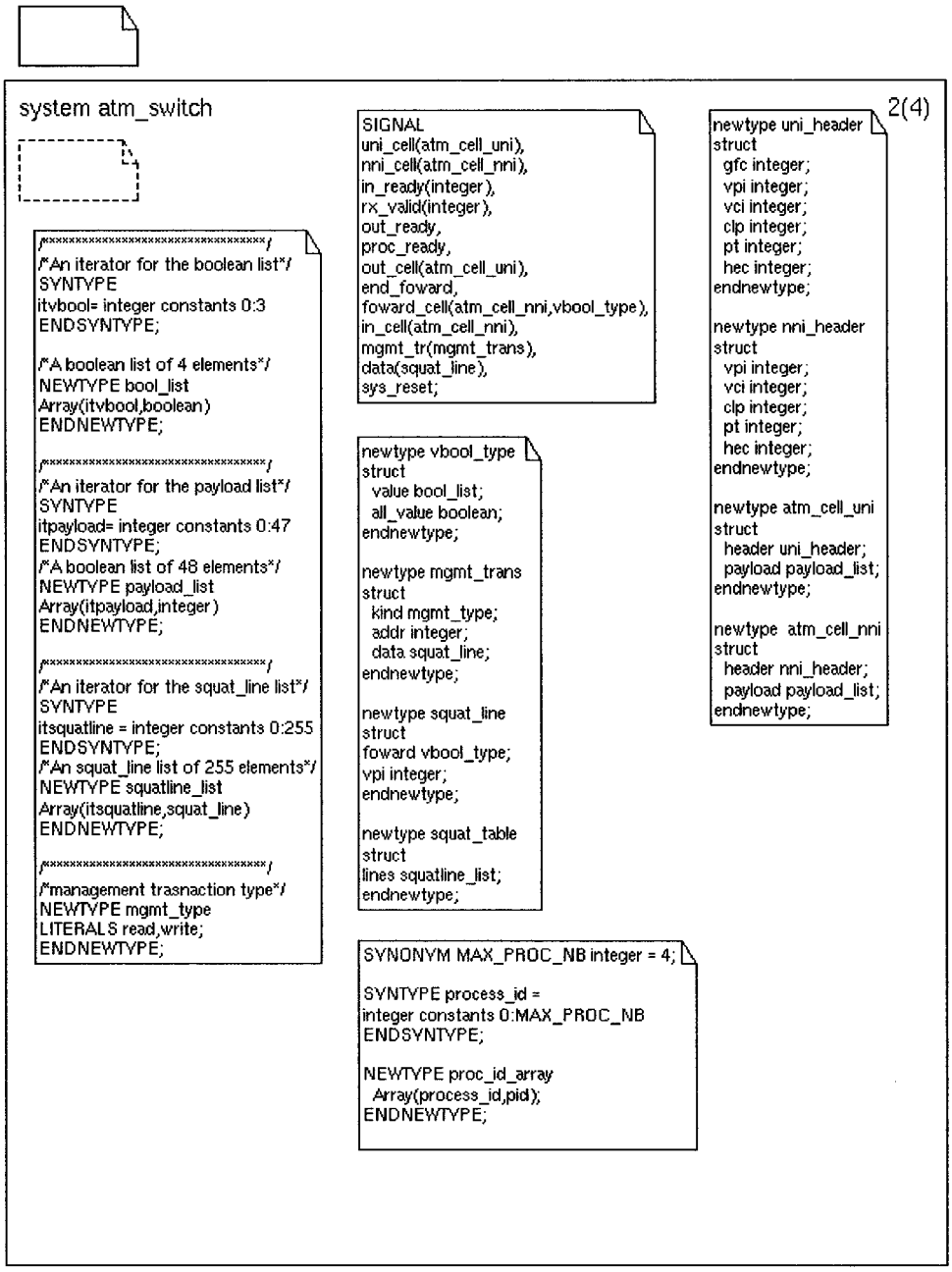
[10] R. J. Walker, E.L.A. Baniassad, G.C. Murphy, "An initial Assessment of Aspect-Oriented Programming", *Proceeding of ICSE*, 1999, pp. 120-130.

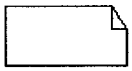
Annexe C

MODÈLE SDL DU COMMUTATEUR ATM

C.1 System atm_switch

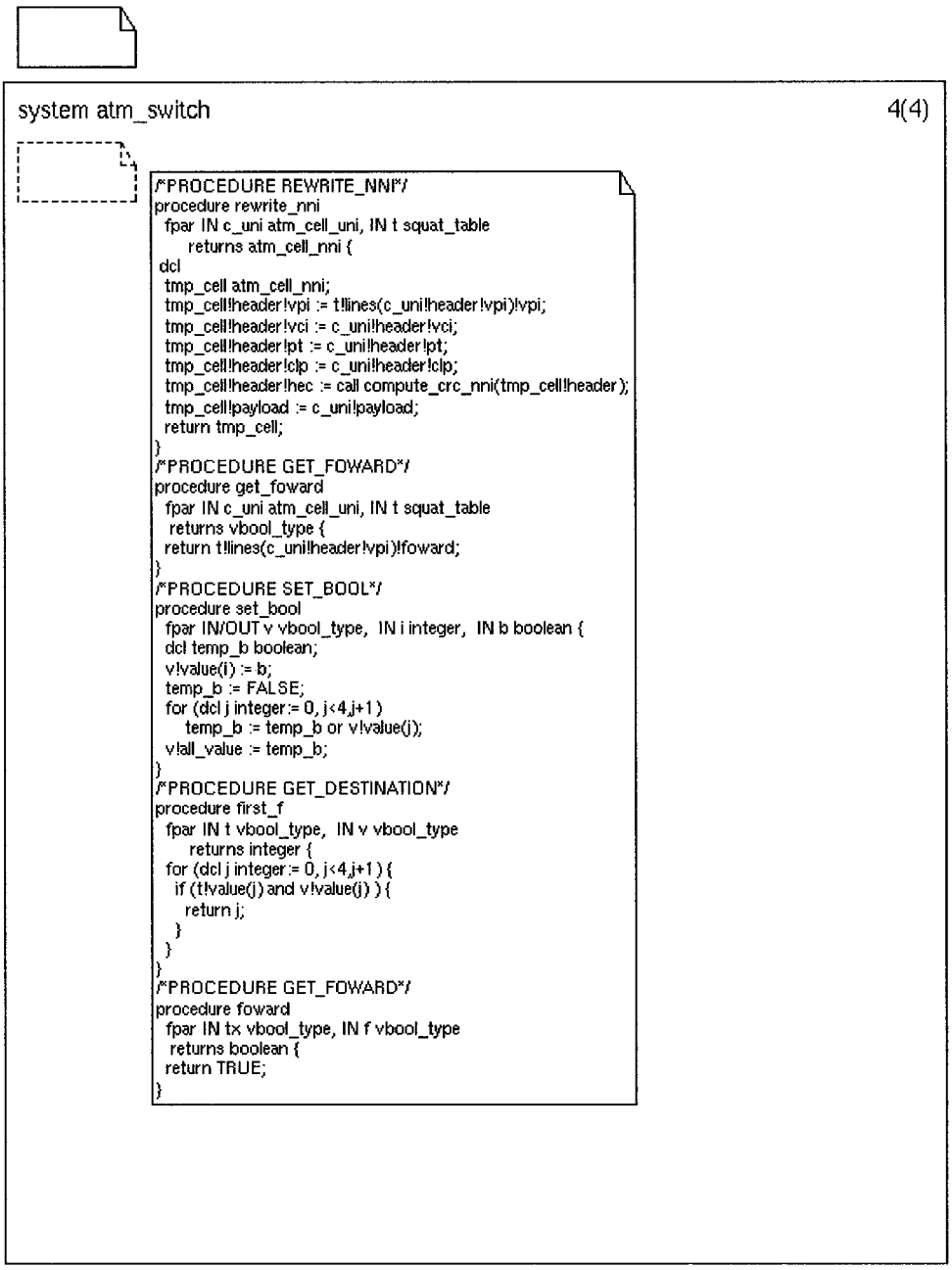



system atm_switch
2(4)
/*****/
/*An iterator for the boolean list*/
SYNTYPE
itvbool= integer constants 0:3
ENDSYNTYPE;
/*A boolean list of 4 elements*/
NEWTYPE bool_list
Array(itvbool,boolean)
ENDNEWTYPE;
/*****/
/*An iterator for the payload list*/
SYNTYPE
itpayload= integer constants 0:47
ENDSYNTYPE;
/*A boolean list of 48 elements*/
NEWTYPE payload_list
Array(itpayload,integer)
ENDNEWTYPE;
/*****/
/*An iterator for the squat_line list*/
SYNTYPE
itsquatline = integer constants 0:255
ENDSYNTYPE;
/*An squat_line list of 255 elements*/
NEWTYPE squatline_list
Array(itsquatline,squat_line)
ENDNEWTYPE;
/*****/
/*management trasnaction type*/
NEWTYPE mgmt_type
LITERALS read,write;
ENDNEWTYPE;
SIGNAL
uni_cell(atm_cell_uni),
nni_cell(atm_cell_nni),
in_ready(integer),
rx_valid(integer),
out_ready,
proc_ready,
out_cell(atm_cell_uni),
end_foward,
foward_cell(atm_cell_nni,vbool_type),
in_cell(atm_cell_nni),
mgmt_tr(mgmt_trans),
data(squat_line),
sys_reset;
newtype vbool_type
struct
value bool_list;
all_value boolean;
endnewtype;
newtype mgmt_trans
struct
kind mgmt_type;
addr integer;
data squat_line;
endnewtype;
newtype squat_line
struct
foward vbool_type;
vpi integer;
endnewtype;
newtype squat_table
struct
lines squatline_list;
endnewtype;
SYNONYM MAX_PROC_NB integer = 4;
SYNTYPE process_id =
integer constants 0:MAX_PROC_NB
ENDSYNTYPE;
NEWTYPE proc_id_array
Array(process_id,pid);
ENDNEWTYPE;
newtype uni_header
struct
gfc integer;
vpi integer;
vci integer;
clp integer;
pt integer;
hec integer;
endnewtype;
newtype nni_header
struct
vpi integer;
vci integer;
clp integer;
pt integer;
hec integer;
endnewtype;
newtype atm_cell_uni
struct
header uni_header;
payload payload_list;
endnewtype;
newtype atm_cell_nni
struct
header nni_header;
payload payload_list;
endnewtype;

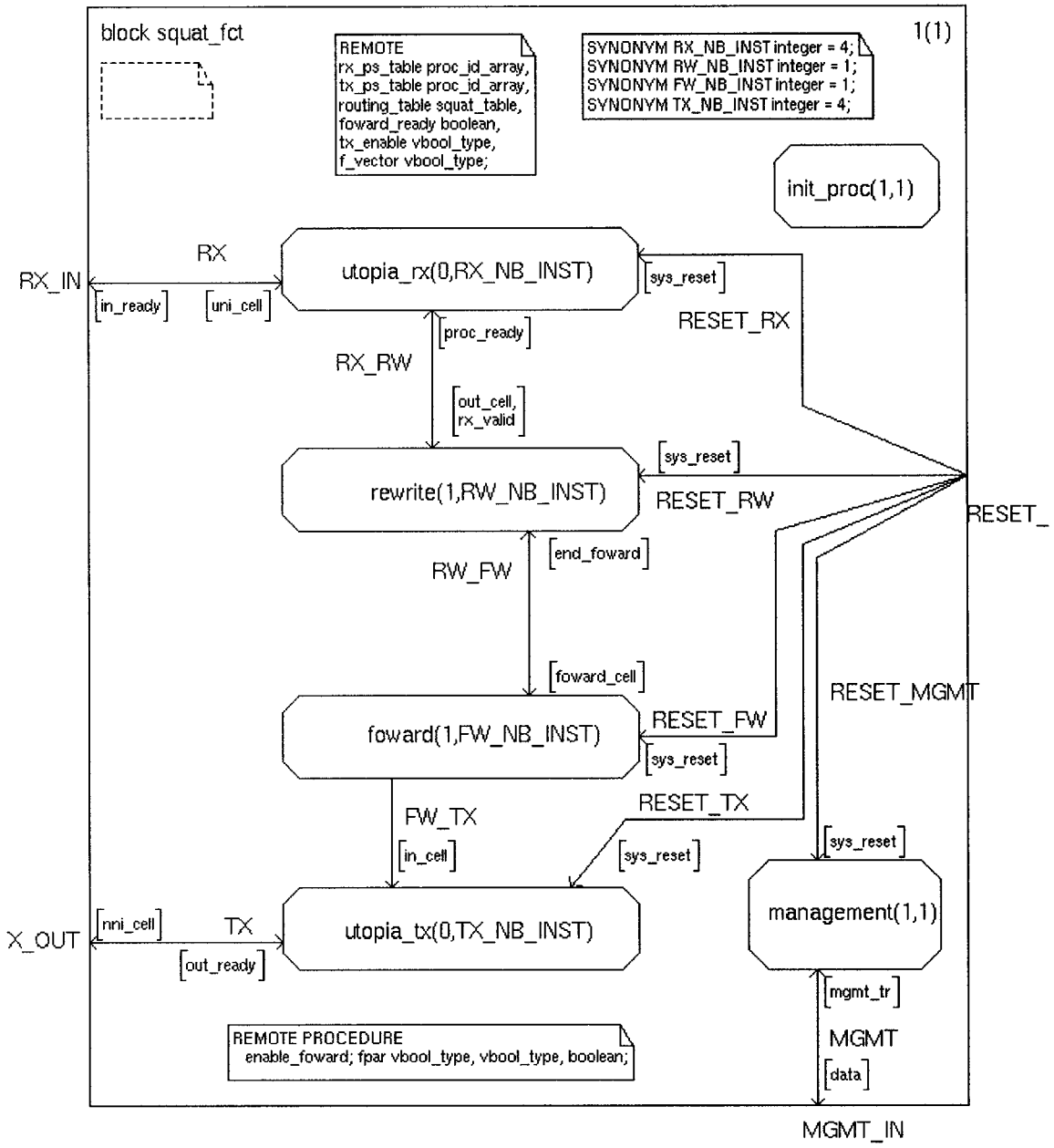


system atm_switch 3(4)

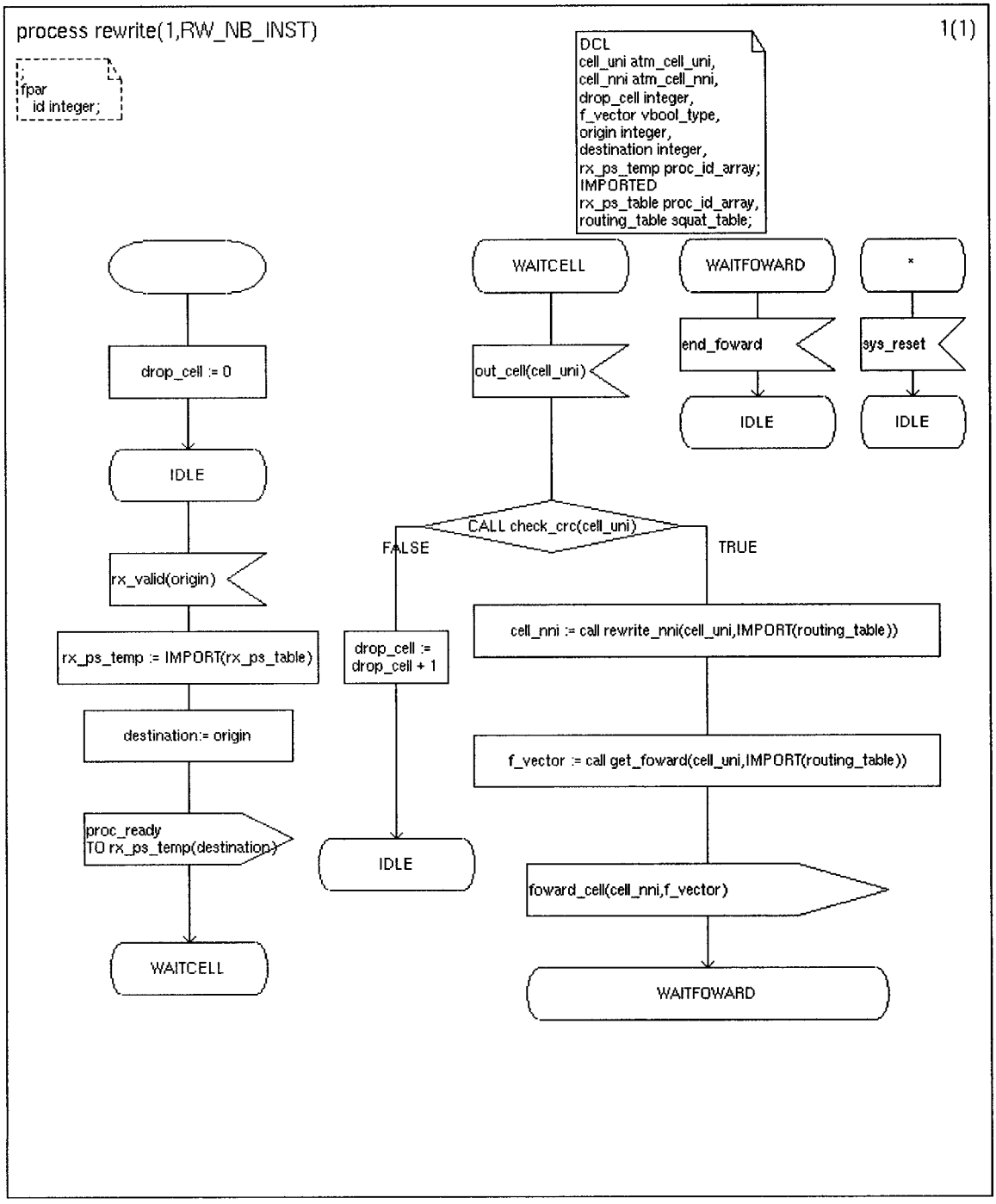
```
/*PROCEDURE COMPUTE_CRC_UNI*/
procedure compute_crc_uni
  fpar IN h uni_header
  returns integer
{
  dcl result integer;
  h!vpi := 0;
  /* some actions to compute the CRC*/
  return result;
}

/*PROCEDURE COMPUTE_CRC_NNI*/
procedure compute_crc_nni
  fpar IN h nni_header
  returns integer
{
  dcl result integer;
  h!vpi := 0;
  /* some actions to compute the CRC*/
  return result;
}

/*PROCEDURE CHECK_CRC*/
procedure check_crc
  fpar IN c_uni atm_cell_uni
  returns boolean
 {
    DCL temp integer;
    temp := call compute_crc_uni(c_uni!header);
    if (temp = c_uni!header!hec )
       return TRUE;
    else
       return FALSE;
 }

/*PROCEDURE GET_DATA*/
procedure get_data
  fpar IN t squat_table,
       IN m mgmt_trans
  returns squat_line
{
  return t!lines(m!addr);
}

/*PROCEDURE SET_DATA*/
procedure set_data
  fpar IN t squat_table,
       IN m mgmt_trans
{
 t!lines(m!addr) := m!data;
}
```


system atm_switch
4(4)
/*PROCEDURE REWRITE_NNI*/
procedure rewrite_nni
fpar IN c_uni atm_cell_uni, IN t squat_table
returns atm_cell_nni {
dcl
tmp_cell atm_cell_nni;
tmp_cell!header!vpi := t!lines(c_uni!header!vpi)!vpi;
tmp_cell!header!vci := c_uni!header!vci;
tmp_cell!header!pt := c_uni!header!pt;
tmp_cell!header!clp := c_uni!header!clp;
tmp_cell!header!hec := call compute_crc_nni(tmp_cell!header);
tmp_cell!payload := c_uni!payload;
return tmp_cell;
}
/*PROCEDURE GET_FOWARD*/
procedure get_foward
fpar IN c_uni atm_cell_uni, IN t squat_table
returns vbool_type {
return t!lines(c_uni!header!vpi)!foward;
}
/*PROCEDURE SET_BOOL*/
procedure set_bool
fpar IN/OUT v vbool_type, IN i integer, IN b boolean {
dcl temp_b boolean;
v!value(i) := b;
temp_b := FALSE;
for (dcl j integer:= 0, j<4,j+1)
temp_b := temp_b or v!value(j);
v!all_value := temp_b;
}
/*PROCEDURE GET_DESTINATION*/
procedure first_f
fpar IN t vbool_type, IN v vbool_type
returns integer {
for (dcl j integer:= 0, j<4,j+1) {
if (t!value(j) and v!value(j)) {
return j;
}
}
}
/*PROCEDURE GET_FOWARD*/
procedure foward
fpar IN tx vbool_type, IN f vbool_type
returns boolean {
return TRUE;
}

