Titre: Title:	Synthèse assistée de moniteurs d'assertions à partir d'une méthodologie d'encapsulation d'assertions dans une spécification exécutable
Auteur: Author:	Jean-François Lemire
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Lemire, J.-F. (2003). Synthèse assistée de moniteurs d'assertions à partir d'une méthodologie d'encapsulation d'assertions dans une spécification exécutable [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7131/

In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

UNIVERSITÉ DE MONTRÉAL

SYNTHÈSE ASSISTÉE DE MONITEURS D'ASSERTIONS À PARTIR D'UNE MÉTHODOLOGIE D'ENCAPSULATION D'ASSERTIONS DANS UNE SPÉCIFICATION EXÉCUTABLE

JEAN-FRANÇOIS LEMIRE
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

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

Your file Votre référence
ISBN: 0-612-86409-X
Our file Notre référence
ISBN: 0-612-86409-X

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

SYNTHÈSE ASSISTÉE DE MONITEURS D'ASSERTIONS À PARTIR D'UNE MÉTHODOLOGIE D'ENCAPSULATION D'ASSERTIONS DANS UNE SPÉCIFICATION EXÉCUTABLE

présenté par: LEMIRE Jean-François

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. MULLINS John, Ph.D., président

M. BOIS Guy, Ph.D., membre et directeur de recherche

M. ABOULHAMID El Mostapha, Ph.D., membre et codirecteur de recherche

Mme. BOUCHENEB Hanifa, Ph.D., membre

REMERCIEMENTS

Je tiens à remercier ceux et celles qui ont, de près ou de loin, contribué à ce projet de recherche. Plus particulièrement, je tiens d'abord à remercier mon directeur de recherche M. Guy Bois pour le temps qu'il a consacré à l'avancement du projet ainsi que pour la confiance qu'il m'a accordée tout au long de mon cheminement. Ses judicieux conseils, sa très grande disponibilité ainsi que sa jovialité ont été des facteurs déterminants qui ont soutenu ma motivation tout au long du projet. Un énorme merci à M. Yvon Savaria pour son précieux encadrement, ses judicieux conseils ainsi que pour le temps qu'il a consacré au projet. J'aimerais remercier également mon codirecteur M. El Mostapha Aboulhamid pour ses judicieux conseils. J'ai grandement apprécié mon expérience aux études supérieures et je suis reconnaissant envers mon directeur et envers tous les professeurs qui ont investi leur temps pour me permettre de réaliser mes objectifs. Encore une fois, merci.

J'aimerais également remercier mon collègue Sébastien Regimbal pour son aide à la réalisation de ce projet ainsi qu'à la rédaction d'articles. Je tiens aussi à remercier l'entreprise PMC-Sierra, et plus particulièrement M. André Baron pour le support qu'il a généreusement accordé au projet ainsi que pour ses judicieux conseils pratiques.

Je tiens aussi à remercier le ReSMiQ pour la bourse qu'il m'a octroyée. J'aimerais en plus remercier PMC-Sierra, Micronet, et le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour avoir supporté financièrement le projet.

J'aimerais réserver un merci particulier à mes parents et à ma sœur qui n'ont cessé de m'encourager à atteindre mes objectifs. Votre support n'a été autre qu'essentiel. Enfin, je réserve ce dernier merci tout spécial à mon épouse Mai Ly pour ses mots d'encouragement ainsi que son aide plus que précieuse.

RÉSUMÉ

La vérification fonctionnelle est devenue un sérieux goulot d'étranglement dans tout processus de développement microélectronique. De nouvelles méthodologies sont requises afin d'améliorer le processus de conception de bancs d'essai ainsi que la qualité de la vérification. Ce qui était, il n'y a pas si longtemps, un domaine où des méthodes *ad hoc* étaient couramment utilisées est maintenant devenu un domaine clé pour introduire de nouvelles pratiques et de nouveaux standards. Une méthodologie de vérification axée sur l'utilisation d'assertions aide à formaliser le processus de vérification et permet de rehausser la qualité de la vérification.

L'objectif de ce projet est de proposer une méthode pour accélérer et optimiser la conception d'assertions lors de l'implantation de bancs d'essai en langage de vérification de modèles matériels (*HVL*). L'implantation de moniteurs d'assertions dans un banc d'essai est un processus long et sujet aux erreurs. Ainsi, un outil permettant la création automatique d'assertions en langage *e* est proposé. L'outil est au centre d'une méthodologie d'implantation d'assertions qui débute par la définition de celles-ci au niveau d'une spécification exécutable. Ainsi, un ensemble de types d'assertions permettant de spécifier de façon simple des comportements complexes, c'est-à-dire sans utiliser de primitives propres aux langages de vérification, est proposé. De façon plus précise, l'ensemble de types d'assertions proposé permet de spécifier des propriétés à différents niveaux d'abstraction d'une spécification SDL et regroupe les quatre types d'assertions suivants : assertions d'exclusion d'états, assertions de temps de préparation, assertions de temps de maintien et assertions de temps de validité d'un signal. L'utilisation de l'outil proposé permet de migrer automatiquement les assertions définies dans la spécification vers des moniteurs d'assertions utilisés dans un banc d'essai conçu en langage *e*. L'outil utilise des patrons d'implantation, définis au cours de ce travail de recherche, pour générer les quatre types d'assertions proposées. Ces patrons permettent

de générer des moniteurs d'assertions optimisés pour la réutilisation grâce à une méthodologie de partitionnement des aspects, aussi définie dans le cadre de ce travail de recherche, propres à l'implantation de tels moniteurs.