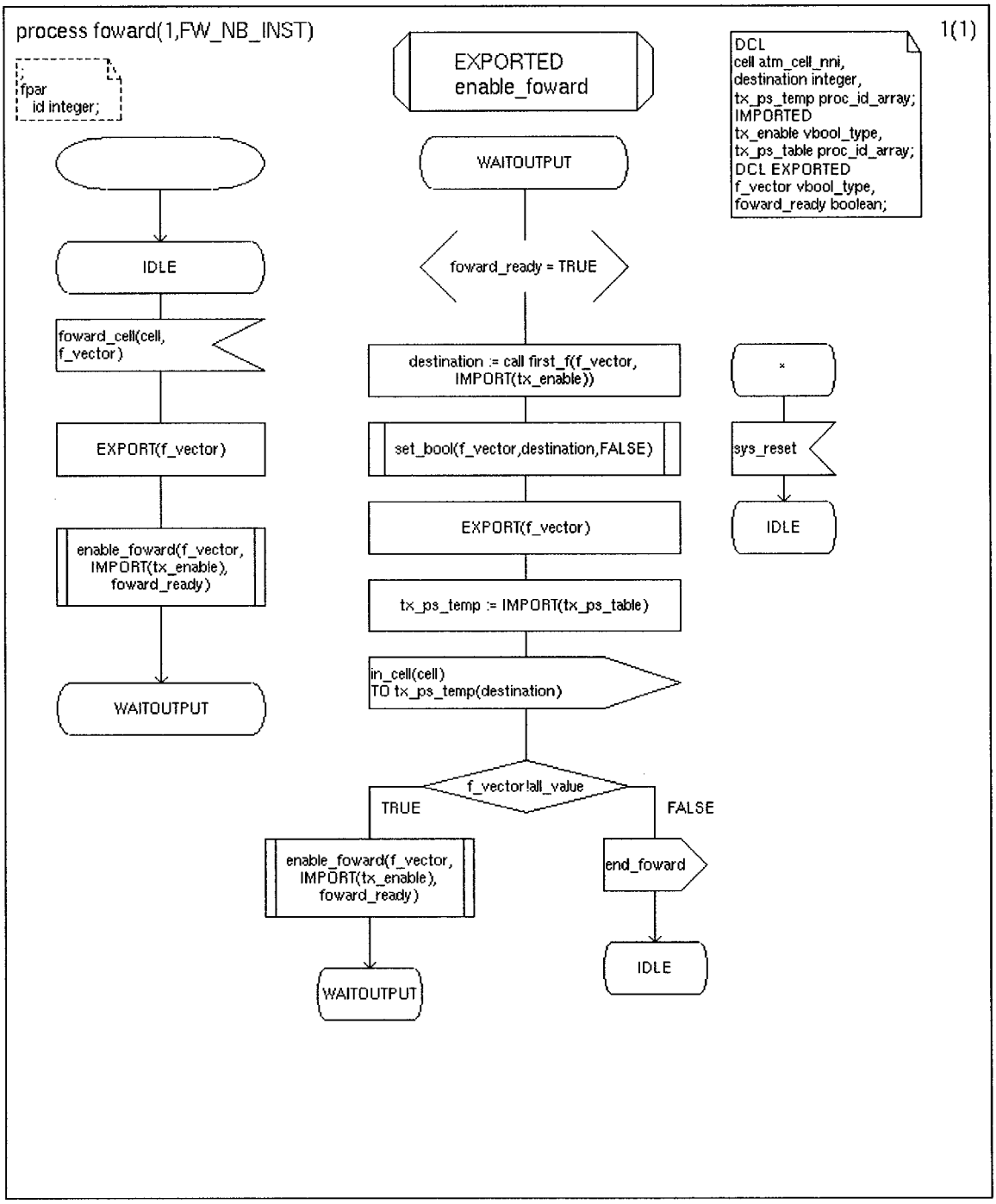
C.1.1 Block squat_fct



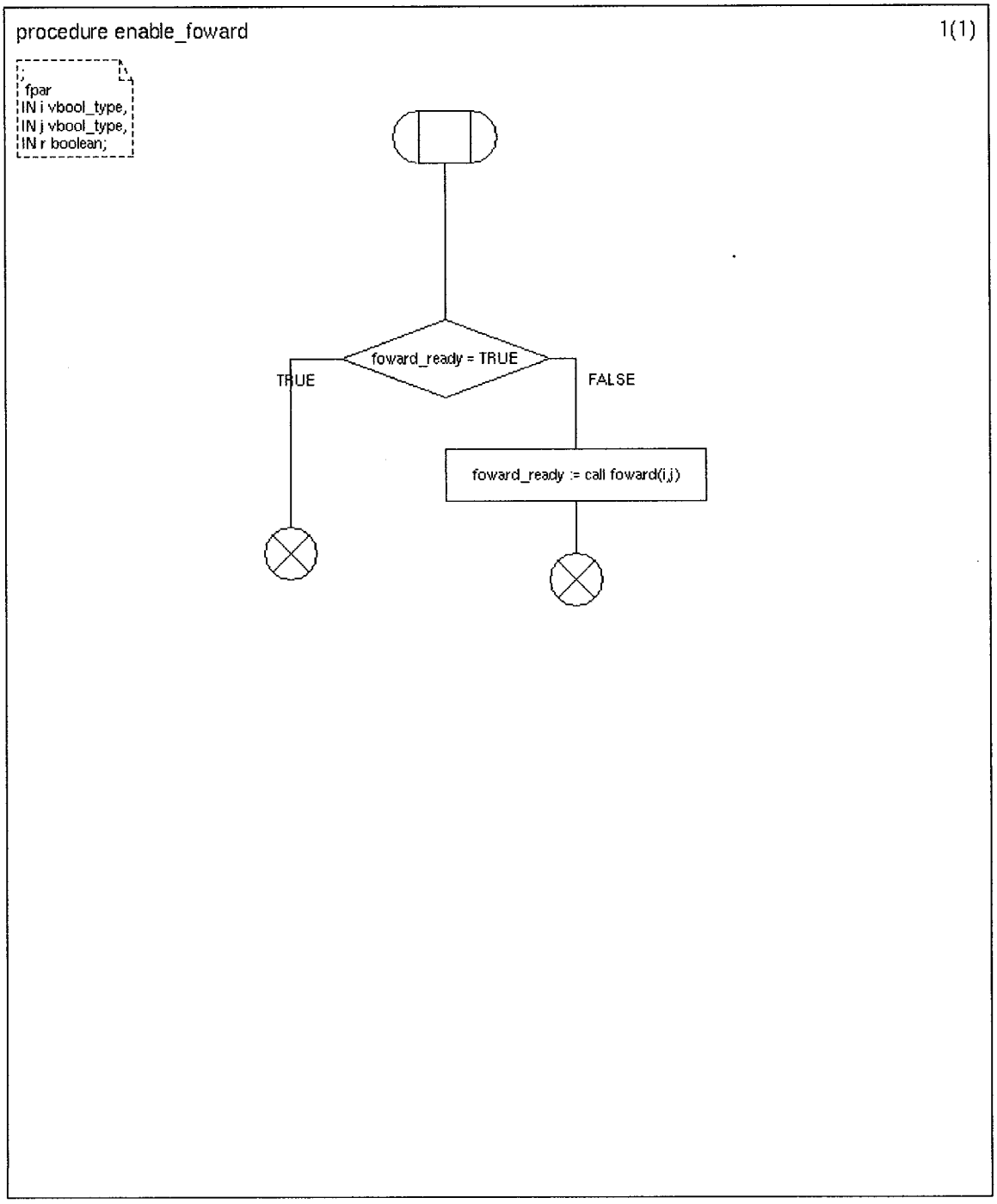
C.1.1.1 Process rewrite



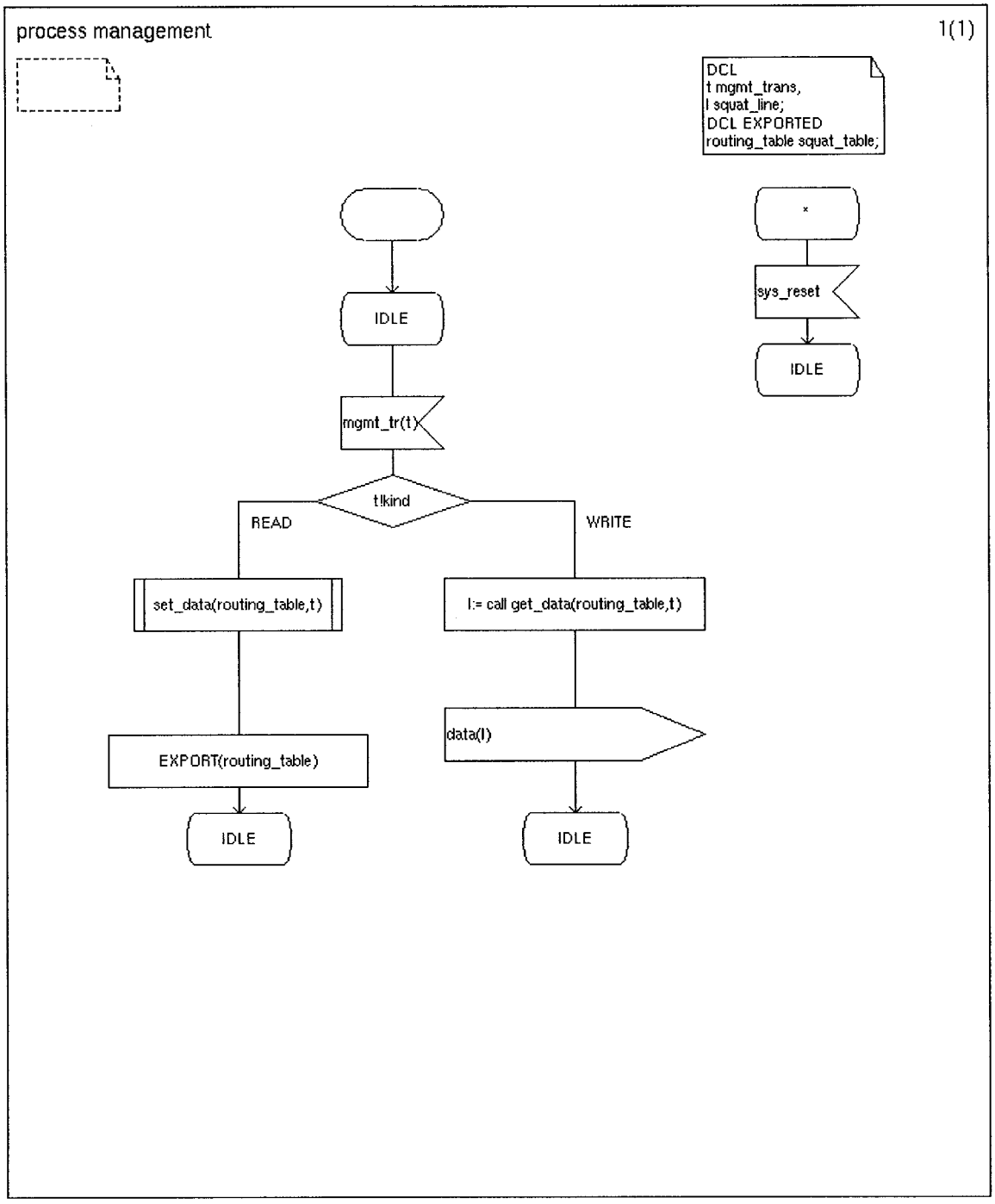
C.1.1.2 Process forward



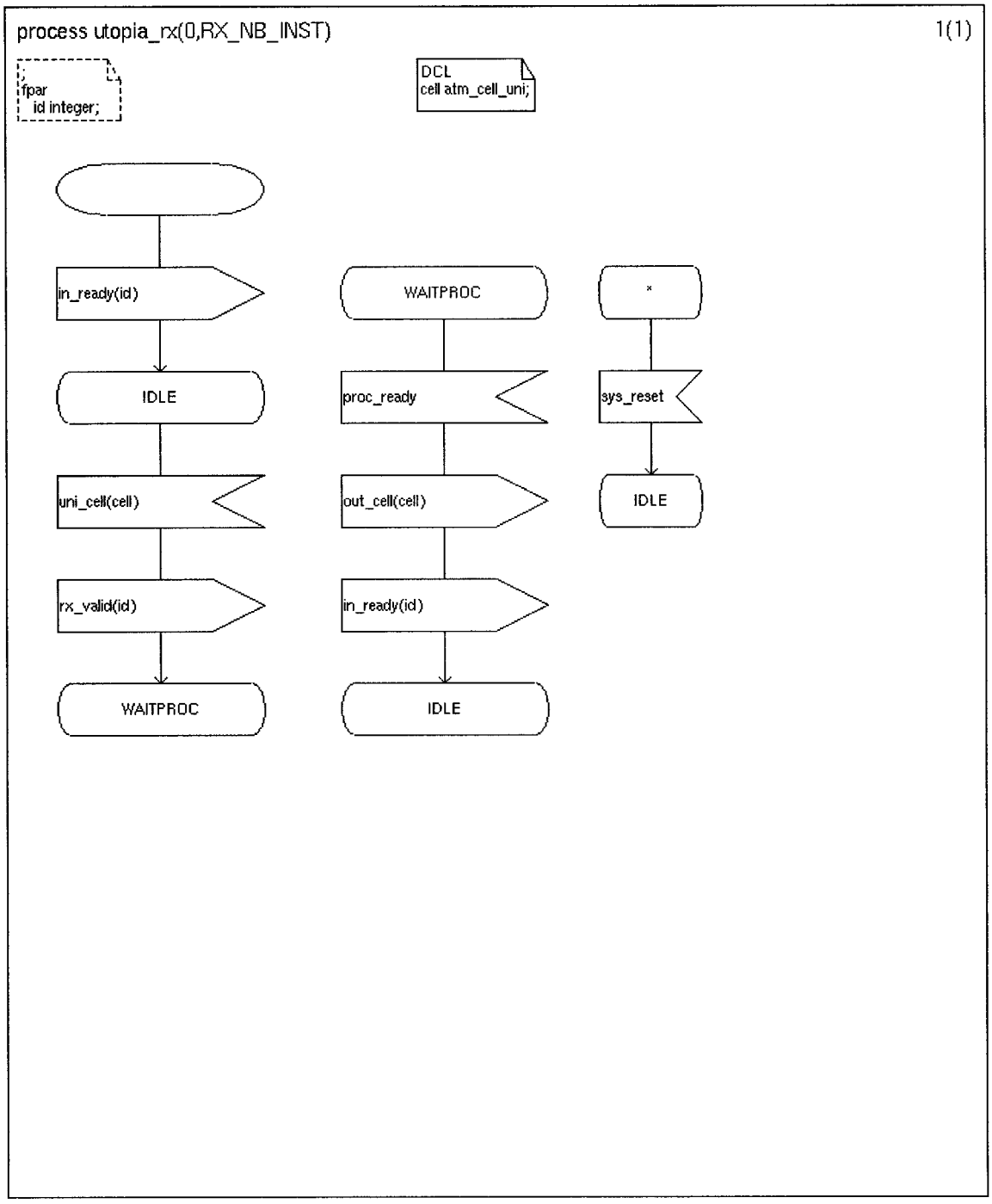
C.1.1.2.1 Procedure enable_forward



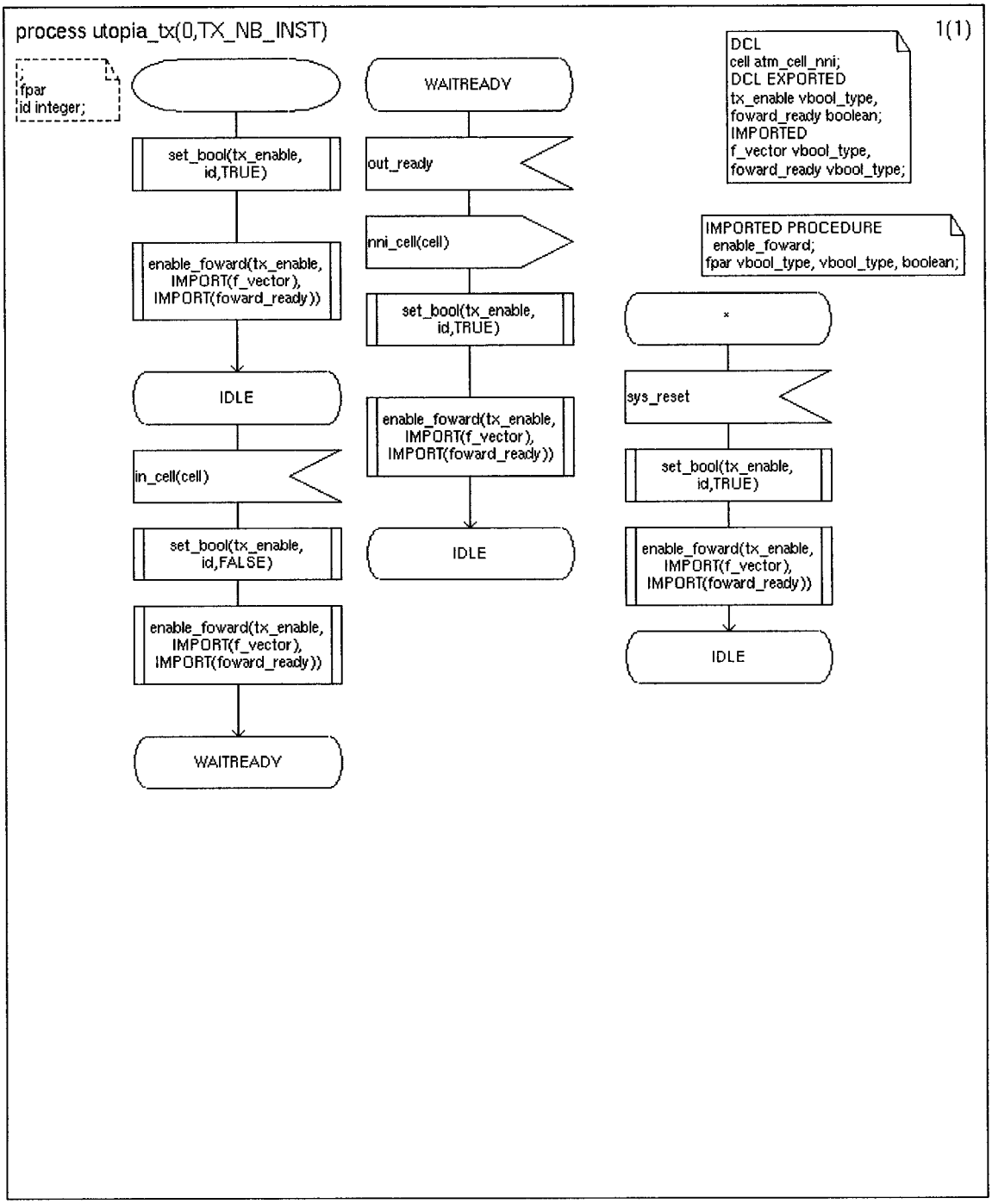
C.1.1.3 Process management



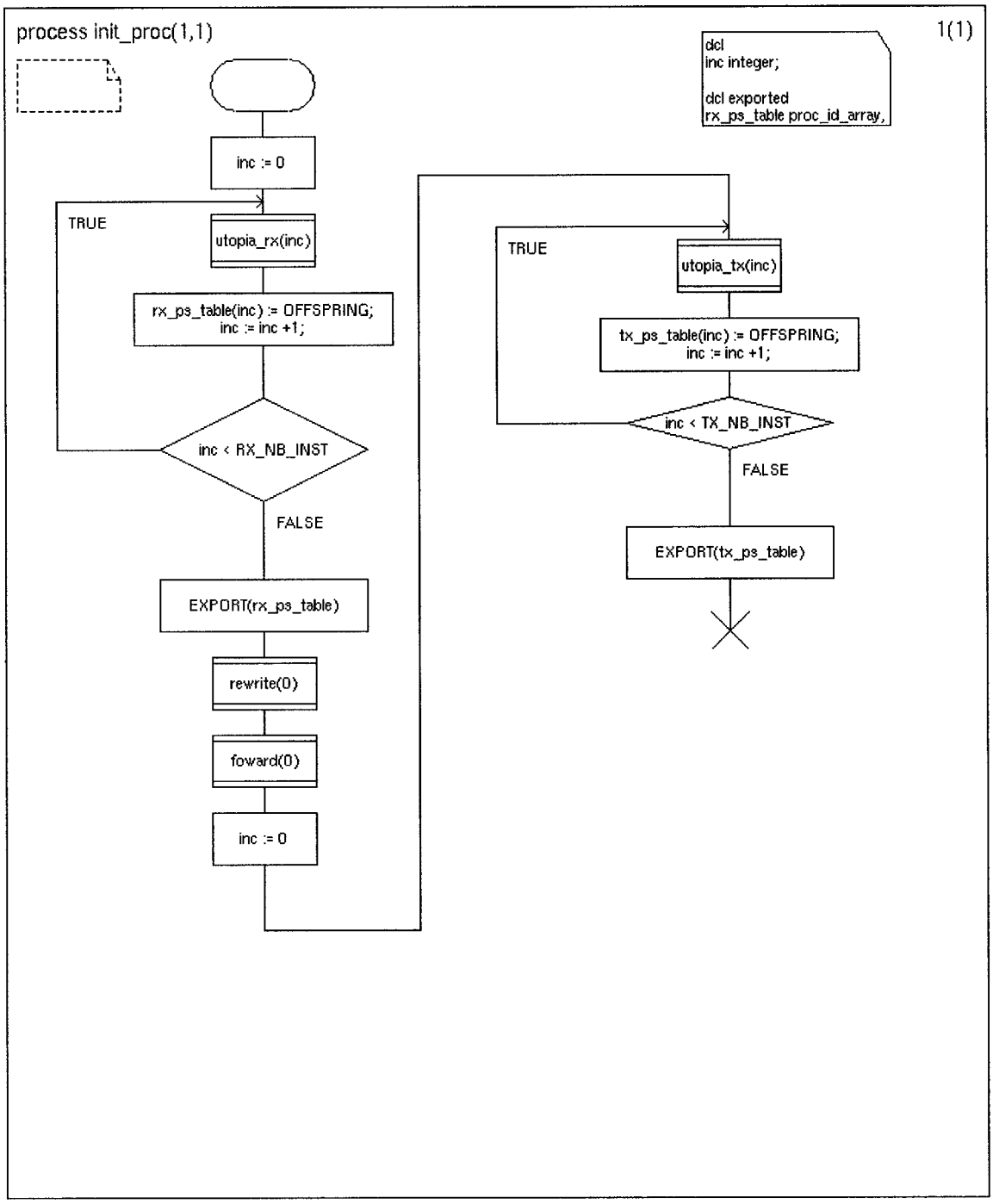
C.1.1.4 Process utopia_rx



C.1.1.5 Process utopia_tx



C.1.1.6 Process init_proc



Annexe D

L'IMPLÉMENTATION DU MODULE D'ANALYSE DE LA COUVERTURE

Cette annexe inclut les fichiers sources du module d'analyse de la couverture produit pour le commutateur ATM. La version donnée dans cette annexe n'est pas générique puisqu'elle a été configurée pour le commutateur ATM. Le Tableau D-1 donne une description sommaire des différents fichiers utilisés pour définir un module d'analyse de la couverture.

Tableau D-1: Description des fichiers du module d'analyse de la couverture

Section	Fichier	Description
D.1	mac_emodel.e	La spécifcation exécutable, en language e, dérivé du modèle SDL est définie. De plus, la spécification est instrumentée afin extraire les événements servant à mesurer les flots transactionnels.
D.2	mac_emodel_def.e	Dans ce fichier, on retrouve la définition d'éléments servant à implémenter la spécification. On retrouve, par exemple, la définition des canaux de communication et des variables d'états.
D.3	mac_tracking_lib.e	Dans ce fichier, les éléments nécessaires à la couverture de flots transactionnels sont définis. On retrouve, par exemple, le contrôleur des flots transactionnels, le traqueur de chemin et le système d'échantillonnage des flots transactionnels.
D.4	mac_tracking_def.e	Dans ce fichier, on retrouve plusieurs définitions et initialisations pour le module d'anlayse de la couverture. On retouve, par exemple, la définition et l'initialisation de la base de donnée des branches de base et du dictionnaire des chemins
D.5	mac_tracking_macros.e	Ce fichier décrit les macros utilisées pour créer les groupes de couverture du module d'analyse de la couverture.
D.6	mac_utilities.e	Ce fichier un fifo générique utilisé par le module d'analyse de la couverture.

D.1 Fichier MAC emodel

```
-------------------------------------------------------------------------------
-- Title          : Mac emodel
-- Project        : MAC (Module d'Analyse de la Couverture)
-------------------------------------------------------------------------------
-- File           : mac_emodel.e
-- Author         : Sebastien REGIMBAL
-- Created        : 17/11/2002
-- Last modified  : 17/11/2002
-------------------------------------------------------------------------------
-- Description    :
-- The module includes the transactional graph mapped to the e language.
-- This file must be generated from the SDL description
-------------------------------------------------------------------------------
    module: mac_emodel
    aspect: coverage
<'
    -- transactions definition (used inside the testbench)
    import fvg_transactions;
    -- mac utilities used inside the coverage module
    import mac_utilities;
    -- mac library for the tracking of the transactional flow
    import mac_tracking_lib;
    -- definition for the emodel
    import mac_emodel_def;
'>

-------------------------------------------------------------------------------
-- name: mac_emodel
-- description:
-- the transactional graph
-------------------------------------------------------------------------------
<'
struct mac_emodel {
   -----------------------------------------------------
   -- TRANSACTION FLOW CONTROLLER
   tflow_ctr: tflow_controller;

   -----------------------------------------------------
   -- CHANNEL DECLARATION
   RESET    : RESET mac_channel;
   RX_IN[4] : list of RX_IN mac_channel;
   RX_R[4]  : list of RX_R mac_channel;
   RF       : RF mac_channel;
   F_TX[4]  : list of F_TX mac_channel;
   TX_OUT[4]: list of TX_OUT mac_channel;
   MGMT_CH  : MGMT_CH mac_channel;

   -----------------------------------------------------
   -- STATE DECLARATION
   !rx_st[4] : list of utopia_rx_state;
   !tx_st[4] : list of utopia_tx_state;
   !rw_st    : rw_state;
   !fw_st    : fw_state;
   !mgmt_st  : mgmt_state;

   -----------------------------------------------------
   -- GLOBAL VARIABLE DECLARATION
   rw_table: squat_table;
   tx_enable: vbool_type;

   ------------------------------------------------------------
   --************************************************************
   -- FUNCTION THAT STARTS THE EXECUTABLE SPECIFICATION
   --************************************************************
   ------------------------------------------------------------
   transactional_graph() is {
      -- PROCESS INSTANTIATION
      for i from 0 to 3 do {
          start rx_utopia_ps(i);
          start tx_utopia_ps(i);
      };
      start rewrite_ps(0);
      start foward_ps(0);
      start mgmt_ps(0);
   };   -- transactional_graph

   ------------------------------------------------------------
   --************************************************************
   -- UTOPIA_RX process
   --************************************************************
   ------------------------------------------------------------
   rx_utopia_ps(id: int)@sys.any is {
      -- LOCAL TOKEN DECLARATION
      var token: mac_token = new;

      -- LOCAL VARIABLE DECLARATION
      var cell: UNI atm_cell;

      -- INITIAL ACTION
      RX_IN[id].in_ready_out(id);

      -- EXTENDED STATE MACHINE DESCRIPTION
      state machine rx_st[id] {
         -- state IDLE to nextstate WAITPROC
         IDLE     =>  WAITPROC   {
            -- INPUT
            RX_IN[id].uni_cell_in(cell);

            ------------------------------------------------
            -- ************COVERAGE STUFFS*****************
            -- TRANSACTION BIRTH
            token.set(cell); -- token definition
            tflow_ctr.trans_birth(id,token);
            -- TRACK DOWN THE FLOW
            tflow_ctr.add_track(RX_IW,id,token);
            -- ********************************************
            ------------------------------------------------

            -- ACTIONS
            RX_R[id].rx_valid_out();
         };
         -- state WAITPROC to nextstate IDLE
         WAITPROC =>  IDLE {
            -- INPUT
```

```
        RX_R[id].proc_ready_in();

        ------------------------------------------------
        -- *************COVERAGE STUFFS*****************
        -- TRACK DOWN THE FLOW
        tflow_ctr.add_track(RX_WI,id,token);
        -- ********************************************
        ------------------------------------------------

        -- ACTIONS
        RX_R[id].out_cell_out(cell,token);
        RX_IN[id].in_ready_out(id);
      };
      -- any state to nextstate IDLE
      *        => IDLE       {
        -- INPUT
        RESET.reset_in();

        -- ACTIONS
        -- NONE
      };
    };
};   -- END UTOPIA_RX process

------------------------------------------------------------------
--*****************************************************************
-- UTOPIA_TX process
--*****************************************************************
------------------------------------------------------------------
tx_utopia_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARTION
   var cell: NNI atm_cell;

   -- INITIAL ACTIONS
   tx_enable.set(id,TRUE);

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine tx_st[id] {
      -- state IDLE to nextstate WAITCELL
      IDLE     => WAITREADY   {
         -- INPUT
         F_TX[id].in_cell_in(cell,token);

         ------------------------------------------------
         -- *************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(TX_IW,id,token);
         -- ********************************************
         ------------------------------------------------

         -- ACTIONS
         tx_enable.set(id,FALSE);
      };
      -- state WAITCELL to nextstate WAITFOWARD
      WAITREADY => IDLE {
         -- INPUT
         TX_OUT[id].out_ready_in();

         ------------------------------------------------
         -- *************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(TX_WI,id,token);
         -- TRANSACTION DEATH
         tflow_ctr.trans_death(id,TX,token);
         -- ********************************************
         ------------------------------------------------

         -- ACTIONS
         TX_OUT[id].nni_cell_out(cell);
         tx_enable.set(id,TRUE);

      };
      -- any state to nextstate IDLE
      *        => IDLE       {
         -- INPUT
         RESET.reset_in();

         -- ACTIONS
         tx_enable.set(id,TRUE);
      };
   };
};   -- END UTOPIA_TX process

------------------------------------------------------------------
--*****************************************************************
-- MANAGEMENT process
--*****************************************************************
------------------------------------------------------------------
mgmt_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARATION
   var t: mgmt_trans;

   -- INITIAL ACTIONS
   -- none

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine mgmt_st {
      -- state IDLE to nextstate IDLE
      IDLE     => IDLE {
         -- INPUT
         MGMT_CH.mgmt_req_in(t);

         -- ACTIONS
         if t.kind == WRITE then {
            rw_table.set_data(t.addr,t.data);
         } else { -- READ
            MGMT_CH.data_out(rw_table.get_data(t.addr));
         };
      };
      -- any state to nextstate IDLE
      *        => IDLE  {
         -- INPUT
         RESET.reset_in();
```

```
        -- ACTIONS
        -- none
      };
   }; -- END MANAGEMENT process
};

-----------------------------------------------------------------
--***************************************************************
-- REWRITE process
--***************************************************************
-----------------------------------------------------------------
rewrite_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARATION
   var origin: int;
   var cell_in: UNI atm_cell;
   var cell_out: NNI atm_cell;
   var f_vector: vbool_type;
   var drop_cell: int;

   -- INITIAL ACTIONS
   drop_cell = 0;

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine rw_st {
      -- state IDLE to nextstate WAITCELL
      IDLE     =>  WAITCELL    {
         -- INPUT
         first of {
            {
               RX_R[0].rx_valid_in();
               origin = 0;
            };
            {
               RX_R[1].rx_valid_in();
               origin = 1;
            };
            {
               RX_R[2].rx_valid_in();
               origin = 2;
            };
            {
               RX_R[3].rx_valid_in();
               origin = 3;
            };
         };

         -- ACTIONS
         RX_R[origin].prcc_ready_out();
      };
      -- state WAITCELL to nextstate INT_WAITCELL1
      WAITCELL =>  INT_WAITCELL1{
         -- INPUT
         RX_R[origin].out_cell_in(cell_in,token);

         ------------------------------------------------
         -- *************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(RW_WIW1,id,token);
         -- ************************************************
         ------------------------------------------------


         -- ACTIONS
         -- none
      };
      -- state INT_WAITCELL1 to nextstate WAITFOWARD
      INT_WAITCELL1 => WAITFOWARD {
         -- INPUT
         sync true(cell_in.check_crc() == TRUE);

         ------------------------------------------------
         -- *************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(RW_IW1W,id,token);
         -- ************************************************
         ------------------------------------------------


         -- ACTIONS
         f_vector = cell_in.foward(rw_table);
         cell_out = cell_in.rewrite(rw_table);
         RF.foward_cell_out(cell_out,f_vector,token);
      };
      -- state INT_WAITCELL1 to nextstate IDLE
      INT_WAITCELL1 => IDLE {
         -- INPUT
         sync true(cell_in.check_crc() == FALSE);

         ------------------------------------------------
         -- *************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(RW_IW1I,id,token);
         -- TRANSACTION DEATH
         tflow_ctr.trans_death(id,CRC,token);
         -- ************************************************
         ------------------------------------------------


         -- ACTIONS
         drop_cell += 1;

      };

      -- state WAITFOWARD to nextstate IDLE
      WAITFOWARD => IDLE      {
         -- INPUT
         RF.end_foward_in();

         -- ACTIONS
         -- none
      };
      -- any state to nextstate IDLE
      *        =>  IDLE       {
         -- INPUT
         RESET.reset_in();

         -- ACTIONS
         drop_cell = 0;
      };
};
```

```
};   -- END REWRITE process

------------------------------------------------------------
--************************************************************
-- FOWARD process
--************************************************************
------------------------------------------------------------
foward_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token  = new;

   -- LOCAL VARIABLE DECLARTION
   var cell: NNI atm_cell;
   var destination: int;
   var f_vector: vbool_type;

   -- INITIAL ACTIONS
   -- none

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine fw_st {
      -- state IDLE to nextstate WAITOUT
      IDLE     =>  WAITOUT {
         -- INPUT
         RF.foward_cell_in(cell,f_vector,token);

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(FW_IW,id,token);
         -- TRANSACTION DEATH INDICATOR
         -- *** that should be removed and by a more
         -- *** efficient implementation (direct gen)
         tflow_ctr.trans_death_indicator(f_vector,token);
         -- ************************************************
         ------------------------------------------------

         -- ACTIONS
         -- none
      };
      -- state WAITOUT to nextstate INT_WAITOUT1
      WAITOUT => INT_WAITOUT1{
         -- INPUT
         sync true(tx_enable.foward(f_vector) != FALSE);

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(FW_WIW1,id,token);
         -- ************************************************
         ------------------------------------------------

         -- ACTIONS
         destination = f_vector.first_f(tx_enable);
         f_vector.set(destination,FALSE);
         F_TX[destination].in_cell_out(cell,token);
      };
      -- state INT_WAITOUT1 to nextstate WAITOUT
      INT_WAITOUT1 => WAITOUT{
         -- INPUT
         sync true(f_vector.all() == TRUE);

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(FW_IW1W,id,token);
         -- ************************************************
         ------------------------------------------------
         --
         -- ACTIONS
         -- none
      };
      -- state INT_WAITOUT1 to nextstate IDLE
      INT_WAITOUT1 => IDLE{
         -- INPUT
         sync true(f_vector.all() == FALSE);

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(FW_IW1I,id,token);
         -- ************************************************
         ------------------------------------------------

         -- ACTIONS
         RF.end_foward_out();
      };
      -- any state to nextstate IDLE
      *        =>  IDLE  {
         -- INPUT
         RESET.reset_in();

         -- ACTIONS
         -- none
      };
   };
}; -- END FOWARD process
};
'>
```

D.2 Fichier MAC emodel

```
------------------------------------------------------------------------------
-- Title         : Mac emodel
-- Project       : MAC (Module d'Analyse de la Couverture)
------------------------------------------------------------------------------
-- File          : mac_emodel.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
------------------------------------------------------------------------------
-- Description   :
-- The module includes the transactional graph mapped to the e language.
-- This file must be generated from the SDL description
```

```
-------------------------------------------------------------------------------
    module: mac_emodel
    aspect: coverage
<'
    -- transactions definition (used inside the testbench)
    import fvg_transactions;
    -- mac utilities used inside the coverage module
    import mac_utilities;
    -- mac library for the tracking of the transactional flow
    import mac_tracking_lib;
    -- definition for the emodel
    import mac_emodel_def;
'>

-------------------------------------------------------------------------------
-- name: mac_emodel
-- description:
-- the transactional graph
-------------------------------------------------------------------------------
<'
struct mac_emodel {
   -----------------------------------------------------
   -- TRANSACTION FLOW CONTROLLER
   tflow_ctr: tflow_controller;

   -----------------------------------------------------
   -- CHANNEL DECLARATION
   RESET    : RESET mac_channel;
   RX_IN[4] : list of RX_IN mac_channel;
   RX_R[4]  : list of RX_R mac_channel;
   RF       : RF mac_channel;
   F_TX[4]  : list of F_TX mac_channel;
   TX_OUT[4]: list of TX_OUT mac_channel;
   MGMT_CH  : MGMT_CH mac_channel;

   -----------------------------------------------------
   -- STATE DECLARATION
   !rx_st[4] : list of utopia_rx_state;
   !tx_st[4] : list of utopia_tx_state;
   !rw_st    : rw_state;
   !fw_st    : fw_state;
   !mgmt_st  : mgmt_state;

   -----------------------------------------------------
   -- GLOBAL VARIABLE DECLARATION
   rw_table: squat_table;
   tx_enable: vbool_type;

   ------------------------------------------------------------
   --************************************************************
   -- FUNCTION THAT STARTS THE EXECUTABLE SPECIFICATION
   --************************************************************
   ------------------------------------------------------------
   transactional_graph() is {
      -- PROCESS INSTANTIATION
      for i from 0 to 3 do {
          start rx_utopia_ps(i);
          start tx_utopia_ps(i);
      };
      start rewrite_ps(0);
      start foward_ps(0);
      start mgmt_ps(0);
   };   -- transactional_graph

   ------------------------------------------------------------
   --************************************************************
   -- UTOPIA_RX process
   --************************************************************
   ------------------------------------------------------------
   rx_utopia_ps(id: int)@sys.any is {
      -- LOCAL TOKEN DECLARATION
      var token: mac_token = new;

      -- LOCAL VARIABLE DECLARATION
      var cell: UNI atm_cell;

      -- INITIAL ACTION
      RX_IN[id].in_ready_out(id);

      -- EXTENDED STATE MACHINE DESCRIPTION
      state machine rx_st[id] {
         -- state IDLE to nextstate WAITPROC
         IDLE     =>  WAITPROC   {
            -- INPUT
            RX_IN[id].uni_cell_in(cell);

            ------------------------------------------------
            -- ************COVERAGE STUFFS*****************
            -- TRANSACTION BIRTH
            token.set(cell); -- token definition
            tflow_ctr.trans_birth(id,token);
            -- TRACK DOWN THE FLOW
            tflow_ctr.add_track(RX_IW,id,token);
            -- ********************************************
            ------------------------------------------------

            -- ACTIONS
            RX_R[id].rx_valid_out();
         };
         -- state WAITPROC to nextstate IDLE
         WAITPROC =>  IDLE {
            -- INPUT
            RX_R[id].proc_ready_in();

            ------------------------------------------------
            -- ************COVERAGE STUFFS*****************
            -- TRACK DOWN THE FLOW
            tflow_ctr.add_track(RX_WI,id,token);
            -- ********************************************
            ------------------------------------------------

            -- ACTIONS
            RX_R[id].out_cell_out(cell,token);
            RX_IN[id].in_ready_out(id);
         };
         -- any state to nextstate IDLE
         *        =>  IDLE       {
            -- INPUT
            RESET.reset_in();
```

```
        -- ACTIONS
        -- NONE
      };
   };
};   -- END UTOPIA_RX process

------------------------------------------------------------
--************************************************************
-- UTOPIA_TX process
--************************************************************
------------------------------------------------------------
tx_utopia_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARTION
   var cell: NNI atm_cell;

   -- INITIAL ACTIONS
   tx_enable.set(id,TRUE);

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine tx_st[id] {
      -- state IDLE to nextstate WAITCELL
      IDLE     =>  WAITREADY   {
         -- INPUT
         F_TX[id].in_cell_in(cell,token);

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(TX_IW,id,token);
         -- ********************************************
         ------------------------------------------------

         -- ACTIONS
         tx_enable.set(id,FALSE);
      };
      -- state WAITCELL to nextstate WAITFOWARD
      WAITREADY => IDLE {
         -- INPUT
         TX_OUT[id].out_ready_in();

         ------------------------------------------------
         -- ************COVERAGE STUFFS*****************
         -- TRACK DOWN THE FLOW
         tflow_ctr.add_track(TX_WI,id,token);
         -- TRANSACTION DEATH
         tflow_ctr.trans_death(id,TX,token);
         -- ********************************************
         ------------------------------------------------

         -- ACTIONS
         TX_OUT[id].nni_cell_out(cell);
         tx_enable.set(id,TRUE);

      };
      -- any state to nextstate IDLE
      *        =>  IDLE       {
         -- INPUT
         RESET.reset_in();

         -- ACTIONS
         tx_enable.set(id,TRUE);
      };
   };
};   -- END UTOPIA_TX process

------------------------------------------------------------
--************************************************************
-- MANAGEMENT process
--************************************************************
------------------------------------------------------------
mgmt_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARATION
   var t: mgmt_trans;

   -- INITIAL ACTIONS
   -- none

   -- EXTENDED STATE MACHINE DESCRIPTION
   state machine mgmt_st {
      -- state IDLE to nextstate IDLE
      IDLE     =>  IDLE {
         -- INPUT
         MGMT_CH.mgmt_req_in(t);

         -- ACTIONS
         if t.kind == WRITE then {
            rw_table.set_data(t.addr,t.data);
         } else { -- READ
            MGMT_CH.data_out(rw_table.get_data(t.addr));
         };
      };
      -- any state to nextstate IDLE
      *        =>  IDLE  {
         -- INPUT
         RESET.reset_in();

         -- ACTIONS
         -- none
      };
   }; -- END MANAGEMENT process
};

------------------------------------------------------------
--************************************************************
-- REWRITE process
--************************************************************
------------------------------------------------------------
rewrite_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token = new;

   -- LOCAL VARIABLE DECLARATION
   var origin: int;
   var cell_in: UNI atm_cell;
```

```
var cell_out: NNI atm_cell;
var f_vector: vbool_type;
var drop_cell: int;

-- INITIAL ACTIONS
drop_cell = 0;

-- EXTENDED STATE MACHINE DESCRIPTION
state machine rw_st {
   -- state IDLE to nextstate WAITCELL
   IDLE     =>  WAITCELL   {
      -- INPUT
      first of {
         {
            RX_R[0].rx_valid_in();
            origin = 0;
         };
         {
            RX_R[1].rx_valid_in();
            origin = 1;
         };
         {
            RX_R[2].rx_valid_in();
            origin = 2;
         };
         {
            RX_R[3].rx_valid_in();
            origin = 3;
         };
      };

      -- ACTIONS
      RX_R[origin].proc_ready_out();
   };
   -- state WAITCELL to nextstate INT_WAITCELL1
   WAITCELL =>  INT_WAITCELL1{
      -- INPUT
      RX_R[origin].out_cell_in(cell_in,token);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(RW_WIW1,id,token);
      -- ********************************************
      ------------------------------------------------

      -- ACTIONS
      -- none
   };
  -- state INT_WAITCELL1 to nextstate WAITFOWARD
   INT_WAITCELL1 => WAITFOWARD {
      -- INPUT
      sync true(cell_in.check_crc() == TRUE);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(RW_IW1W,id,token);
      -- ********************************************
      ------------------------------------------------

      -- ACTIONS
      f_vector = cell_in.foward(rw_table);
      cell_out = cell_in.rewrite(rw_table);
      RF.foward_cell_out(cell_out,f_vector,token);
   };
  -- state INT_WAITCELL1 to nextstate IDLE
   INT_WAITCELL1 => IDLE {
      -- INPUT
      sync true(cell_in.check_crc() == FALSE);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(RW_IW1I,id,token);
      -- TRANSACTION DEATH
      tflow_ctr.trans_death(id,CRC,token);
      -- ********************************************
      ------------------------------------------------

      -- ACTIONS
      drop_cell += 1;

   };

   -- state WAITFOWARD to nextstate IDLE
   WAITFOWARD => IDLE      {
      -- INPUT
      RF.end_foward_in();

      -- ACTIONS
      -- none
   };
   -- any state to nextstate IDLE
   *        =>  IDLE       {
      -- INPUT
      RESET.reset_in();

      -- ACTIONS
      drop_cell = 0;
   };
};   -- END REWRITE process

------------------------------------------------------------------
--*****************************************************************
-- FOWARD process
--*****************************************************************
------------------------------------------------------------------
foward_ps(id: int)@sys.any is {
   -- LOCAL TOKEN DECLARATION
   var token: mac_token  = new;

   -- LOCAL VARIABLE DECLARTION
   var cell: NNI atm_cell;
   var destination: int;
   var f_vector: vbool_type;

   -- INITIAL ACTIONS
   -- none
```

```
-- EXTENDED STATE MACHINE DESCRIPTION
state machine fw_st {
   -- state IDLE to nextstate WAITOUT
   IDLE     =>  WAITOUT {
      -- INPUT
      RF.foward_cell_in(cell,f_vector,token);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(FW_IW,id,token);
      -- TRANSACTION DEATH INDICATOR
      -- *** that should be removed and by a more
      -- *** efficient implementation (direct gen)
      tflow_ctr.trans_death_indicator(f_vector,token);
      -- ************************************************
      ------------------------------------------------

      -- ACTIONS
      -- none
   };
   -- state WAITOUT to nextstate INT_WAITOUT1
   WAITOUT => INT_WAITOUT1{
      -- INPUT
      sync true(tx_enable.foward(f_vector) != FALSE);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(FW_WIW1,id,token);
      -- ************************************************
      ------------------------------------------------

      -- ACTIONS
      destination = f_vector.first_f(tx_enable);
      f_vector.set(destination,FALSE);
      F_TX[destination].in_cell_out(cell,token);
   };
   -- state INT_WAITOUT1 to nextstate WAITOUT
   INT_WAITOUT1 => WAITOUT{
      -- INPUT
      sync true(f_vector.all() == TRUE);

      ------------------------------------------------
      -- ************COVERAGE STUFFS*****************
      -- TRACK DOWN THE FLOW
      tflow_ctr.add_track(FW_IW1W,id,token);
      -- ************************************************
      ------------------------------------------------
      --
      -- ACTIONS
      -- none
   };
   -- state INT_WAITOUT1 to nextstate IDLE
   INT_WAITOUT1 => IDLE{
      -- INPUT
      sync true(f_vector.all() == FALSE);

      ------------------------------------------------
```

```
            -- ************COVERAGE STUFFS*****************
            -- TRACK DOWN THE FLOW
            tflow_ctr.add_track(FW_IW1I,id,token);
            -- ************************************************
            ------------------------------------------------

            -- ACTIONS
            RF.end_foward_out();
         };
         -- any state to nextstate IDLE
         *        =>  IDLE  {
            -- INPUT
            RESET.reset_in();

            -- ACTIONS
            -- none
         };
      };
   }; -- END FOWARD process
};
'>
```

D.3 Fichier MAC emodel_def

```
-------------------------------------------------------------------------------
-- Title         : Mac emodel definition
-- Project       : MAC (Module d'Analyse de la Couverture)
-------------------------------------------------------------------------------
-- File          : mac_emodel_def.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
-------------------------------------------------------------------------------
-- Description   :
-- The module includes some definitions used inside the e model of a
-- transactional graph.
-- This file must be generated from the SDL description
-- The following elements are included:
--
-- Process state type definition:
--    rx_utopia process
--    rewrite process
--    foward process
--    tx_utopia process
--    management process
--
-- Channel definition (will be generated automatically)
--    Base definiton of a channel
--       mac_channel
--    Specific channel
--      RESET   --> between the external environment and all process
--      RX_IN   --> between the external environment and the utopia_rx process
--      RX_R    --> between the utopia_rx process and the rewrite process
--      RF      --> between the rewrite process and the foward process
--      F_TX    --> between the foward process and the utopia_tx process
--      TX_OUT  --> between the utopia_tx process and the external environment
--      MGMT_CH --> between the external environment and the management process
```

```
--------------------------------------------------------------------------------

    module: mac_emodel
    aspect: coverage

--------------------------------------------------------------------------------
-- PROCESS STATE TYPE DECLARATION
--------------------------------------------------------------------------------
<'
    type utopia_rx_state:[IDLE,WAITPROC];
    type utopia_tx_state:[IDLE,WAITREADY];
    type mgmt_state     :[IDLE];
    type rw_state       :[IDLE,WAITCELL,INT_WAITCELL1,WAITFOWARD];
    type fw_state       :[IDLE,WAITOUT,INT_WAITOUT1];
'>


--------------------------------------------------------------------------------
-- name: mac_channel
-- description:
-- base implementation of a channel
-- the fifo feature of a channel is not yet implemented
--------------------------------------------------------------------------------
<'
type channel_type:[EMPTY];
struct mac_channel {
   kind: channel_type;
   FIFO: bool;
   name: string;
};
'>

--------------------------------------------------------------------------------
-- name: RESET mac_channel
-- description:
--     implementation of the RESET mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | reset           | NONE
--------------------------------------------------------------------------------
<'
extend channel_type: [RESET];
extend RESET mac_channel {
   ------------------------------------------------------
   -- Declaration of the reset signal
   ------------------------------------------------------
   !reset_ch: bool;
     -- OUTPUT function
   reset_out() is {
      reset_ch = TRUE;
   };
     -- INPUT function
   reset_in()@sys.any is {
      sync true(reset_ch);
      reset_ch = FALSE;
   };
};
'>
```

```
--------------------------------------------------------------------------------
-- name: RX_IN mac_channel
-- description:
--     implementation of the RX_IN mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | in_ready        | int
--   | uni_cell        | UNI atm_cell
--------------------------------------------------------------------------------
<'
extend channel_type: [RX_IN];
extend RX_IN mac_channel {
   ------------------------------------------------------
   -- Declaration of the in_ready signal
   ------------------------------------------------------
   in_ready: int;
   !in_ready_ch: bool;
     -- OUTPUT function
   in_ready_out(m: int) is {
      in_ready = m;
      in_ready_ch = TRUE;
   };
     -- INPUT function
   in_ready_in(m: *int)@sys.any is {
      sync true(in_ready_ch);
      in_ready_ch = FALSE;

      m = in_ready;
   };

   ------------------------------------------------------
   -- Declaration of the uni_cell signal
   ------------------------------------------------------
   uni_cell: UNI atm_cell;
   !uni_cell_ch: bool;
     -- OUTPUT function
   uni_cell_out(cell: UNI atm_cell) is {
      uni_cell = deep_copy(cell);
      uni_cell_ch = TRUE;
   };
     -- INPUT function
   uni_cell_in(cell: *UNI atm_cell)@sys.any is {
      sync true(uni_cell_ch);
      uni_cell_ch = FALSE;
      cell = deep_copy(uni_cell);
   };
};
'>

--------------------------------------------------------------------------------
-- name: RX_R mac_channel
-- description:
--     implementation of the RX_R mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
```

```
--   | proc_ready      | int
--   | cell_valid      | int
--   | out_cell        | UNI atm_cell
-------------------------------------------------------------------------------
<'
extend channel_type: [RX_R];
extend RX_R mac_channel {
   ------------------------------------------------
   -- Declaration of the proc_ready signal
   ------------------------------------------------
   !rx_valid_ch: bool;
        -- OUTPUT function
   rx_valid_out() is {
      rx_valid_ch = TRUE;
   };
        -- INPUT function
   rx_valid_in()@sys.any is {
      sync true(rx_valid_ch);
      rx_valid_ch = FALSE;
   };


   ------------------------------------------------
   -- Declaration of the proc_ready signal
   ------------------------------------------------
   !proc_ready_ch: bool;
        -- OUTPUT function
   proc_ready_out() is {
      proc_ready_ch = TRUE;
   };
        -- INPUT function
   proc_ready_in()@sys.any is {
      sync true(proc_ready_ch);
      proc_ready_ch = FALSE;
   };

   ------------------------------------------------
   -- Declaration of the out_cell signal
   ------------------------------------------------
   token1: mac_token;
   out_cell: UNI atm_cell;
   !out_cell_ch: bool;
        -- OUTPUT function
   out_cell_out(cell: UNI atm_cell,t: mac_token) is {
      out_cell = cell;
      token1 = deep_copy(t);
      out_cell_ch = TRUE;
   };
        -- INPUT function
   out_cell_in(cell: *UNI atm_cell,t: *mac_token)@sys.any is {
      sync true(out_cell_ch);
      out_cell_ch = FALSE;
      cell = deep_copy(out_cell);
      t = deep_copy(token1);
   };
};
'>

-------------------------------------------------------------------------------
-- name: RF mac_channel
```

```
-- description:
--     implementation of the RF mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | foward_cell     | NNI atm_cell, vbool_type
--   | end_foward      | NONE
-------------------------------------------------------------------------------
<'
extend channel_type: [RF];
extend RF mac_channel {
   ------------------------------------------------
   -- Declaration of the foward_cell signal
   ------------------------------------------------
   foward_cell1: NNI atm_cell;
   foward_cell2: vbool_type;
   !foward_cell_ch: bool;
   token1: mac_token;
        -- OUTPUT function
   foward_cell_out(c1: NNI atm_cell,c2: vbool_type,t: mac_token) is {
      foward_cell1 = deep_copy(c1);
      foward_cell2 = deep_copy(c2);
      token1 = deep_copy(t);
      foward_cell_ch = TRUE;
   };
        -- INPUT function
   foward_cell_in(c1: *NNI atm_cell, c2: *vbool_type,t: *mac_token)@sys.any is {
      sync true(foward_cell_ch);
      foward_cell_ch = FALSE;
      c1 = deep_copy(foward_cell1);
      c2 = deep_copy(foward_cell2);
      t = deep_copy(token1);
   };

   ------------------------------------------------
   -- Declaration of the end_foward signal
   ------------------------------------------------
   !end_foward_ch: bool;
        -- OUTPUT function
   end_foward_out() is {
      end_foward_ch = TRUE;
   };
        -- INPUT function
   end_foward_in()@sys.any is {
      sync true(end_foward_ch);
      end_foward_ch = FALSE;
   };
};
'>

-------------------------------------------------------------------------------
-- name: F_TX mac_channel
-- description:
--     implementation of the  mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | in_cell         | NNI atm_cell
```

```
--------------------------------------------------------------------------------
<'
extend channel_type: [F_TX];
extend F_TX mac_channel {
   ------------------------------------------------
   -- Declaration of the  signal
   ------------------------------------------------
   in_cell: NNI atm_cell;
   !in_cell_ch: bool;
   token1: mac_token;
        -- OUTPUT function
   in_cell_out(c: NNI atm_cell,t: mac_token) is {
      in_cell= deep_copy(c);
      token1 = deep_copy(t);
      in_cell_ch = TRUE;
   };
        -- INPUT function
   in_cell_in(c: *NNI atm_cell,t: *mac_token)@sys.any is {
      sync true(in_cell_ch);
      in_cell_ch = FALSE;
      c = deep_copy(in_cell);
      t = deep_copy(token1);
   };

};
'>

--------------------------------------------------------------------------------
-- name: TX_OUT mac_channel
-- description:
--     implementation of the TX_OUT mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | nni_cell        | NNI atm_cell
--   | out_ready       | NONE
--------------------------------------------------------------------------------
<'
extend channel_type: [TX_OUT];
extend TX_OUT mac_channel {
   ------------------------------------------------
   -- Declaration of the nni_cell signal
   ------------------------------------------------
   nni_cell: NNI atm_cell;
   !nni_cell_ch: bool;

        -- OUTPUT function
   nni_cell_out(c: NNI atm_cell) is {
      nni_cell= deep_copy(c);
      nni_cell_ch = TRUE;
   };
        -- INPUT function
   nni_cell_in(c: *NNI atm_cell)@sys.any is {
      sync true(nni_cell_ch);
      nni_cell_ch = FALSE;
      c = deep_copy(nni_cell);
   };

   ------------------------------------------------
   -- Declaration of the out_ready out_ready signal
   ------------------------------------------------
   !out_ready_ch: bool;
        -- OUTPUT function
   out_ready_out() is {
      out_ready_ch = TRUE;
   };
        -- INPUT function
   out_ready_in()@sys.any is {
      sync true(out_ready_ch);
      out_ready_ch = FALSE;
   };
};
'>

--------------------------------------------------------------------------------
-- name: MGMT_CH mac_channel
-- description:
--     implementation of the MGMT mac_channel
-- signals tansported by the channel:
--   /-------------------------------------------------------------------------
--   | Name            | The types carried by the signal
--   /-------------------------------------------------------------------------
--   | mgmt_req        | mgmt_trans
--   | data            | int
--------------------------------------------------------------------------------
<'
extend channel_type: [MGMT_CH];
extend MGMT_CH mac_channel {
   ------------------------------------------------
   -- Declaration of the mgmt_req signal
   ------------------------------------------------
   mgmt_req: mgmt_trans;
   !mgmt_req_ch: bool;
        -- OUTPUT function
   mgmt_req_out(c: mgmt_trans) is {
      mgmt_req = c;
      mgmt_req_ch = TRUE;
   };
        -- INPUT function
   mgmt_req_in(c: *mgmt_trans)@sys.any is {
      sync true(mgmt_req_ch);
      mgmt_req_ch = FALSE;
      c = mgmt_req;
   };

   ------------------------------------------------
   -- Declaration of the  signal
   ------------------------------------------------
   data : int;

   data_ch : bool;
        -- OUTPUT function
   data_out(c: int) is {
      data = c;
      data_ch = TRUE;
   };
        -- INPUT function
   data_in(c: *int)@sys.any is {
      sync true(data_ch);
```

```
      data_ch = FALSE;
      c = data ;
   };
};
'>
```

D.4 Fichier MAC tracking_lib

```
-------------------------------------------------------------------------------
-- Title         : Mac tracking definitions
-- Project       : MAC (Module d'Analyse de la Couverture)
-------------------------------------------------------------------------------
-- File          : mac_tracking_lib.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
-------------------------------------------------------------------------------
-- Description   :
-- This module includes module declartion used inside by the tric tracking library
--
-- The following elements are included inside this module:
--     * the tflow tracker
--     * the tflow controller (including the tflow sampling system)
--     * the tflow collector
-------------------------------------------------------------------------------
n
<'
    -- import the defintions for the tric tracking library
    import mac_tracking_def;
    import mac_tracking_macros;
'>

-------------------------------------------------------------------------------
-- name:         tflow_tracker
-- author:       sebastien regimbal
-- description:
-- This struct characterize a simple element of a path. A path is built with
-- several base tracks
-------------------------------------------------------------------------------
<'

struct tflow_tracker{
   parent          : tflow_controller;

   flow_id         : tflow_ps_type;
   num_id          : int;

   status          : tflow_ps_state;

   !flow           : list of base_track;
   cov_switch      : mac_cov_switch_type;

   current_track   : base_track_name_type;
   previous_track  : base_track_name_type;
   token           : mac_token; -- the token of the transaction tracked

   keep status == EMPTY;
   keep soft cov_switch == f_sequencer.get_switch(flow_id,num_id);

   event completed;

   add_track(t: base_track, sz: int, tk: mac_token) is {
      if status == COMPLETED then {
         flow.clear();
         status = EMPTY;
      }
      if status != EMPTY then {
         previous_track = current_track;
      } else {
         token = deep_copy(tk);
      };
      current_track  = t.id_name;
      flow.add(t);
      if t.track_kind == START then {
         status = INPROGRESS;
      } else if t.track_kind == END{
         status = COMPLETED;

         -- fill the flow with EMPTY if it's not full
         if flow.size() < sz then {
            var b_empty: base_track;
            while flow.size() != sz do {
                b_empty = new base_track with {
                   .id_name = EMPTY;
                };
               flow.add(b_empty);
            };
         };

         parent.collector.add_flow(flow,flow_id,token,num_id);
         emit completed;
      };
   };


   abstract_coverage(t: base_track,tk: mac_token) is {
      if t.track_kind == START then {
         token = deep_copy(tk);
         parent.collector.add_abstract_coverage(flow_id,token,num_id);
      };
   };

   cover completed is empty;
};
'>
 -------------------------------------------------------------------------------
-- name:         base_track
-- author:       sebastien regimbal
-- description:
-- This struct characterize a simple element of a path. A path is built with
-- several base tracks
-- date:         may 28, 2002
-------------------------------------------------------------------------------
<'

    tflow_ps_cover RX with RX_MAX_FLOW and RX_NB_INST;
    tflow_ps_cover RW with RW_MAX_FLOW and RW_NB_INST;
```

```
    tflow_ps_cover FW with FW_MAX_FLOW and FW_NB_INST;
    tflow_ps_cover TX with TX_MAX_FLOW and TX_NB_INST;

struct tflow_controller{

   ------------------------------------------------------------------------
   -- FLOW TRACKING STUFFS
   ------------------------------------------------------------------------
   -- Declaration of a path tracker for process instance
   utopia_rx[RX_NB_INST]: list of RX tflow_tracker;
   rewrite[RW_NB_INST]  : list of RW tflow_tracker;
   foward[FW_NB_INST]   : list of FW tflow_tracker;
   utopia_tx[RX_NB_INST]: list of TX tflow_tracker;

   keep for each (p) in utopia_rx {
      p.parent     == me;
      p.flow_id    == RX;
      p.num_id     == index;
      p.current_track == append("RX_IW_",index).as_a(base_track_name_type);
   };
   keep for each (p) in rewrite{
      p.parent     == me;
      p.flow_id    == RW;
      p.num_id     == 0;
      p.current_track == append("RW_IW_",index).as_a(base_track_name_type);
   };
   keep for each (p) in foward{
      p.parent     == me;
      p.flow_id    == FW;
      p.num_id     == 0;
      p.current_track == append("FW_IW_",index).as_a(base_track_name_type);
   };
   keep for each (p) in utopia_tx {
      p.parent     == me;
      p.flow_id    == TX;
      p.num_id     == index;
      p.current_track == append("TX_IW_",index).as_a(base_track_name_type);
   };

   -- Function used to add a track to a path
   add_track(t: base_track_name_type,p_id: int,token: mac_token) is {
      var new_track: base_track  = deep_copy(bts_global.key(t));
      new_track.refine_track(p_id);
      case new_track.tflow_ps_kind {
         RX: {
            if utopia_rx[p_id].cov_switch == ON then {
               utopia_rx[p_id].add_track(new_track,RX_MAX_FLOW,token);
            } else {
              utopia_rx[p_id].abstract_coverage(new_track,token);
            };
         };
         RW: {
            if rewrite[p_id].cov_switch == ON then {
               rewrite[p_id].add_track(new_track,RW_MAX_FLOW,token);
            } else {
              rewrite[p_id].abstract_coverage(new_track,token);
            };
         };
         FW: {
            if foward[p_id].cov_switch == ON then {
               foward[p_id].add_track(new_track,FW_MAX_FLOW,token);
            } else {
              foward[p_id].abstract_coverage(new_track,token);
            };
         };
         TX: {
            if utopia_tx[p_id].cov_switch == ON then {
               utopia_tx[p_id].add_track(new_track,TX_MAX_FLOW,token);
            } else {
              utopia_tx[p_id].abstract_coverage(new_track,token);
            };
         };
      };
   };
};

------------------------------------------------------------------------
-- TRANSACTION BIRTH AND DEATH TRACKER
------------------------------------------------------------------------
token_bad_crc: int;
token_tx0: int;
token_tx1: int;
token_tx2: int;
token_tx3: int;
cur_token: int;

!tx_indicator: list(key: token_id) of vbool_token_type;
event t_completed;
trans_birth(ps_id: int,token: mac_token) is {
   start transaction_live_handler(token.id);
};

trans_death(ps_id:int,td: mac_death_type,token: mac_token) is {
   case td {
      CRC:{
         token_bad_crc = token.id;
      };
      TX:{
         case ps_id{
            0:{
               token_tx0 = token.id;
            };
            1:{
               token_tx1 = token.id;
            };
            2:{
               token_tx2 = token.id;
            };
            3:{
               token_tx3 = token.id;
            };
         };
      };
   };
};

trans_death_indicator(indicator: vbool_type,token: mac_token) is {
   var ind: vbool_token_type = new vbool_token_type;
   gen ind keeping {
      .value == indicator.value;
      .token_id == token.id;
```

```
      };
      tx_indicator.add(ind);
   };

   transaction_live_handler(token:int)@sys.any is {
      first of {
         {
            -- wait for a CRC death
            wait true(token == token_bad_crc);
         };
         {
            -- wait for transmissions
            wait true(tx_indicator.key(token)!= NULL);
            var v := tx_indicator.key(token).value;
            all of {
               {
                  if v[0] == TRUE then {
                     wait true(token == token_tx0);
                  };
               };
               {
                  if v[1] == TRUE then {
                     wait true(token == token_tx1);
                  };
               };
               {
                  if v[2] == TRUE then {
                     wait true(token == token_tx2);
                  };
               };
               {
                  if v[3] == TRUE then {
                     wait true(token == token_tx3);
                  };
               };
            };
            tx_indicator.delete(tx_indicator.key_index(token));
         };
      };
      cur_token = token;
      emit collector.cov_collect;
   };

   ----------------------------------------------------------------------------
   -- COVERAGE EXTRACTION STUFFS
   ----------------------------------------------------------------------------
   collector: tflow_collector;
   keep collector.parent == me;
};
'>



-------------------------------------------------------------------------------
-- name:         tflow_collector
-- author:       sebastien regimbal
-- description:
--
-------------------------------------------------------------------------------
```

```
<'
struct tflow_collector{
   parent : tflow_controller;

   flow: fifo_tflow_ps;
   !trans_flow: tflow;

   add_flow(p:list of base_track,t: tflow_ps_type,tk:mac_token,ps_num:int) is{
      var t_p: tflow_ps = new;
      t_p.set(p,t,tk,ps_num);
      flow.add(t_p);
   };

   add_abstract_coverage(t: tflow_ps_type,tk:mac_token,ps_num:int) is{
      var t_p: tflow_ps = new;
      t_p.set_abstract(t,tk,ps_num);
      flow.add(t_p);
   };

   get_flow(tr: int)is {
      trans_flow = new;
      var flow_t: list of tflow_ps;
      flow_t = flow.get_same(tr);
      trans_flow.set(flow_t);
   };

   event cov_collect;
   on cov_collect {
      get_flow(parent.cur_token);
   };

   cover cov_collect is empty;
};

build_tflow_cover FILE;
'>
```

D.5 Fichier MAC tracking_def

```
-------------------------------------------------------------------------------
-- Title         : Mac tracking definitions
-- Project       : MAC (Module d'Analyse de la Couverture)
-------------------------------------------------------------------------------
-- File          : mac_tracking_def.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
-------------------------------------------------------------------------------
-- Description   :
-- This module includes definitions used inside by the tric tracking library
--
-- The following elements are included inside this module:
--    * types and constants definition for the tracking
--    * base track definition
--    * transactional flow (process part) definition
--    * transactional flow definition
--    * token definition
```

```
--    * Algorithm to construct the base track repositary and the transactional
--      flow dictionary
-------------------------------------------------------------------------------


-------------------------------------------------------------------------------
-- name: none
-- description:
--     types and constants defintion for the tric tracking library
-------------------------------------------------------------------------------
<'
    -- BASE TRACK TYPE DEFINITION
    -- the type of the base track
    type base_track_type:[START,END,INTERNAL,NONE];
    -- identification name for each base track
    type base_track_name_type:[
       EMPTY, ENV,
       -- rx utopia process
       RX_IW, RX_IW_0, RX_IW_1, RX_IW_2, RX_IW_3,
       RX_WI, RX_WI_0, RX_WI_1, RX_WI_2, RX_WI_3,
       -- rewrite process track
       RW_IW,   RW_IW_0,
       RW_WIW1, RW_WIW1_0,
       RW_IW1I, RW_IW1I_0,
       RW_IW1W, RW_IW1W_0,
       RW_WI,   RW_WI_0,
       -- foward process
       FW_IW,   FW_IW_0,
       FW_WIW1, FW_WIW1_0,
       FW_IW1W, FW_IW1W_0,
       FW_IW1I, FW_IW1I_0,
       -- tx utopia process
       TX_IW, TX_IW_0, TX_IW_1, TX_IW_2, TX_IW_3,
       TX_WI, TX_WI_0, TX_WI_1, TX_WI_2, TX_WI_3];

    -- TRANSACTIONAL FLOW TYPE DEFINITION
    -- which process the transactional flow come from
    type tflow_ps_type: [RX,RW,FW,TX,ENV];
    -- which process is the state of the transactional flow
    type tflow_ps_state  : [EMPTY,INPROGRESS,COMPLETED];
    -- identification name for each transactionnal flow
    type tflow_ps_name_type:[
       EMPTY,ABSTRACT_COVER,
       -- <Process>_P_<path_number>_<process_number>
       -- rx utopia process
       RX_ABSTRACT_COVER, RX_ABSTRACT_COVER_0, RX_ABSTRACT_COVER_1,
       RX_ABSTRACT_COVER_2, RX_ABSTRACT_COVER_3,
       RX_P_0, RX_P_0_0, RX_P_0_1, RX_P_0_2, RX_P_0_3,
       -- rewrite process track
       RW_ABSTRACT_COVER, RW_ABSTRACT_COVER_0,
       RW_P_0, RW_P_0_0,
       RW_P_1, RW_P_1_0,
       -- foward process
       FW_ABSTRACT_COVER, FW_ABSTRACT_COVER_0,
       FW_P_0, FW_P_0_0,
       FW_P_1, FW_P_1_0,
       FW_P_2, FW_P_2_0,
       FW_P_3, FW_P_3_0,
       -- tx utopia process
       TX_ABSTRACT_COVER, TX_ABSTRACT_COVER_0, TX_ABSTRACT_COVER_1,
       TX_ABSTRACT_COVER_2, TX_ABSTRACT_COVER_3,
       TX_P_0, TX_P_0_0, TX_P_0_1, TX_P_0_2, TX_P_0_3];

    -- TRIC TOKEN TYPE
    -- which kind of transaction is involved with the token
    type mac_token_type : [UNI,NNI,MGMT];

    -- TRIC COVERAGE SWITCH
    -- to enable coverage on each tflow_tracker
    type mac_cov_switch_type : [ON,OFF];

    -- CONSTANTS DEFINITION
    -- Longest transactional flow inside each process
    define RX_MAX_FLOW      2;
    define RW_MAX_FLOW      2;
    define FW_MAX_FLOW      9;
    define TX_MAX_FLOW      2;
    -- Number of instance of each process
    define RX_NB_INST       4;
    define RW_NB_INST       1;
    define FW_NB_INST       1;
    define TX_NB_INST       4;
    -- Number of parallel instance of each process
    define RX_NB_MIN_PAR       1;
    define RX_NB_MAX_PAR       1;
    define RW_NB_MIN_PAR       1;
    define RW_NB_MAX_PAR       1;
    define FW_NB_MIN_PAR       0;
    define FW_NB_MAX_PAR       1;
    define TX_NB_MIN_PAR       0;
    define TX_NB_MAX_PAR       4;


    -- Abstraction Defines
    define ABS_GLOBAL  ON;
    define ABS_RX      OFF;
    define ABS_RX_PS   ON;

    define ABS_RW_FW   ON;
    define ABS_RW_PS   ON;
    define ABS_FW_PS   ON;

    define ABS_TX      ON;
    define ABS_TX_PS   ON;

    type mac_death_type: [CRC,TX];
    struct vbool_token_type like vbool_type {
       token_id: int;
    };
'>

-------------------------------------------------------------------------------
-- name: base_track
-- description:
-- This struct characterize a simple element of a transactional flow.
-- A transactional flow is built with several base tracks
-------------------------------------------------------------------------------
<'
struct base_track{
   -- the process where the base track can be involved
   tflow_ps_kind  : tflow_ps_type;
```

```
   -- the role of the base track inside a process tflow
   track_kind : base_track_type;
   -- the name of the base track (string format)
   name       : string;
   -- the identification name of the base track
   id_name    : base_track_name_type;
   -- lists used to know which are the legal previous
   -- and next track of this base track
   !internal_prev: list of base_track_name_type;
   !internal_next: list of base_track_name_type;
   !external_prev: list of base_track_name_type;
   !external_next: list of base_track_name_type;

   -- Refine the track for multiple instance of a process
   refine_track(p:int) is {
      id_name = (id_name.as_a(int)+p+1).as_a(base_track_name_type);
   };
   -- function used to configure the legal internal prev and next track
   set_internal_prev(l:list of base_track_name_type) is {
      internal_prev = l;
   };
   set_internal_next(l:list of base_track_name_type) is {
      internal_next = l;
   };
   get_internal_prev():list of base_track_name_type is {
      return internal_prev;
   };
   get_internal_next():list of base_track_name_type is {
      return internal_next;
   };
   -- function used to configure the legal external prev and next track
   set_external_prev(l:list of base_track_name_type) is {
      external_prev = l;
   };
   set_external_next(l:list of base_track_name_type) is {
      external_next = l;
   };
   get_external_prev():list of base_track_name_type is {
      return external_prev;
   };
   get_external_next():list of base_track_name_type is {
      return external_next;
   };
};
'>

--------------------------------------------------------------------------------
-- name: tflow_ps
-- description:
-- The part of a transactional flow involved inside a specific process
--------------------------------------------------------------------------------
<'
struct tflow_dictionary {
   !dict: list of tflow_ps;

   get_same_type(t: tflow_ps_type): list of tflow_ps is {
      result = dict.all(it.tflow_ps == t);
   };

   add(t: tflow_ps) is {
      dict.add(t);
   };

   get_flowid(l:list of base_track_name_type):tflow_ps_name_type is {
      result =  (dict.first(it.flow_bs == l)).tflow_ps_name;
   };

   get_flowid_name(t:tflow_ps_type): list of tflow_ps_name_type is {
      var temp: list of tflow_ps_name_type;
      for each (f)  in dict.all(it.tflow_ps == t and it.abstract == FALSE) do {
         temp.add(f.tflow_ps_name);
      };
      return temp;
   };

   get_next_flow(t: tflow_ps_name_type): list of tflow_ps_name_type is {
      var l: list of tflow_ps_name_type;
      var temp_bt: base_track_name_type;
      var temp_bt_list: list of base_track_name_type;
      var temp_tflow_def: tflow_ps;
      var temp_tflow_def_list: list of tflow_ps;
      if t == EMPTY then {
         l.add(EMPTY);
      }else{
         temp_tflow_def = dict.first(it.tflow_ps_name == t);
         if temp_tflow_def.abstract == FALSE then {
            temp_bt =  temp_tflow_def.flow_bs.last(TRUE);
            temp_bt_list = bts_global.key(temp_bt).get_external_next();
            l.add(all_flow(temp_bt_list));
         } else {
            temp_tflow_def_list = dict.all(it.tflow_ps == temp_tflow_def.tflow_ps
and it.abstract == FALSE);
            for each (tf) in temp_tflow_def_list do {
               temp_bt =  tf.flow_bs.last(TRUE);
               temp_bt_list = bts_global.key(temp_bt).get_external_next();
               l.add(all_flow(temp_bt_list));
            };
         };
      };
      return l.unique(it);
   };

   all_flow(tbt_l: list of base_track_name_type):list of tflow_ps_name_type is {
      var l: list of tflow_ps_name_type;

      for each (f) in dict do {
         for each (t) in tbt_l do {
            if f.abstract == FALSE then {
               if t == EMPTY then {
                  l.add(EMPTY);
               } else if f.flow_bs[0] == t then {
                  l.add(f.tflow_ps_name);
               };
            };
         };
      };
      return l;
   };
};
```

```
-------------------------------------------------------------------------------
-- name: tflow_def
-- description:
--
-------------------------------------------------------------------------------
struct tflow_def {
   -- the list of base track identification name inside the tflow_ps
   !flow_bs: list of base_track_name_type;
   -- the identification name of the tflow_ps
   tflow_ps_name: tflow_ps_name_type;
   -- the identification name of the process
   tflow_ps: tflow_ps_type;
};

-------------------------------------------------------------------------------
-- name: tflow_ps
-- description:
--
-------------------------------------------------------------------------------
struct tflow_ps like  tflow_def{
   -- the list of base track inside the tflow_ps
   flow: list of base_track;
   -- the identification number of the process
   process_id: int;
   -- the token used with this specific tflow_ps
   token: mac_token;

   !abstract : bool;

   -- a method used to set the tflow_ps
   set(f: list of base_track,ps_n: tflow_ps_type,t: mac_token,ps_id:int) is {
      var btn_temp: list of base_track_name_type;
      for each (b) in f do {
         if b.id_name != EMPTY then {
            btn_temp.add(str_replace(b.id_name.as_a(string),
                                     "/(.*)_(.*)_(.*)/",
                                     "\1_\2").as_a(base_track_name_type));
         };
      };
      abstract = FALSE;
      flow = f;
      token = t;
      tflow_ps = ps_n;
      process_id = ps_id;
      tflow_ps_name = append((get_flowid_global(btn_temp)).as_a(string),
                        "_",process_id.as_a(string)).as_a(tflow_ps_name_type);
   };

   -- a method used to set an abtract tflow_ps
   set_abstract(ps_n: tflow_ps_type,t: mac_token,ps_id:int) is {
      token = t;
      tflow_ps = ps_n;
      process_id = ps_id;
      abstract = TRUE;
      tflow_ps_name = append(ps_n.as_a(string),"_ABSTRACT_COVER_",
                             ps_id).as_a(tflow_ps_name_type);
   };
};
'>
```

```
-------------------------------------------------------------------------------
-- name: tflow_ps
-- description:
-- The transactional flow covered by the coverage analysis module
-------------------------------------------------------------------------------
<'
struct tflow{
   -- lists used to keep all tflow_ps of the complete transactional flow
   RX_flow[RX_NB_MAX_PAR]: list of tflow_ps_name_type;
   RW_flow[RW_NB_MAX_PAR]: list of tflow_ps_name_type;
   FW_flow[FW_NB_MAX_PAR]: list of tflow_ps_name_type;
   TX_flow[TX_NB_MAX_PAR]: list of tflow_ps_name_type;
   keep for each (p) in RX_flow {
      p == EMPTY;
   };
   keep for each (p) in RW_flow {
      p == EMPTY;
   };
   keep for each (p) in FW_flow {
      p == EMPTY;
   };
   keep for each (p) in TX_flow {
      p == EMPTY;
   };

   -- method used to set the transaction flow with a list of tflow_ps
   set(f: list of tflow_ps) is {
      if !f.is_empty() then {
         for each (f_ps) in f do {
            case f_ps.tflow_ps {
               RX: {
                  RX_flow[0] = f_ps.tflow_ps_name;
               };
               RW: {
                  RW_flow[0] = f_ps.tflow_ps_name;
               };
               FW: {
                  FW_flow[0] = f_ps.tflow_ps_name;
               };
               TX: {
                  TX_flow[f_ps.process_id] = f_ps.tflow_ps_name;
               };
            };
         };
      };
   };
};
'>

-------------------------------------------------------------------------------
-- name: mac_token
-- description:
-- the tric token is used to follow a transaction inside the transactional graph
-- with the aim to track the transactional flow
-------------------------------------------------------------------------------
<'
type tflow_hierarchy_type: [SEQUENTIAL, CONCURRENT,TERMINAL];
struct tflow_hierarchy {
   -- general stuffs
   !kind              : tflow_hierarchy_type;
```

```
!switch            : mac_cov_switch_type;
!mandatory_element : bool;
!id  : int;

-- sequential and concurrent stuffs
!f: list of tflow_hierarchy;
!nb_element  : int;

-- concurrent stuffs
!min_parallel: int;
!max_parallel: int;


-- terminal
!name: tflow_ps_type;

get_deep_mandatory(): bool is {
   result = TRUE;
   case kind {
      SEQUENTIAL: {
         result = (f.last(TRUE)).get_deep_mandatory();
      };
      CONCURRENT: {
         result = mandatory_element;
      };
      TERMINAL: {
         result = mandatory_element;
      };
   };
};
get_length():int is {
   result =  0;
   case kind {
      SEQUENTIAL: {
         for each (tf) in f do {
            result += tf.get_length();
         };
      };
      CONCURRENT: {
         for each (tf) in f do {
            if index == 0 then {
               result  = tf.get_length();
            } else if tf.get_length() > result then {
               result  = tf.get_length();
            };
         };
      };
      TERMINAL: {
         result  = 1;
      };
   };
};
add(t: tflow_hierarchy) is {
   f.add(t);
};

get_switch(n: tflow_ps_type, i :int): mac_cov_switch_type is {
   var temp_switch: mac_cov_switch_type;
   if find_abs_switch(n,i,temp_switch) == TRUE then {
      return temp_switch;
   } else {
      error("Coverage Abstraction Switch is not found: \nName: ",
            n,"\nId: ",i);
   };
};

find_abs_switch(n: tflow_ps_type, i :int, s: *mac_cov_switch_type): bool is {
   if kind == TERMINAL then {
      if name == n and id == i then {
         s = switch;
         return TRUE;
      } else {
         return FALSE;
      };
   } else {
      for each (ts) in f do {
         if ts.find_abs_switch(n,i,s) == TRUE then {
            return TRUE;
         };
      };
   };
   return FALSE;
};
};
'>

--------------------------------------------------------------------------------
-- name: mac_token
-- description:
-- the tric token is used to follow a transaction inside the transactional graph
-- with the aim to track the transactional flow
--------------------------------------------------------------------------------
<'
struct mac_token{
   -- the unique number taht identify the token
   id   : int;
   -- the type of token (same as the transaction type)
   kind : mac_token_type;

   -- A method to set the parameters of the token
   set(c: atm_cell) is {
      var temp_bytes: list of byte;
      temp_bytes = pack(NULL,c);
      kind = c.kind.as_a(mac_token_type);
      id = temp_bytes.crc_32(0,temp_bytes.size());
   };
};
'>

--------------------------------------------------------------------------------
-- name: extention of the global struct of specman elite
-- description:
-- here we define the base struct repository and the transactional flow dictionary
--------------------------------------------------------------------------------
<'
    build_struct fifo_tflow_ps;
extend fifo_tflow_ps{
   get_start(): list of tflow_ps is {
      var tflow: list of tflow_ps;
      tflow.add(pop());
```

```
      --tflow.add(entries.all(tflow[0].token.id == it.token.id));
      return tflow;
   };
   get_same(id: int): list of tflow_ps  is {
      result = entries.all(id == it.token.id);
      entries = entries.all(id != it.token.id);
   };
};
'>

-------------------------------------------------------------------------------
-- name: extention of the global struct of specman elite
-- description:
-- here we define the base struct repository and the transactional flow dictionary
-------------------------------------------------------------------------------
<'
    -- utility class used for combinatory manipulation
struct comb_vect {
   !l:list of bool;
   get():list of bool is {
      return l;
   };
   add(b: list of bool) is {
      l.add(b);
   };
};

extend global{
   -- all the base used by the transactional graph
   !bts_global: list(key:id_name) of  base_track;
   -- the transaction flow dictionary of the trasnactional graph
   !pdict_global: tflow_dictionary;
   -- The sequencer of the transactional graph
   !f_sequencer: tflow_hierarchy;

   -- method used to initiate bt repository and tflow dictionary
   init() is also {
      var unique_id: int = 0;
      var bt: base_track;

      ------------------------------------------------------
      -- Generation of all the base track
      ------------------------------------------------------
      ------------------------------------------------------
      -- RX utopia process
      ------------------------------------------------------
      // -- rx_idle_2_waitproc
      bt = new base_track with {
         .tflow_ps_kind = RX;
         .track_kind = START;
         .id_name = RX_IW;
         .name    = "rx_idle_2_waitproc";
      };
      unique_id += 1;
      bts_global.add(bt);
      -- rx_waitproc_2_idle
      bt = new base_track with {
         .tflow_ps_kind = RX;
         .track_kind = END;
         .id_name = RX_WI;
         .name    = "rx_waitproc_2_idle";
      };
      unique_id += 1;
      bts_global.add(bt);
      ------------------------------------------------------
      -- Rewrite process
      -- rw_idle_2_waitcell
      ------------------------------------------------------
      bt = new base_track with {
         .tflow_ps_kind = RW;
         .track_kind = INTERNAL;
         .id_name = RW_IW;
         .name    = "rw_idle_2_waitcell";
      };
      unique_id += 1;
      bts_global.add(bt);
      -- rw_waitcell_int_waitcell1
      bt = new base_track with {
         .tflow_ps_kind = RW;
         .track_kind = START;
         .id_name = RW_WIW1;
         .name    = "rw_waitcell_int_waitcell1";
      };
      unique_id += 1;
      bts_global.add(bt);
      -- rw_int_waitcell1_idle
      bt = new base_track with {
         .tflow_ps_kind = RW;
         .track_kind = END;
         .id_name = RW_IW1I;
         .name    = "rw_int_waitcell1_idle";
      };
      unique_id += 1;
      bts_global.add(bt);
      -- rw_int_waitcell1_waitfoward
      bt = new base_track with {
         .tflow_ps_kind = RW;
         .track_kind = END;
         .id_name = RW_IW1W;
         .name    = "rw_int_waitcell1_waitfoward";
      };
      unique_id += 1;
      bts_global.add(bt);
      -- waitfoward_idle
      bt = new base_track with {
         .tflow_ps_kind = RW;
         .track_kind = INTERNAL;
         .id_name = RW_WI;
         .name    = "rw_waitfoward_idle";
      };
      unique_id += 1;
      bts_global.add(bt);
      ------------------------------------------------------
      -- Foward process
      ------------------------------------------------------
      -- fw_idle_2_waitout
      bt = new base_track with {
         .tflow_ps_kind = FW;
         .track_kind = START;
         .id_name = FW_IW;
```

```
    .name    = "fw_idle_2_waitout";
  };
  unique_id += 1;
  bts_global.add(bt);
  -- fw_waitout_2_int_waitout1
  bt = new base_track with {
    .tflow_ps_kind = FW;
    .track_kind = INTERNAL;
    .id_name = FW_WIW1;
    .name    = "fw_waitout_2_int_waitout1";
  };
  unique_id += 1;
  bts_global.add(bt);
  -- fw_int_waitout1_2_waitout
  bt = new base_track with {
    .tflow_ps_kind = FW;
    .track_kind = INTERNAL;
    .id_name = FW_IW1W;
    .name    = "fw_int_waitout1_2_waitout";
  };
  unique_id += 1;
  bts_global.add(bt);
  -- fw_int_waitout1_2_idle
  bt = new base_track with {
    .tflow_ps_kind = FW;
    .track_kind = END;
    .id_name = FW_IW1I;
    .name    = "fw_int_waitout1_2_idle";
  };
  unique_id += 1;
  bts_global.add(bt);
  ------------------------------------------------------------
  -- TX utopia process
  ------------------------------------------------------------
  -- tx_idle_2_waitready
  bt = new base_track with {
    .tflow_ps_kind = TX;
    .track_kind = START;
    .id_name = TX_IW;
    .name    = "tx_idle_2_waitready";
  };
  unique_id += 1;
  bts_global.add(bt);
  -- tx_waitready_2_idle
  bt = new base_track with {
    .tflow_ps_kind = TX;
    .track_kind = END;
    .id_name = TX_WI;
    .name    = "tx_waitready_2_idle";
  };
  unique_id += 1;
  bts_global.add(bt);

  -- Initialization of the legal next and previous bt
  init_legal_next_prev_bt();
  -- Initialization of the path dictionary
  init_path_dict();
  -- Initialization of the tflow_hierarchy
  init_hierarchy();
};
```

```
init_hierarchy() is{
  var fh_term       : tflow_hierarchy;
  var fh_term_lst   : list of tflow_hierarchy;
  var fh_conc       : tflow_hierarchy;
  var fh_seq        : tflow_hierarchy;
  var fh_seq_lst1   : list of tflow_hierarchy;
  var fh_seq_lst2   : list of tflow_hierarchy;
  -- RX process init
  fh_term_lst.clear();
  -- construct the terminal element for each RX process
  for i from 0 to (RX_NB_INST-1) do {
    fh_term_lst.add(deep_copy(create_th_terminal(RX,i,FALSE,ABS_RX_PS)));
  };

  -- Create hierarchical element for RX process
  fh_conc = new;
  fh_conc = deep_copy(create_th_concurrent(fh_term_lst,0,TRUE,
                                           RX_NB_MIN_PAR,
                                           RX_NB_MAX_PAR,ABS_RX));

  -- add the concurrent element to the sequential list
  fh_seq_lst1.add(fh_conc);



  -- we will create a sequence with the process RW and FW
  -- RW process init
  -- construct the terminal element the RW process
  fh_term = new;
  fh_term = deep_copy(create_th_terminal(RW,0,TRUE,ABS_RW_PS));
  -- add the terminal element to the sequential list
  fh_seq_lst2.add(fh_term);

  -- FW process init
  -- construct the terminal element the FW process
  fh_term = new;
  fh_term = deep_copy(create_th_terminal(FW,0,FALSE,ABS_FW_PS));
  -- add the terminal element to the sequential list
  fh_seq_lst2.add(fh_term);

  -- Create hierarchical element for each RW and FW  process
  fh_seq = new;
  fh_seq = deep_copy(create_th_sequential(fh_seq_lst2,0,TRUE,ABS_RW_FW));

  -- add the sequential element to the sequential list
  fh_seq_lst1.add(fh_seq);

  -- TX process init
  fh_term_lst.clear();
  -- construct the terminal element for each TX process
  for i from 0 to (TX_NB_INST-1) do {
    fh_term_lst.add(deep_copy(create_th_terminal(TX,i,FALSE,ABS_TX_PS)));
  };

  -- Create hierarchical element for each TX process
  fh_conc = new;
  fh_conc = deep_copy(create_th_concurrent(fh_term_lst,0,FALSE,
                                           TX_NB_MIN_PAR,
                                           TX_NB_MAX_PAR,ABS_TX));
```

```
   -- add the concurrent element to the sequential list
   fh_seq_lst1.add(fh_conc);


    -- final init
   f_sequencer = new;
   f_sequencer = deep_copy(create_th_sequential(fh_seq_lst1,0,TRUE,ABS_GLOBAL));
};

create_th_terminal(n: tflow_ps_type,i: int,mandatory: bool,
                   cov_s: mac_cov_switch_type): tflow_hierarchy is {
   var term_ts: tflow_hierarchy = new tflow_hierarchy  with {
      .kind = TERMINAL;
      .switch = cov_s;
      .name = n;
      .id = i;
      .mandatory_element = mandatory;
   };
   return term_ts;
};

create_th_sequential(ts: list of tflow_hierarchy,i:int,mandatory: bool,
                     cov_s: mac_cov_switch_type): tflow_hierarchy is {
   var seq_ts: tflow_hierarchy = new tflow_hierarchy  with {
      .kind = SEQUENTIAL;
      .switch = cov_s;
      .id = i;
      .mandatory_element = mandatory;
   };
   for each (t) in ts do {
      seq_ts.add(t);
   };
   seq_ts.nb_element = seq_ts.f.size();
   return seq_ts;
};

create_th_concurrent(ts: list of tflow_hierarchy, i:int,mandatory: bool,
                     min_p: int,max_p: int,
                     cov_s: mac_cov_switch_type): tflow_hierarchy is {
   var concurrent_th: tflow_hierarchy = new tflow_hierarchy  with {
      .kind = CONCURRENT;
      .min_parallel = min_p;
      .max_parallel = max_p;
      .id = i;
      .mandatory_element = mandatory;
      .switch = cov_s;

   };
   for each (t) in ts do {
      concurrent_th.add(t);
   };
   concurrent_th.nb_element = concurrent_th.f.size();
   return concurrent_th;
};



-- method used to set the legal next and previous base track
-- inside the base repository
```

```
init_legal_next_prev_bt() is {
   ------------------------------------------------
   -- adding legal previous and next tracks
   -- the order of the rack is very important for the
   -- costruction of the paths
   ------------------------------------------------
   -- Utopia Rx process
   bts_global.key(RX_IW).set_internal_prev({RX_WI});
   bts_global.key(RX_IW).set_internal_next({RX_WI});
   bts_global.key(RX_IW).set_external_prev({ENV});

   bts_global.key(RX_WI).set_internal_prev({RX_IW});
   bts_global.key(RX_WI).set_internal_next({RX_IW});
   bts_global.key(RX_WI).set_external_next({RW_WIW1});
   -- Rewrite Process
   bts_global.key(RW_IW).set_internal_prev({RW_IW1I;RW_WI});
   bts_global.key(RW_IW).set_internal_next({RW_WIW1});

   bts_global.key(RW_WIW1).set_internal_prev({RW_IW});
   bts_global.key(RW_WIW1).set_internal_next({RW_IW1I;RW_IW1W});
   bts_global.key(RW_WIW1).set_external_prev({RX_WI});

   bts_global.key(RW_IW1I).set_internal_prev({RW_WIW1});
   bts_global.key(RW_IW1I).set_internal_next({RW_IW});
   bts_global.key(RW_IW1I).set_external_next({EMPTY});

   bts_global.key(RW_IW1W).set_internal_prev({RW_WIW1});
   bts_global.key(RW_IW1W).set_internal_next({RW_WI});
   bts_global.key(RW_IW1W).set_external_next({FW_IW});

   bts_global.key(RW_WI).set_internal_prev({RW_IW1W});
   bts_global.key(RW_WI).set_internal_next({RW_IW});

   -- Foward Process
   bts_global.key(FW_IW).set_internal_prev({FW_IW1I});
   bts_global.key(FW_IW).set_internal_next({FW_WIW1});
   bts_global.key(FW_IW).set_external_prev({RW_IW1I});

   bts_global.key(FW_WIW1).set_internal_prev({FW_IW;FW_IW1W});
   bts_global.key(FW_WIW1).set_internal_next({FW_IW1I;FW_IW1W});

   bts_global.key(FW_IW1W).set_internal_prev({FW_WIW1});
   bts_global.key(FW_IW1W).set_internal_next({FW_WIW1});

   bts_global.key(FW_IW1I).set_internal_prev({FW_WIW1});
   bts_global.key(FW_IW1I).set_internal_next({FW_IW});
   bts_global.key(FW_IW1I).set_external_next({TX_IW});
   -- Utopia Tx Process
   bts_global.key(TX_IW).set_internal_prev({TX_WI});
   bts_global.key(TX_IW).set_internal_next({TX_WI});
   bts_global.key(TX_IW).set_external_prev({FW_IW1I});

   bts_global.key(TX_WI).set_internal_prev({TX_IW});
   bts_global.key(TX_WI).set_internal_next({TX_IW});
   bts_global.key(TX_WI).set_external_next({ENV});
};

-- method used to define the tflow dictionary
init_path_dict() is {
   var bt: list (key:id_name) of base_track;
```

```
var t_p: tflow_ps;
var the_path: string;
var id: int = 0;
bt = bts_global.all(it.track_kind == START);
pdict_global = new;

for each (b) in bt do {
   t_p = new tflow_ps;
   t_p.flow_bs.add(b.id_name);
   t_p.tflow_ps = b.tflow_ps_kind;
   expand_flow(b,t_p);
};

var pd_temp: list of tflow_ps;
pd_temp = pdict_global.get_same_type(RX);
for each (d) in pd_temp do {
   the_path = append(d.tflow_ps.as_a(string),
                     "_p_",
                     (index).as_a(string));
   d.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
   d.abstract = FALSE;
};
pd_temp = pdict_global.get_same_type(RW);
for each (d) in pd_temp do {
   the_path = append(d.tflow_ps.as_a(string),
                     "_p_",
                     (index).as_a(string));
   d.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
   d.abstract = FALSE;
};
pd_temp = pdict_global.get_same_type(FW);
for each (d) in pd_temp do {
   the_path = append(d.tflow_ps.as_a(string),
                     "_p_",
                     (index).as_a(string));
   d.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
   d.abstract = FALSE;
};
pd_temp = pdict_global.get_same_type(TX);
for each (d) in pd_temp do {
   the_path = append(d.tflow_ps.as_a(string),
                     "_p_",
                     (index).as_a(string));
   d.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
   d.abstract = FALSE;
};

-- build abtract flow for each instance of a process
-- RX constructi

t_p = new;
the_path = append("RX_ABSTRACT_COVER");
t_p.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
t_p.tflow_ps = RX;
t_p.abstract = TRUE;
pdict_global.add(t_p);

-- RW construction

t_p = new;
the_path = append("RW_ABSTRACT_COVER");
t_p.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
t_p.tflow_ps = RW;
t_p.abstract = TRUE;
pdict_global.add(t_p);

-- FW construction

t_p = new;
the_path = append("FW_ABSTRACT_COVER");
t_p.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
t_p.tflow_ps = FW;
t_p.abstract = TRUE;
pdict_global.add(t_p);

-- TX construction

t_p = new;
the_path = append("TX_ABSTRACT_COVER");
t_p.tflow_ps_name = the_path.as_a(tflow_ps_name_type);
t_p.tflow_ps = TX;
t_p.abstract = TRUE;
pdict_global.add(t_p);

}; -- end of init_path_dict()

-- method used to expand a tflow
expand_flow(bt:base_track,p: tflow_ps) is {
   var temp1: list of base_track_name_type;
   var temp2: list of base_track_name_type;
   var bt_temp: base_track = new base_track;
   var t_p: tflow_ps;

   if bt.track_kind == END or p.flow_bs.size() == 9 then {

      -- we need to handle the loop
      if bt.track_kind == END then { -- to be removed
         t_p = new;
         t_p = deep_copy(p);
         pdict_global.add(p);
      }; -- to be removed
   } else {
      temp1 = bt.get_internal_next();
      for each (t) in temp1 do {
         t_p = new;
         t_p = deep_copy(p);
         t_p.flow_bs.add(t);
         bt_temp = deep_copy(bts_global.key(t));
         expand_flow(bt_temp,t_p);
      };
   };
};

-- method used retrieve an tflow identification name from the tflow dictionary
get_flowid_global(l:list of base_track_name_type):tflow_ps_name_type is {
   result = pdict_global.get_flowid(l);
};

};
'>
```

D.6 Fichier MAC tracking_macros

```
-------------------------------------------------------------------------------
-- Title         : Mac tracking macros
-- Project       : MAC (Module d'Analyse de la Couverture)
-------------------------------------------------------------------------------
-- File          : mac_tracking_macros.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
-------------------------------------------------------------------------------
-- Description   :
-- This module includes definitions used inside by the tric tracking library
--
-- The following elements are included inside this module:
--    * build_tflow_cover macro
--            A macro to generate the coverage group for the transactional flow
--            collector
--    * Global functions related to build_tflow_cover macro
--    * tflow_ps_cover macro
--            A macro to generate coverage group for the transactional flow tracker
--            Global functions related to
--    * Global functions related to build_tflow_cover macro
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- name: MACRO build_tflow_cover
-- description:
-- This macro build the coverage group for a tflow_collector
-------------------------------------------------------------------------------
<'
define <build_tflow_cover'statement> "build_tflow_cover <name>" as computed {
   var temp_str: string;
   var cross_test: mac_cov_switch_type = ABS_GLOBAL;
   var legal_info: gb_cross_illegal_info;
   temp_str = append("extend  tflow_collector{cover cov_collect is also{");

   --************************************
   -- items declaration
   --************************************
   temp_str = append(temp_str,gb_flow_collect_cov_item(f_sequencer,NULL,0));

   --************************************
   -- cross declaration
   --************************************
   if cross_test == ON then {
      temp_str = append(temp_str," cross ");
      temp_str = append(temp_str,gb_flow_collect_cross(f_sequencer,NULL,0));
      temp_str = append(temp_str," using illegal = not(");

      temp_str =
append(temp_str,gb_flow_collect_cross_illegal(f_sequencer,NULL,0,legal_info));
      temp_str = append(temp_str,")");
   };
   temp_str = append(temp_str,";");

   -- end of the coverage group
   temp_str = append(temp_str,"};}");

   if <1>.as_a(string) == "FILE" then {
      var m_file: file;
      m_file = files.open("mac_cov_group.e", "w", "Text File");
      files.write(m_file,clean_file(temp_str));
      files.close(m_file);
   };
   return temp_str;
};
'>

-------------------------------------------------------------------------------
-- name: MACRO tflow_ps_cover
-- description:
-- This macro build the coverage group for a tflow_tracker_ps
-------------------------------------------------------------------------------
<'
define <tflow_ps_cover'statement> "tflow_ps_cover <name> with <num> and <num>" as
computed {
   var temp_str: string;
   var top_path: int = <2>.as_a(int);
   var ins_num: int = <3>.as_a(int);
   var illegal_cause_nb: int;
   var bt_legal_name: list of base_track_name_type;

   temp_str = append("extend ",<1>," tflow_tracker{when ON ",
                     <1>," tflow_tracker{cover completed is also{");
   for i from 0 to (top_path-1) do {
      bt_legal_name.clear();
      bt_legal_name = bts_item_illegal(<1>.as_a(tflow_ps_type),i,top_path,ins_num);
      temp_str = append(temp_str,"item ",<1>,"_t",i,
                        ": base_track_name_type = flow[",i,
                        "].id_name using illegal = not (",<1>,
                        "_t",i," in [");
      for each (b) in bt_legal_name do {
         if index != 0 then {
            temp_str = append(temp_str,",");
         };
         temp_str = append(temp_str,b.as_a(string));
      };
      temp_str = append(temp_str,"]), no_collect ;"); -- no_collect
   };

   -- Cross Definition
   temp_str = append(temp_str," cross ");
   for i from 0 to (top_path-1) do {
      temp_str = append(temp_str,<1>,"_t",i);
      if i != (top_path-1) then{
         temp_str = append(temp_str,", ");
      };
   };
   -- illegal cross definition
   temp_str = append(temp_str," using name = ",<1>,"_path, illegal = not (");

   for i from 0 to (top_path-1) do {
      if i != 0 then {
```

```
            temp_str = append(temp_str," and ");
        };
        temp_str = append(temp_str,bts_cross_illegal(<1>.as_a(tflow_ps_type),
                                                     i,top_path,ins_num));
    };
    -- close the cross item
    temp_str = append(temp_str," );");

    -- close the coverage group of the extension of the struct
    temp_str = append(temp_str,"};};}");
    return temp_str;
};
'>

-------------------------------------------------------------------------------
-- name: global functions
-- description:
-- Global functions related to build_tflow_cover macro
-------------------------------------------------------------------------------
<'
  struct gb_cross_illegal_info{
      !abstract : bool;
      !const_name : list of tflow_ps_name_type;
      !th: tflow_hierarchy;
      !name : string;
      clean() is {
         var temp_cn: list of tflow_ps_name_type;
         for each (c) in const_name do {
            if index == 0 then {
               temp_cn.add(c);
            } else {
               if (temp_cn.all(it == c)).is_empty() then {
                  temp_cn.add(c);
               };
            };
         };
         const_name.clear();
         const_name.add(temp_cn);
      };
};
extend global {

   clean_file(t_s:string): string is {
      var temp_str1: string;
      var temp_str2: string;
      var s_split: list of string;

      -- split the "{" character
      s_split = str_split(t_s, "{");
      for each (s) in s_split do {
         if index != 0 then {
            temp_str1 = append(temp_str1,"{\n");
         };
         temp_str1 = append(temp_str1,s);
      };
      -- split the ";" character
      s_split = str_split(temp_str1, ";");
      for each (s) in s_split do {
         if index != 0 then {
           temp_str2 = append(temp_str2,";\n");
         };
         temp_str2 = append(temp_str2,s);
      };
      -- split the "using illegal =" character
      temp_str1 = "";
      s_split = str_split(temp_str2, "using illegal =");
      for each (s) in s_split do {
         if index != 0 then {
           temp_str1 = append(temp_str1,"using illegal =\n");
         };
         temp_str1 = append(temp_str1,s);
      };
      -- split the "and" character
      temp_str2 = "";
      s_split = str_split(temp_str1, "and");
      for each (s) in s_split do {
         if index != 0 then {
           temp_str2 = append(temp_str2,"and\n");
         };
         temp_str2 = append(temp_str2,s);
      };
      -- split the "or" character
      temp_str1 = "";
      s_split = str_split(temp_str2, "or");
      for each (s) in s_split do {
         if index != 0 then {
           temp_str1 = append(temp_str1,"or\n");
         };
         temp_str1 = append(temp_str1,s);
      };
      return temp_str1;
   };

   comb_nr(n:int,r:int):int is {
      var n_fact: int = 1;
      var r_fact: int = 1;
      var nr_fact: int = 1;
      if n == 0 then {
         return 0;
      } else {
         for i from 1 to n do {
            n_fact *= i;
         };
         if r == 0 then {
            r_fact = 1;
         } else {
            for i from 1 to r do {
               r_fact *= i;
            };
         };
         if (n-r) <= 0 then {
            nr_fact = 1;
         }else {
            for i from 1 to (n-r) do {
               nr_fact *= i;
            };
         };
         return (n_fact)/(r_fact*nr_fact);
      };
   }; -- end comb_nr
```

```
   -- set_comb_vector
   set_comb_vector(n:int,r:int): list of comb_vect is {
      var l: list of comb_vect;
      var b: list of bool;

      b.resize(n,TRUE,FALSE,FALSE);
      recursive_comb(l,b,n,r,0);

      return l;
   }; -- set_comb

   recursive_comb(l: *list of comb_vect, b:list of bool,n:int,r: int, itr:int) is {
      var c_v: comb_vect = new;
      if itr != n then {
         recursive_comb(l,b,n,r,itr+1);
         b[itr] = TRUE;
         if (b.all(it == TRUE)).size() == r then {
            c_v.add(b);
            l.add(c_v);
         };
         recursive_comb(l,b,n,r,itr+1);
         b[itr] = FALSE;
      };
   };

   ------------------------------------------------------------------------------
--
   -- name:         gb_flow_collect_cov_item
   -- author:       sebastien regimbal
   -- description:
   -- this function product the coverage item for the coverage of the transactional
   -- graph
   -- date:         july 16, 2002
   ------------------------------------------------------------------------------
--
   gb_flow_collect_cov_item(tf_hierarchy :tflow_hierarchy,
                            tag: string, level_nb: int): string is{
      -- VARIABLE DECLARATIONS
      -- the string buffer
      var temp_str: string;
      -- list that contain all constraint of a coverage item
      var constraint_name_lst: list of tflow_ps_name_type;

      -- depending of the kind of tflow hierarchical element
      case tf_hierarchy.kind {
         -- sequential element case
         SEQUENTIAL: {
            if tf_hierarchy.switch == ON then {
               for each (sequential_element) in tf_hierarchy.f do {
                  temp_str = append(temp_str,
                                    gb_flow_collect_cov_item(sequential_element,

append(tag,"sq",index,"_"),
                                                             level_nb + 1));
                  -- if a TERMINAL element, we must specify the illegal value
                  if sequential_element.kind == TERMINAL then {
                     -- start item illegal values
                     temp_str = append(temp_str," using illegal = not(
L",level_nb+1,"_",
                                       (sequential_element.switch ==
ON?"":"abstract_"),
                                       tag,"sq",index,"_",
                                       sequential_element.name.as_a(string)," in
[");
                     -- depending if the element is abstracted
                     if sequential_element.switch == ON then {
                        constraint_name_lst =
pdict_global.get_flowid_name(sequential_element.name);
                        for each (constraint_name) in constraint_name_lst do {
                           if index != 0 then {
                              temp_str = append(temp_str,",");
                           };
                           temp_str = append(temp_str,constraint_name.as_a(string),
                                             "_",sequential_element.id);
                        };
                     } else {
                        temp_str = append(temp_str,"ABSTRACT_COVER");
                     };
                     if sequential_element.mandatory_element == FALSE then {
                        temp_str = append(temp_str,",EMPTY");
                     };
                     temp_str = append(temp_str,"]), no_collect;");
                  }; -- end of TERMINAL case
               };
            } else {-- we have an abstract coverage
               -- creation of an item
               temp_str = append(temp_str," item L",level_nb,"_abstract_",tag,
                                 "sq0 : tflow_ps_name_type =
gb_item_value_abstract({",
                                 gb_find_item_value_abstract(tf_hierarchy),"})");
               -- start item illegal values
               temp_str = append(temp_str," using illegal = not(
L",level_nb,"_abstract_",
                                 tag,"sq0 in [ABSTRACT_COVER");
               if tf_hierarchy.mandatory_element == FALSE then {
                  temp_str = append(temp_str,",EMPTY");
               };
               temp_str = append(temp_str,"]), no_collect;");
            };


         }; -- end of sequential element case
         -- concurrent element case
         CONCURRENT: {
            case {
               -- case different max parallel instance and number of instance
               tf_hierarchy.max_parallel != tf_hierarchy.nb_element : {
                  if tf_hierarchy.switch == ON then {
                     -- sub element creation
                     -- for all possible parallel instance
                     for max_itr from 0 to tf_hierarchy.max_parallel -1 do {
                        -- creation of the sub element
                        temp_str = append(temp_str,

gb_flow_collect_cov_item(tf_hierarchy.f[max_itr],

append(tag,"cc",max_itr,"_"),
                                                                level_nb + 1));
                        -- if a TERMINAL element, we must specify the illegal value
```

```
                        if tf_hierarchy.f[max_itr].kind == TERMINAL then {
                           -- start item illegal values
                           temp_str = append(temp_str," using illegal = not(
L",level_nb+1,"_",
                                             (tf_hierarchy.f[max_itr].switch ==
ON?"":"abstract_"),
                                             tag,"cc",max_itr,"_",
                                             tf_hierarchy.name.as_a(string)," in
[");
                           -- depending if the element is abstracted
                           if tf_hierarchy.f[max_itr].switch == ON then {
                              constraint_name_lst =
pdict_global.get_flowid_name(tf_hierarchy.name);
                              for each (constraint_name) in constraint_name_lst do
{
                                 if index != 0 then {
                                    temp_str = append(temp_str,",");
                                 };
                                 for inst_itr from 0 to tf_hierarchy.nb_element - 1
do {
                                    if inst_itr != 0 then {
                                       temp_str = append(temp_str,",");
                                    };
                                    temp_str =
append(temp_str,constraint_name.as_a(string),
                                                   "_",inst_itr);
                                 };
                              };
                           } else {
                              temp_str = append(temp_str,"ABSTRACT_COVER");
                           };
                           -- HERE the mandatory value of the CONCURRENT ELEMENT
                           if tf_hierarchy.mandatory_element == FALSE then {
                              temp_str = append(temp_str,",EMPTY");
                           };
                           temp_str = append(temp_str,"]), no_collect;");
                        }; -- end of illegal value for TERMINAL element
                     }; -- end of for all possible parallel instance

                  } else { -- we have an abstract coverage
                     -- creation of an item
                     temp_str = append(temp_str," item
L",level_nb,"_abstract_",tag,
                                       "cc0 : tflow_ps_name_type =
gb_item_value_abstract({",
gb_find_item_value_abstract(tf_hierarchy),"})");
                     -- start item illegal values
                     temp_str = append(temp_str," using illegal = not(
L",level_nb,"_abstract_",
                                       tag,"cc0 in {ABSTRACT_COVER");
                     if tf_hierarchy.mandatory_element == FALSE then {
                        temp_str = append(temp_str,",EMPTY");
                     };
                     temp_str = append(temp_str,"]), no_collect;");
                  }; -- end of abstract or none asbtract
               }; -- end of case different max parallel instance and number of
instance
               -- case same max parallel instance and number of instance
               tf_hierarchy.max_parallel == tf_hierarchy.nb_element : {
                  -- if the element is specified asbtracted
                  if tf_hierarchy.switch == ON then {
                     -- creation of the sub elements
                     for each (concurrent_element) in tf_hierarchy.f do {
                        -- creation of a sub element
                        temp_str = append(temp_str,
gb_flow_collect_cov_item(concurrent_element,
append(tag,"cc",index,"_"),
                                                                  level_nb + 1));
                        -- if we have a TERMINAL element, we must specify the
illegal value
                        if concurrent_element.kind == TERMINAL then {
                           -- start item illegal values
                           temp_str = append(temp_str," using illegal = not(
L",level_nb+1,"_",
                                             (concurrent_element.switch ==
ON?"":"abstract_"),
                                             tag,"cc",index,"_",
                                             concurrent_element.name.as_a(string),"
in [");
                           -- depending if the element is abstracted
                           if concurrent_element.switch == ON then {
                              constraint_name_lst =
pdict_global.get_flowid_name(concurrent_element.name);
                              for each (constraint_name) in constraint_name_lst do
{
                                 if index != 0 then {
                                    temp_str = append(temp_str,",");
                                 };
                                 temp_str =
append(temp_str,constraint_name.as_a(string),
                                                "_",concurrent_element.id);
                              };
                           } else {
                              temp_str = append(temp_str,"ABSTRACT_COVER");
                           };
                           if concurrent_element.mandatory_element == FALSE then {
                              temp_str = append(temp_str,",EMPTY");
                           };
                           temp_str = append(temp_str,"]), no_collect;");
                        }; -- end of TERMINAL case
                     }; -- end of creation of the sub elements
                  } else { -- we have an abstract coverage
                     temp_str = append(temp_str," item
L",level_nb,"_abstract_",tag,
                                       "cc0 : tflow_ps_name_type =
gb_item_value_abstract({",
gb_find_item_value_abstract(tf_hierarchy),"})");
                     -- start item illegal values
                     temp_str = append(temp_str," using illegal = not(
L",level_nb,"_abstract_",
                                       tag,"cc0 in {ABSTRACT_COVER");
                     if tf_hierarchy.mandatory_element == FALSE then {
                        temp_str = append(temp_str,",EMPTY");
                     };
                     temp_str = append(temp_str,"]), no_collect;");
                  };
```

```
            };--  end of case same max parallel instance and number of instance
         }; -- end of internal concurrent case
      }; -- end of concurrent element case
      -- terminal element case
      TERMINAL: {
         -- creation of an item
         temp_str = append(temp_str," item L",level_nb,"_",(tf_hierarchy.switch
== ON?"":"abstract_"),
                           tag,tf_hierarchy.name.as_a(string),
                           " :tflow_ps_name_type = gb_item_value(trans_flow.",
                           tf_hierarchy.name.as_a(string),
                           "_flow[",tf_hierarchy.id,"],",
                           tf_hierarchy.switch.as_a(string),")");

      }; -- end of terminal element case
   };
   return temp_str;
};

-------------------------------------------------------------------------------
--
   -- name:         utilities function for coverage group declaration
   -- author:       sebastien regimbal
   -- description:
   -- gb_item_value
   -- gb_item_value_abstract
   -- gb_find_item_value_abstract
   -- date:         july 16, 2002
   -------------------------------------------------------------------------------
--
   gb_item_value(f:tflow_ps_name_type,s:mac_cov_switch_type): tflow_ps_name_type is
{
      if f == EMPTY then {
         return EMPTY;
      } else if s == OFF then {
         return ABSTRACT_COVER;
      } else {
         return f;
      };
   };
   gb_item_value_abstract(f: list of tflow_ps_name_type): tflow_ps_name_type is {
      if !(f.all(it != EMPTY)).is_empty() then {
         return ABSTRACT_COVER;
      } else {
         return EMPTY;
      };
   };
   gb_find_item_value_abstract(ts :tflow_hierarchy) : string is {
      var temp_str: string;
      if ts.kind == CONCURRENT and ts.max_parallel != ts.nb_element then {
         for max_itr from 0 to ts.max_parallel - 1 do {
            temp_str = append(temp_str,gb_find_item_value_abstract(ts.f[max_itr]));
         };
      } else if ts.kind == TERMINAL then {
         temp_str = append(temp_str,"trans_flow.",ts.name.as_a(string),
                           "_flow[",ts.id,"]");
      } else {
         for each (t) in ts.f do {
            if index != 0  then {
               temp_str = append(temp_str,";");
            };
            if t.kind != TERMINAL then {
               temp_str = append(temp_str,gb_find_item_value_abstract(t));
            } else {
               temp_str = append(temp_str,"trans_flow.",t.name.as_a(string),
                                 "_flow[",t.id,"]");
            };
         };
      };
      return temp_str;

   };

   -------------------------------------------------------------------------------
--
   -- name:         gb_flow_collect_cross
   -- author:       sebastien regimbal
   -- description:
   -- this function product the cross item for the tflow coverage  group
   -- date:         july 16, 2002
   -------------------------------------------------------------------------------
--
   gb_flow_collect_cross(tf_hierarchy :tflow_hierarchy, tag: string, level_nb:
int): string is{
      -- VARIABLE DECLARATIONS
      -- the string buffer
      var temp_str: string;
      -- list that contain all constraint of a coverage item
      var constraint_name_lst: list of tflow_ps_name_type;

      -- depending of the kind of tflow hierarchical element
      case tf_hierarchy.kind {
         -- sequential element case
         SEQUENTIAL: {
            -- put a comma to separe cross element
            if tf_hierarchy.switch == ON then {
               for each (sequential_element) in tf_hierarchy.f do {
                  if index != 0 then {
                     temp_str = append(temp_str,",");
                  };
                  temp_str = append(temp_str,
                                    gb_flow_collect_cross(sequential_element,
append(tag,"sq",index,"_"),
                                                          level_nb + 1));
               };
            } else {-- we have an abstract coverage
               -- creation of a cross element
               temp_str = append(temp_str,"L",level_nb,"_abstract_",tag,"sq0");;
            };

         }; -- end of sequential element case
         -- concurrent element case
         CONCURRENT: {
            case {
               -- case different max parallel instance and number of instance
               tf_hierarchy.max_parallel != tf_hierarchy.nb_element : {
                  if tf_hierarchy.switch == ON then {
                     -- sub element creation
                     -- for all possible parallel instance
```

```
                    for max_itr from 0 to tf_hierarchy.max_parallel -1 do {
                       -- put a comma to separe cross element
                       if max_itr != 0  then {
                          temp_str = append(temp_str,",");
                       };
                       -- creation of the sub element
                       temp_str = append(temp_str,

gb_flow_collect_cross(tf_hierarchy.f[max_itr],

append(tag,"cc",max_itr,"_"),
                                                                level_nb + 1));
                    }; -- end of for all possible parallel instance
                 } else { -- we have an abstract coverage
                    -- creation of an item
                    temp_str =
append(temp_str,"L",level_nb,"_abstract_",tag,"cc0");
                 }; -- end of abstract or none asbtract
              }; -- end of case different max parallel instance and number of
instance
              -- case same max parallel instance and number of instance
              tf_hierarchy.max_parallel == tf_hierarchy.nb_element : {
                 -- if the element is specified asbtracted
                 if tf_hierarchy.switch == ON then {
                    -- creation of the sub elements
                    for each (concurrent_element) in tf_hierarchy.f do {
                       if index != 0  then {
                          temp_str = append(temp_str,",");
                       };
                       -- creation of a sub element
                       temp_str = append(temp_str,
                                    gb_flow_collect_cross(concurrent_element,

append(tag,"cc",index,"_"),
                                                                level_nb + 1));
                    }; -- end of creation of the sub elements
                 } else { -- we have an abstract coverage
                    temp_str =
append(temp_str,"L",level_nb,"_abstract_",tag,"cc0");
                 };
              };--  end of case same max parallel instance and number of instance
           }; -- end of internal concurrent case
        }; -- end of concurrent element case
        -- terminal element case
        TERMINAL: {
           -- creation of an item
           temp_str = append(temp_str,"L",level_nb,"_",(tf_hierarchy.switch ==
ON?"":"abstract_"),
                             tag,tf_hierarchy.name.as_a(string));

        }; -- end of terminal element case
     };
     return temp_str;
  };

  ------------------------------------------------------------------------------
--
  -- name:         gb_flow_collect_cross
  -- author:       sebastien regimbal
  -- description:
  -- this function product the cross item for the tflow coverage  group
  -- date:         july 16, 2002
  ------------------------------------------------------------------------------
--
  gb_flow_collect_cross_illegal(tf_hierarchy :tflow_hierarchy, tag:
string,level_nb:int, prev_info: *gb_cross_illegal_info ): string is{
     -- variable declarations
     var temp_str             : string;

     var cur_const_name           : list of tflow_ps_name_type;
     var abs_name             : list of tflow_ps_name_type;

  --    var const_name_cur  : list of tflow_ps_name_type;
    --  var prev_ts             : tflow_hierarchy;
    --  var const_name      : list of tflow_ps_name_type;
    --  var prev_abstract       : bool = FALSE;

     var next_prev_info       : gb_cross_illegal_info;

     var nb_combination       : int;
     var comb_vect_temp       : list of comb_vect;

     -- depending of the kind of tflow hierarchical element
     case tf_hierarchy.kind {
        -- sequential element case
        SEQUENTIAL: {

           if tf_hierarchy.switch == ON then {
              -- for each sewuential of the current hierchical element
              for each (sequential_element) using index (seq_itr) in
tf_hierarchy.f do {
                 next_prev_info = new;
                 -- put a AND if we are not at the first hierarchical element
                 if seq_itr != 0 then {
                    temp_str = append(temp_str,"and");
                 };
                 -- put a parenthesis to start a hierarchical element
                 temp_str = append(temp_str,"(");
                 -- if this element is the first of the top hierarchical sequence
                 if sequential_element.kind == TERMINAL then {
                    if prev_info == NULL then {
                       error("Cross illegal value: First hierarchical element \
                             (Sequential) is not yet implemented");
                    } else {
                       next_prev_info.abstract = (sequential_element.switch ==
ON?FALSE:TRUE);
                       next_prev_info.th = sequential_element;
                       next_prev_info.name = append("L",level_nb+1,"_",
                                                    (sequential_element.switch ==
ON?"":"abstract_"),
                                                    tag,"sq",seq_itr,"_",
                                                    sequential_element.name);

                       -- for all the previous constraint
                       for each (prev_constraint_name) in prev_info.const_name do
{
                          -- put a AND if we are not at the first hierarchical
element
                          if index != 0 then {
                             temp_str = append(temp_str,"and");
```

```
                        };
                        -- put a parenthesis to start a hierarchical element
                        temp_str = append(temp_str,"(");
                        -- for all the previous instance
                        var nb_inst := prev_info.th.kind == CONCURRENT ?
(prev_info.th.nb_element - 1) : 0;
                        for prev_inst_itr from 0 to nb_inst do {
                           -- put a AND if we are at the first constraint of the
previous element
                           if prev_inst_itr != 0 then {
                           temp_str = append(temp_str,"and");
                           };
                           -- put a parenthesis to start a hierarchical element
                           temp_str = append(temp_str,"(");



                           -- construct the legal value constraint
                           temp_str = append(temp_str,prev_info.name," ==
",prev_constraint_name);
                           if prev_constraint_name not in [ABSTRACT_COVER,EMPTY]
then {
                              temp_str = append(temp_str,"_",prev_inst_itr);
                           };
                           temp_str = append(temp_str," => L",level_nb+1,"_",
                                             (sequential_element.switch ==
ON?"":"abstract_"),
                                             tag,"sq",seq_itr,"_",
                                             sequential_element.name," in [");


                           -- when the previous element is abtracted and it is
not EMPTY
                           if prev_constraint_name == ABSTRACT_COVER then {
                              if prev_constraint_name == EMPTY then {
                                 cur_const_name.add(EMPTY);
                              }else {
                                 -- all possibility are taken for the current
element
                                 cur_const_name =
pdict_global.get_flowid_name(sequential_element.name);
                                 if sequential_element.mandatory_element ==
FALSE then {
                                    cur_const_name.add0(EMPTY);
                                 };
                              };
                           } else {
                              -- otherwise we retrieve good constraint according
to the
                              -- previous contraint
                              cur_const_name =
pdict_global.get_next_flow(prev_constraint_name);
                           };

                           if sequential_element.switch == ON then {
                              for each (cur_const) using index(cur_const_itr) in
cur_const_name do {
                                 if cur_const_itr != 0 then {
                                    temp_str = append(temp_str,",");
                                 };
                                 temp_str = append(temp_str,cur_const);
                                 if cur_const != EMPTY then {
                                    temp_str =
append(temp_str,"_",sequential_element.id);
                                 };
                              };
                              next_prev_info.const_name.add(cur_const_name);
                           } else {
                              if not cur_const_name.all(it != EMPTY).is_empty()
then {
                                 temp_str = append(temp_str,"ABSTRACT_COVER");
                                 next_prev_info.const_name.add(ABSTRACT_COVER);
                                 if not cur_const_name.all(it ==
EMPTY).is_empty() then {
                                    temp_str = append(temp_str,",");
                                 };
                              };
                              if not cur_const_name.all(it == EMPTY).is_empty()
then {
                                 temp_str = append(temp_str,"EMPTY");
                                 next_prev_info.const_name.add(EMPTY);
                              };

                           };

--                           if next_prev_info.abstract
                           -- put a parenthesis for the end of the cross element
constraint
                           temp_str = append(temp_str,"]|");
                           if prev_constraint_name in [ABSTRACT_COVER,EMPTY]
then {
                              break;
                           };
                        }; -- end of for all the previous instance
                        -- put a parenthesis for the end of the cross element
constraint
                        temp_str = append(temp_str,")");
                     };  -- end of    for all the previous constraint
                  }; -- end of prev_info is not NULL

               } else {
                  temp_str = append(temp_str,

gb_flow_collect_cross_illegal(sequential_element,

append(tag,"sq",seq_itr,"_"),
                                                                  level_nb +
1,prev_info));
                  next_prev_info = deep_copy(prev_info);
               };

               -- put a parenthesis for the end of the hierarchical element
               temp_str = append(temp_str,")");
               next_prev_info.clean();
               prev_info = deep_copy(next_prev_info);

            };
         } else {-- we have an abstract coverage
            next_prev_info = new;
            next_prev_info.abstract = TRUE;
            next_prev_info.th = tf_hierarchy;
```

```
            next_prev_info.name = append("L",level_nb,"_abstract_",tag,"sq0");
            -- creation of an item
            if prev_info == NULL then {
               next_prev_info.const_name.add(ABSTRACT_COVER);
               temp_str = append(temp_str,"L",level_nb,"_abstract_",tag,
                                 "sq0 in [ABSTRACT_COVER");
               if tf_hierarchy.mandatory_element == FALSE then {
                  temp_str = append(temp_str,",EMPTY");
                  next_prev_info.const_name.add(EMPTY);
               };
               temp_str = append(temp_str,"]");
            } else{
               for each (prev_const_name) in prev_info.const_name do {
                  -- put a AND if we are not at the first hierarchical element
                  if index != 0 then {
                     temp_str = append(temp_str,"and");
                  };
                  -- put a parenthesis to start a hierarchical element
                  temp_str = append(temp_str,"(");


                  -- for all the previous instance
                  for prev_inst_itr from 0 to (prev_info.th.nb_element - 1) do {
                     -- put a AND if we are at the first constraint of the
previous element
                     if prev_inst_itr != 0 then {
                        temp_str = append(temp_str,"and");
                     };
                     -- put a parenthesis to start a hierarchical element
                     temp_str = append(temp_str,"(");
                     -- construct the legal value constraint
                     temp_str = append(temp_str,prev_info.name," ==
",prev_const_name);
                     if prev_const_name not in [ABSTRACT_COVER,EMPTY]  then {
                        temp_str = append(temp_str,"_",prev_inst_itr);
                     };
                     temp_str = append(temp_str," =>
L",level_nb,"_abstract_",tag,
                                       "sq0 in [");


                     if prev_const_name == EMPTY then {
                        temp_str = append(temp_str,"EMPTY");
                        next_prev_info.const_name.add(EMPTY);
                     } else {
                        if prev_info.abstract then {
                           temp_str = append(temp_str,"ABSTRACT_COVER");
                           next_prev_info.const_name.add(ABSTRACT_COVER);
                           if tf_hierarchy.mandatory_element
                           == FALSE and prev_info.th.get_deep_mandatory()
                           == FALSE then {
                              temp_str = append(temp_str,",EMPTY");
                              next_prev_info.const_name.add(EMPTY);
                           };
                        } else {
                           cur_const_name =
pdict_global.get_next_flow(prev_const_name);
                           if not cur_const_name.all(it != EMPTY).is_empty()
then {
                              temp_str = append(temp_str,"ABSTRACT_COVER");
                              next_prev_info.const_name.add(ABSTRACT_COVER);
                           };
                           if not cur_const_name.all(it == EMPTY).is_empty()
then {
                              temp_str = append(temp_str,"EMPTY");
                              next_prev_info.const_name.add(EMPTY);
                           };
                        };
                     };


                     temp_str = append(temp_str,"])");
                     if prev_const_name in [ABSTRACT_COVER,EMPTY] then {
                        break;
                     };
                  }; -- end of for all the previous instance


                  -- put a parenthesis for the end of the hierarchical element
                  temp_str = append(temp_str,")");
               };
               next_prev_info.clean();
               prev_info = deep_copy(next_prev_info);
            };
         }; -- end of sequential element case
         -- concurrent element case
         CONCURRENT: {
            case {
               -- case different max parallel instance and number of instance
               tf_hierarchy.max_parallel != tf_hierarchy.nb_element : {
                  next_prev_info = new;
                  if tf_hierarchy.switch == ON then {
                     next_prev_info.abstract = FALSE;
                     next_prev_info.th = tf_hierarchy;


                     -- if this element is the first of the top hierarchical
sequence
                     if prev_info == NULL then {
                        -- sub element creation
                        -- for all possible parallel instance
                        for max_itr from 0 to tf_hierarchy.max_parallel -1 do {
                           -- put a AND if we are not at the first hierarchical
element
                           if max_itr != 0 then {
                              temp_str = append(temp_str,"and");
                           };
                           -- put a parenthesis to start a hierarchical element
                           temp_str = append(temp_str,"(");
                           -- if the hierarchical element is TERMINAL
                           if tf_hierarchy.f[max_itr].kind == TERMINAL then {
                              temp_str = append(temp_str,"L",level_nb+1,"_",
                                                (tf_hierarchy.f[max_itr].switch ==
ON?"":"abstract_"),
                                                tag,"cc",max_itr,
                                                "_",tf_hierarchy.f[max_itr].name,"
in [");
                              next_prev_info.name
                              = append("L",level_nb+1,"_",
```

```
                                    (tf_hierarchy.f[max_itr].switch ==
ON?"":"abstract_"),
                                    tag,"cc",max_itr,
                                    "_",tf_hierarchy.f[max_itr].name);
                              -- find the constraints of the current element
                              -- the function call depend if the element is
abtracted
                              -- depending if the element is abstracted
                              if tf_hierarchy.f[max_itr].switch == ON then {
                                 cur_const_name  =
pdict_global.get_flowid_name(tf_hierarchy.f[max_itr].name);

                                 -- for all the constraints of the current element
                                 for each (constraint_name) using index
(constraint_itr) in cur_const_name do {
                                    -- put a comma is we are not at the first
contraint
                                    if constraint_itr != 0 then {
                                       temp_str = append(temp_str,",");
                                    };

                                    -- for all the instances of the current element
                                    for instance_itr from 0 to
(tf_hierarchy.nb_element -1) do {
                                       -- put a comma for several instances
                                       if instance_itr != 0 then {
                                          temp_str = append(temp_str,",");
                                       };
                                       -- construct the constraint we the instance
number
                                       temp_str =
append(temp_str,constraint_name.as_a(string),
                                                          "_",instance_itr);
                                    };
                                    next_prev_info.const_name.add(constraint_name);
                                 };  -- end of the const_itr loop


                              } else {
                                 temp_str = append(temp_str,"ABSTRACT_COVER");
                                 next_prev_info.const_name.add(ABSTRACT_COVER);
                              };
                              -- HERE the mandatory value of the CONCURRENT ELEMENT
                              if tf_hierarchy.mandatory_element == FALSE then {
                                 temp_str = append(temp_str,",EMPTY");
                                 next_prev_info.const_name.add(EMPTY);
                              };
                              temp_str = append(temp_str,"]");

                           } else {
                              -- the element  is CONCURRENT or SEQUENTIAL
                              temp_str = append(temp_str,

gb_flow_collect_cross_illegal(tf_hierarchy.f[max_itr],

append(tag,"cc",max_itr,"_",

level_nb + 1,prev_info));
                           };
```

```
                              -- put a parenthesis for the end of the hierarchical
element
                              temp_str = append(temp_str,")");
                           }; -- end of for all possible parallel instance
                        } else{
                           out("allo");
                        };
                        next_prev_info.clean();
                        prev_info = deep_copy(next_prev_info);
                     } else { -- we have an abstract coverage
                        next_prev_info.abstract = TRUE;
                        next_prev_info.th = tf_hierarchy;
                        next_prev_info.name =
append("L",level_nb,"_abstract_",tag,"cc0");
                        if prev_info == NULL then {
                           next_prev_info.const_name.add(ABSTRACT_COVER);
                           temp_str = append(temp_str,"L",level_nb,"_abstract_",tag,
                                             "cc0 in [ABSTRACT_COVER");
                           if tf_hierarchy.mandatory_element == FALSE then {
                              temp_str = append(temp_str,",EMPTY");
                              next_prev_info.const_name.add(EMPTY);
                           };
                           temp_str = append(temp_str,"]");
                        } else{
                           out("allo");
                        };
                        next_prev_info.clean();
                        prev_info = deep_copy(next_prev_info);
                     }; -- end of abstract or none asbtract
                  }; -- end of case different max parallel instance and number of
instance
                  -- case same max parallel instance and number of instance
                  tf_hierarchy.max_parallel == tf_hierarchy.nb_element : {
                     next_prev_info = new;
                     -- if the element is specified asbtracted
                     if tf_hierarchy.switch == ON then {
                        next_prev_info.abstract = FALSE;
                        next_prev_info.th = tf_hierarchy;
                        next_prev_info.name = append("L",level_nb+1,"_",
                                                     tag,"cc0");
                        -- if this element is the first of the top hierarchical
sequence
                        if prev_info == NULL then {
                           out("not yet implemented");
                        } else{
                           -- for all instance of the previous element
                           var nb_inst := prev_info.th.kind == CONCURRENT ?
(prev_info.th.nb_element - 1) : 0;
                           for prev_inst_itr from 0 to nb_inst do {
                              -- put a AND if we are not at the first hierarchical
element
                              if prev_inst_itr != 0 then {
                                 temp_str = append(temp_str,"and");
                              };
                              -- put a parenthesis to start a hierarchical element
                              temp_str = append(temp_str,"("); -- start of
prev_inst_itr

                              -- for all the previous constraint
```

```
                        for each (prev_constraint_name) using index
(prev_const_itr) in prev_info.const_name do {
                           -- put a AND if we are not at the first hierarchical
element
                           if prev_const_itr != 0 then {
                              temp_str = append(temp_str,"and");
                           };
                           -- put a parenthesis to start a hierarchical element
                           temp_str = append(temp_str,"("); -- start of
prev_const_itr

                           -- sub element creation

                           -- construct the legal value constraint
                           temp_str = append(temp_str,prev_info.name," ==
",prev_constraint_name);
                           if prev_constraint_name not in [ABSTRACT_COVER,EMPTY]
then {
                              temp_str = append(temp_str,"_",prev_inst_itr);
                           };
                           temp_str = append(temp_str," => (");


                        -- if the previous element is abtracted
                        if prev_info.abstract then {
                           -- we dont need combinatory computation
                           nb_combination = 1;
                           comb_vect_temp =
set_comb_vector(tf_hierarchy.max_parallel,

tf_hierarchy.max_parallel);
                        } else {
                           -- we compute the number of constraint combinations
                           nb_combination =
comb_nr(tf_hierarchy.max_parallel,prev_const_itr);
                           -- the combinatory vector is computed
                           comb_vect_temp =
set_comb_vector(tf_hierarchy.max_parallel,prev_const_itr);
                        };
                        --- for all the possible combination
                        for combination_itr from 0 to nb_combination-1 do {
                           -- put a OR the multiple combination
                           if combination_itr != 0 then {
                              temp_str = append(temp_str," or ");
                           };
                           temp_str = append(temp_str,"(");

                           -- for all instance of the current element
                           for cur_instance_itr from 0 to
(tf_hierarchy.nb_element -1) do {
                              -- put a AND for multiple constraint
                              if cur_instance_itr != 0 then {
                                 temp_str = append(temp_str," and ");
                              };
                              -- construct the name of the current elememt
                              temp_str = append(temp_str,"(L",level_nb+1,"_",

(tf_hierarchy.f[cur_instance_itr].switch
                                                    == ON?"":"abstract_"),
                                                   tag,"cc",cur_instance_itr,"_",
tf_hierarchy.f[cur_instance_itr].name);
                                    -- start of the constraint for the current element
                                    temp_str = append(temp_str," in [");

                                    -- when the previous element is abtracted and it
is not EMPTY
                                    if prev_info.abstract and prev_constraint_name !=
EMPTY then {
                                       -- all possibility are taken for the current
element
                                       cur_const_name =

pdict_global.get_flowid_name(tf_hierarchy.f[cur_instance_itr].name);
                                       -- if the EMPTY is allowed, we add it
                                       if
tf_hierarchy.f[cur_instance_itr].mandatory_element == FALSE then {
                                          cur_const_name.add(EMPTY);
                                       };
                                    } else {
                                       -- otherwise we retrieve good constraint
according to the
                                       -- previous contraint
                                       cur_const_name =
pdict_global.get_next_flow(prev_constraint_name);
                                    };


                                    -- for all current constraint
                                    for each (cur_const_name) using index(cur_c_itr)
in cur_const_name do {
                                       -- put a comma to separe constraints
                                       if cur_c_itr != 0 then {
                                          temp_str = append(temp_str,",");
                                       };

                                       -- if multiple parallel instance is allowed and
                                       -- the combinatory vector is not empty
                                       if !comb_vect_temp.is_empty() then {
                                          -- if the combinatory authorizes the
constraint writing
                                          if
comb_vect_temp[combination_itr].l[cur_instance_itr] == TRUE then {
                                             if
tf_hierarchy.f[cur_instance_itr].switch == ON then {
                                                temp_str =
append(temp_str,cur_const_name.as_a(string));
                                                if cur_const_name != EMPTY then {
                                                   temp_str =
append(temp_str,"_",cur_instance_itr);
                                                };
                                                -- add constraint for the next element

next_prev_info.const_name.add(cur_const_name);
                                             } else { -- the current element is
abstracted
                                                temp_str =
append(temp_str,"ABSTRACT_COVER");
                                                -- add constraint for the next element
```

```
next_prev_info.const_name.add(ABSTRACT_COVER);
                                        };
                                     } else {
                                        -- otherwise we put EMPTY
                                        temp_str = append(temp_str," EMPTY ");
                                     };
                                  } else {

                                     temp_str =
append(temp_str,cur_const_name.as_a(string));
                                        if cur_const_name != EMPTY then {
                                           temp_str =
append(temp_str,"_",cur_instance_itr);
                                        };
                                     -- add constraint for the next element

next_prev_info.const_name.add(cur_const_name);
                                  };
                               };-- end of the constraints for the current
element


                                  temp_str = append(temp_str,"])");
                               }; -- for all instance of the current element
                               temp_str = append(temp_str,")");
                            }; -- end of for all the possible combination

                               -- put a parenthesis to start a hierarchical element
                               temp_str = append(temp_str,"))");
                            }; -- end of for all the previous constraint
                            temp_str = append(temp_str,")");
                         }; -- end of for instance of the previous element
                         next_prev_info.clean();
                         prev_info = deep_copy(next_prev_info);
                      }; -- end the none NULL case
                   } else { -- we have an abstract coverage
                      next_prev_info.abstract = TRUE;
                      next_prev_info.th = tf_hierarchy;
                      next_prev_info.name =
append("L",level_nb,"_abstract_",tag,"cc0");
                      if prev_info == NULL then {
                         next_prev_info.const_name.add(ABSTRACT_COVER);
                         temp_str = append(temp_str,"L",level_nb,"_abstract_",tag,
                                           "cc0 in [ABSTRACT_COVER");
                         if tf_hierarchy.mandatory_element == FALSE then {
                            temp_str = append(temp_str,",EMPTY");
                            next_prev_info.const_name.add(EMPTY);
                         };
                         temp_str = append(temp_str,"]");
                      } else {
                         for each (constraint_name) in prev_info.const_name do {

                            -- put a AND if we are not at the first hierarchical
element
                            if index != 0 then {
                               temp_str = append(temp_str,"and");
                            };
                            -- put a parenthesis to start a hierarchical element
                            temp_str = append(temp_str,"(");
```

```
                            -- for all the previous instance
                            var nb_inst := prev_info.th.kind == CONCURRENT ?
(prev_info.th.nb_element - 1) : 0;
                            for prev_inst_itr from 0 to nb_inst do {
                               -- put a AND if we are at the first constraint of the
previous element
                               if prev_inst_itr != 0 then {
                                  temp_str = append(temp_str,"and");
                               };
                               -- put a parenthesis to start a hierarchical element
                               temp_str = append(temp_str,"(");


                               temp_str = append(temp_str,prev_info.name," ==
",constraint_name);
                               if constraint_name not in [ABSTRACT_COVER,EMPTY]
then {
                                  temp_str = append(temp_str,"_",prev_inst_itr);
                               };
                               temp_str = append(temp_str," =>
L",level_nb,"_abstract_",tag,
                                                 "cc0 in [");
                               if constraint_name == EMPTY then {
                                  temp_str = append(temp_str,"EMPTY");
                                  next_prev_info.const_name.add(EMPTY);
                               } else {
                                  temp_str = append(temp_str,"ABSTRACT_COVER");
                                  next_prev_info.const_name.add(ABSTRACT_COVER);
                                  if prev_info.th.kind != TERMINAL then {
                                     if tf_hierarchy.mandatory_element == FALSE and
prev_info.th.get_deep_mandatory() == FALSE then {
                                        temp_str = append(temp_str,",EMPTY");
                                        next_prev_info.const_name.add(EMPTY);
                                     };
                                  };
                               };
                               temp_str = append(temp_str,"])");
                               if constraint_name in [ABSTRACT_COVER,EMPTY] then {
                                  break;
                               };
                            }; -- end of for all the previous instance
                            -- put a parenthesis for the end of the cross element
constraint
                            temp_str = append(temp_str,")");
                         };  -- end of    for all the previous constraint
                      };
                      next_prev_info.clean();
                      prev_info = deep_copy(next_prev_info);
                   };

                };--  end of case same max parallel instance and number of instance
             }; -- end of internal concurrent case
          }; -- end of concurrent element case
          -- terminal element case
          TERMINAL: {
             error("Cross illegal value: TERMINAL case is not used!");
          }; -- end of terminal element case
       };
```

```
      return temp_str;

   }; -- end of gb_flow_collect_cross_illegal
};
'>
-------------------------------------------------------------------------------
-- name: global functions
-- description:
-- Global functions related to tflow_ps_cover macro
-------------------------------------------------------------------------------
<'
extend global {
 bts_item_illegal(t:tflow_ps_type ,num: int, top:int,
                  ins_num:int):list of base_track_name_type is {
      var bt_temp1: list (key:id_name) of base_track;
      var bt_temp2: list (key:id_name) of base_track;
      var bt_name_ret: list of base_track_name_type;
      var bt_name_temp: list of base_track_name_type;
      var empty_enable :bool = FALSE;
      var cur_base_track: base_track;
      bt_temp1 = bts_global.all(it.tflow_ps_kind == t and it.track_kind == START);
      if num != 0 then {
         for i from 1 to num do {
            bt_temp2.clear();
            bt_name_temp.clear();
            -- if a END track is reached, all other tracks could be EMPTY
            if !(bt_temp1.all(it.track_kind == END)).is_empty() then{
               empty_enable = TRUE;
            };
            bt_temp2 = bt_temp1.all(it.track_kind != END);
            -- for all current base track, we find the next TRACK
            for each (b) in bt_temp2 do {
               bt_name_temp.add(b.get_internal_next());
            };
            bt_temp1.clear();
            for each (bt) in bt_name_temp do {
               bt_temp1.add(bts_global.key(bt));
            };
            --bt_temp1.clear();
            --bt_temp1 = bt_temp2;
            --bt_name_temp = bt_name_temp.unique(it);
         };
      };
      -- Refinement
      --
      for each (b) in bt_temp1 do {
         for j from 0 to (ins_num-1) do {
            cur_base_track = new base_track;
            cur_base_track = deep_copy(b);
            cur_base_track.refine_track(j);
            bt_name_ret.add(cur_base_track.id_name);
         };
      };
      if empty_enable == TRUE then {
         bt_name_ret.add(EMPTY);
      };

      return bt_name_ret;
   }; // END bts_item_illegal
```

```
   -------------------------------------------------------------------------------
--
   -- name:         base_track
   -- author:       sebastien regimbal
   -- description:
   -- This struct characterize a simple element of a path. A path is built with
   -- several base tracks
   -- date:         may 28, 2002
   -------------------------------------------------------------------------------
--
   bts_cross_illegal(t:tflow_ps_type,ind:int,top:int, ins_num:int):string is {
      var temp_str1     : string;
      var temp_str2     : string;
      var temp_bt_name1 : list of base_track_name_type;
      var temp_bt_name2 : list of base_track_name_type;
      var temp_bt1      : base_track;
      var temp_bt2      : base_track;
      var cur_proc      : int;
      if ind == 0 then {
         -- for the first track, any value of the first track
         temp_bt_name1 = bts_item_illegal(t,ind,top,ins_num);
         temp_str1 = append(temp_str1,"(",t.as_a(string),"_t",ind," in [");
         for each (b1) in temp_bt_name1 do {
             if index != 0 then {
                temp_str1 = append(temp_str1,",");
             };
             temp_str1 = append(temp_str1,b1.as_a(string));
          };
         temp_str1 = append(temp_str1,"])");
      }else {
         -- for the other track, there are dependent on the previous track
         temp_bt_name1 = bts_item_illegal(t,ind-1,top,ins_num);
         for each (b1) in temp_bt_name1 do {

            if index != 0 then {
                temp_str1 = append(temp_str1,"and");
            };

            temp_str1 = append(temp_str1,"(",t.as_a(string),"_t",ind-1," in [",
                               b1.as_a(string),"] => ",t.as_a(string),"_t",ind," in
["");
            if b1 == EMPTY then {
               temp_str1 = append(temp_str1,b1.as_a(string));
            } else {
               cur_proc =
(str_replace(b1.as_a(string),"/(.*)_(.*)_(.*)/","\3")).as_a(int);
               temp_str2 = str_replace(b1.as_a(string),"/(.*)_(.*)_(.*)/","\1_\2");
               temp_bt1 =
deep_copy(bts_global.key(temp_str2.as_a(base_track_name_type)));
               if temp_bt1.track_kind == END then {
                  temp_str1 = append(temp_str1,"EMPTY");
               }else {
                  temp_bt_name2 = temp_bt1.get_internal_next();
                  for each (b2) in temp_bt_name2 do {
                     if index != 0 then {
                        temp_str1 = append(temp_str1,",");
                     };
                     temp_bt2 = new base_track;
                     temp_bt2 = deep_copy(bts_global.key(b2));
                     temp_bt2.refine_track(cur_proc);
```

```
                temp_str1 = append(temp_str1,(temp_bt2.id_name).as_a(string));
              };
            };
          };
          temp_str1 = append(temp_str1,"])");

        };
      };
      return temp_str1;
    };
};
'>
```

D.7 Fichier MAC utilities

```
------------------------------------------------------------------------------
-- Title         : Mac utilities
-- Project       : MAC (Module d'Analyse de la Couverture)
------------------------------------------------------------------------------
-- File          : mac_utilities.e
-- Author        : Sebastien REGIMBAL
-- Created       : 17/11/2002
-- Last modified : 17/11/2002
------------------------------------------------------------------------------
-- Description   :
-- The module includes some utility structures used inside the coverage
-- analysis module
--
-- The following elements are included:
--
-- A Fifo template
--     the fifo is defined with a macro and it can be used to work any kind of
--     data structure
------------------------------------------------------------------------------

    module: mac_utilities
    aspect: coverage

------------------------------------------------------------------------------
-- name: fifo template
-- description:
-- A generic implementation of a fifo
-- to use this structure, you need to use the macro build_fifo_<the type of
-- data structure you want to put inside the fifo>
------------------------------------------------------------------------------
<'


define <fifo'statement> "build_struct fifo_<Type>" as {
  struct fifo_<Type> {
    ------------------------------------------------------------
    -- Class Members
    ------------------------------------------------------------
    -- Actual List
    !entries              : list of <Type> ;
    -- Name of Queue
    name                  : string;
    keep soft name == "Queue";
    -- Max Allowable Queue Size
    max_entries           : uint;
    keep soft max_entries == 65536;
    -- Threshold defining almost full
    almost_full_threshold  : uint;
    keep soft almost_full_threshold  == max_entries;
    -- Threshold defining almost empty
    almost_empty_threshold : uint;
    keep soft almost_empty_threshold == 0;
    -- Max Size attained this run
    max_curr_size         : uint;
    keep soft max_curr_size == 0;
    -- Verobosity
    verbose               : bool;
    keep soft verbose  == FALSE;

    -- Event for Empty Condition
    event                 empty      ;
    -- Almost Empty Event
    event                 almost_empty;
    -- Almost full Event
    event                 almost_full;
    -- Full Event
    event                 full       ;
    -- Fifo Overflow Occurred
    event                 overflow   ;
    -- Fifo Overflow Occurred
    event                 underflow  ;

    ------------------------------------------------------------
    -- Class Methods
    ------------------------------------------------------------
    -- Set the Name
    set_name( val : string ) is inline {
      name = val;
    };
    -- Return the Name
    name() : string is inline {
      result = name;
    };
    -- Set Max Queue Entries
    set_max_entries( val : uint ) is inline {
      max_entries = val;
    };
    -- return the Maximum Queue Entries
    max_entries() : uint is inline {
      result = max_entries;
    };
    -- Set the Almost Full Threshold
    set_almost_full_threshold( val : uint ) is inline {
      almost_full_threshold = val;
    };
    -- Return the Almost Full Threshold
    almost_full_threshold() : uint is inline {
      result = almost_full_threshold;
    };
    -- Set the Almost Empty Threshold
    set_almost_empty_threshold( val : uint ) is inline {
```

```
  almost_empty_threshold = val;
};
-- Return the Almost Empty Threshold
almost_empty_threshold() : uint is inline {
  result = almost_empty_threshold;
};
-- Reset the Max Curr Size
reset_max_curr_size() : uint is inline {
  max_curr_size = 0;
};
-- Return the Max Curr Size
max_curr_size() : uint is inline {
  result = max_curr_size;
};
-- Enqueue an Element
add(val : <Type>) is {
   if (entries.size() < max_entries) then {
      if (verbose) then {
         outf("%12d ns (%s): Note: Adding Entry to Fifo\n",sys.time,name);
         print val;
      };
      -- Set MaxSize
      entries.add(val) ;
      -- Set MaxSize
      if (entries.size() > max_curr_size) then {
         max_curr_size += 1;
      };
      -- Emit Full
      if (entries.size() == max_entries) then {
         emit full;
      };
      -- Emit Almost Full
      if (entries.size() == almost_full_threshold) then {
         emit almost_full;
      };
   } else {
      emit overflow;
      outf(appendf("%12d ns (%s): ERROR Tried to add element to \
                    Queue but queue is already full",sys.time,name));
   };
};

pop() : <Type> is {
   if (entries.size() == 0) then {
      emit underflow;
      outf(appendf("%12d ns (%s): ERROR Tried to dequeue element \
                    from Queue but queue is already empty",sys.time,name));
   } else {
      result = entries.pop0() ;
      if (entries.size() == 0) then {
         emit empty;
      };
      if (entries.size() == almost_empty_threshold) then {
         emit almost_empty;
      };
   };
};
-- return the current size of the queue
size() : uint is {
   result = entries.size();
};
-- Print the Contents of the Queue
print() is {
   out(appendf("(%s) HEAD OF LIST",name));
   for each (p) in entries do {
      print index,p;
   };
   out(appendf("(%s)TAIL OF LIST",name));
};
-- Is the Queue Full
is_full() : bool is inline {
  result = ((entries.size() == max_entries));
};
-- Is the Queue Empty
is_empty() : bool is inline {
   result = ((entries.size() == 0));
};
--Is the Queue Almost Empty
is_almost_empty() : bool is inline {
   result = ((entries.size() <= almost_empty_threshold));
};
-- Is the Queue Almost Full
is_almost_full() : bool is inline {
   result = ((entries.size() >= almost_full_threshold));
};

monitor_empty() @ sys.any is {
  while (TRUE) do {
     wait @empty;
     if (verbose) then {
      outf("%12d ns (%s) Note: Fifo is now empty!", sys.time, name);
     };
     -- Use is also to Extend this method to add more functionality
  }; -- End While
}; -- Method Monitor Empty


monitor_full() @ sys.any is {
   while (TRUE) do {
      wait @full;
      if (verbose) then {
         outf("%12d ns (%s) Note: Fifo is now full!", sys.time, name);
      };
      -- Use is also to Extend this method to add more functionality
   }; -- End While
}; -- Method Monitor Empty

monitor_almost_empty() @ sys.any is {
   while (TRUE) do {
      wait @almost_empty;
      if (verbose) then {
         outf("%12d ns (%s) Note: Fifo is now Almost Empty!",
              sys.time, name);
      };
      -- Use is also to Extend this method to add more functionality
   }; -- End While
}; -- Method Monitor Empty

monitor_almost_full() @ sys.any is {
   while (TRUE) do {
```

```
            wait @almost_full;
            if (verbose) then {
               outf("%12d ns (%s) Note: Fifo is now Almost full!",
                    sys.time, name);
            };
            -- Use is also to Extend this method to add more functionality
         }; -- End While
      }; -- Method Monitor Empty

      monitor_underflow() @ sys.any is {
         while (TRUE) do {
            wait @underflow;
            if (verbose) then {
               outf("%12d ns (%s) Error: Fifo just had an underflow condition!",
                    sys.time, name);
            };
            -- Use is also to Extend this method to add more functionality
         }; -- End While
      }; -- Method Monitor Empty

      monitor_overflow() @ sys.any is {
         while (TRUE) do {
            wait @overflow;
            if (verbose) then {
               outf("%12d ns (%s) Error: Fifo just had an overflow condition!",
                    sys.time, name);
            };
            -- Use is also to Extend this method to add more functionality
         }; -- End While
      }; -- Method Monitor Empty

//       run() is also {
//          start monitor_empty();
//          start monitor_full();
//          start monitor_almost_empty();
//          start monitor_almost_full();
//          start monitor_underflow();
//          start monitor_overflow();
//       };

   }; -- End Template Class for Fifo

}; -- End Build_Struct Macro
'>
```