Dans le but d'expliquer la problématique actuelle de la vérification fonctionnelle et de bien positionner le projet dans ce domaine, un sondage sur les pratiques de la vérification fonctionnelle de modèles matériels en entreprise a été effectué et les résultats sont présentés. Les pratiques des répondants sont comparées aux diverses méthodologies proposées dans la littérature. On y voit entre autres qu'il n'existe aucun consensus sur le moment, à l'intérieur du processus de développement, où les répondants débutent l'implantation des bancs d'essais. Aussi, une méthodologie de vérification axée sur la simulation est encore beaucoup plus utilisée qu'une méthodologie de vérification tirant profit à la fois des méthodes formelles ainsi que de la simulation.

Les résultats de cette recherche démontrent, grâce à une preuve conceptuelle, qu'il est possible de créer des patrons d'implantation d'assertions en langage de vérification de modèles matériels. Aussi, cette recherche révèle la possibilité d'automatiser la création de moniteurs d'assertions à partir de patrons d'implantation prédéfinis. Ceci permet de réduire le temps nécessaire à l'implantation d'assertions et, de ce fait, permet de réduire le temps nécessaire à la conception de bancs d'essai. En effet, une comparaison des temps requis pour une implantation manuelle et une implantation automatique, à l'aide de l'outil proposé, d'un même ensemble d'assertions, pour la vérification de deux modèles matériels, démontre une diminution du temps d'implantation par un facteur d'au moins 20. En plus, cette recherche démontre l'intérêt d'encapsuler des assertions dans une spécification exécutable car ceci permet de spécifier des comportements particuliers dans une représentation d'un modèle qui sera utilisée par plusieurs partis impliqués dans le processus de développement. Finalement, cette recherche expose une méthode efficace pour maximiser le potentiel de réutilisation des moniteurs d'assertions contenus dans un banc d'essais.

ABSTRACT

Functional verification has become an important bottleneck in every microelectronics development process. One of today's greatest verification challenges is to provide new methodologies and more automation to reduce the time required to create functional verification environments and to perform the verification task. What was not so long ago an *ad hoc* process is nowadays one of the most interesting fields to integrate new standards and structured practices. An assertion-based verification methodology helps formalize the verification process and enhances the quality of the verification task.

The main objective of this project is to propose a methodology to accelerate and optimize assertions design during the testbench implementation process using Hardware Verification Languages (HVLs). Assertion monitors implementation in a testbench is a long and error-prone process. Thus, a tool, which enables the automatic creation of *e* language assertions, is proposed. This tool is at the center of an assertions implementation methodology, which begins by the definition of these assertions at an executable specification level. Hence, an assertion set which enables the simple definition of complex behaviors at different abstraction levels of an SDL specification is proposed. It is then possible to specify assertions without using HVL primitives. Precisely, the proposed assertion set regroup four assertion types: state exclusion assertions, setup time assertions, hold time assertions and pulse width assertions. The proposed tool allows the migration of the specification's assertions towards *e* assertion checkers in order to perform the verification task. The tool uses implementation patterns, which were defined during the course of this research project, to generate the assertion checkers. These patterns enable the implementation of checkers that are optimized for reuse by using an aspect partitioning methodology, which has also been defined during the course of this project.

To explain the actual functional verification issue and to position the project in this domain, a survey of the actual industry functional verification practice has been performed and the results are presented. The survey's results are compared with the actual methodologies proposed in the literature. We can observe that, according to our respondents, there exist no standard moment within the development flow to begin testbench design. Also, a simulation-based verification methodology is more used than a mixed formal-simulation verification methodology.

The project's results show, as a proof of concept, that it is possible to create HVL assertion implementation patterns and that the automation of HVL assertion checkers based on these implementation patterns is also possible. Hence, this provides a way to efficiently accelerate the assertions implementation process which is beneficial for the entire functional verification flow. Indeed, a comparison of the time required to manually and automatically implement a set of assertions, for the verification of two hardware designs, reveals an acceleration by a factor of at least 20. Furthermore, this project shows that it is interesting to encapsulate assertions in an executable specification because this process enables the specification of particular design behaviors at a level of abstraction that will be used by many different design development stakeholders. Finally, this research proposes a method to maximise the assertions reuse potential in a HVL testbench.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ACRONYMES

AOP	Aspect-Oriented Programming
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfert Mode
BFM	Bus-Functional Model
EDA	Electronic Design Automation
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HVL	Hardware Verification Language
IEEE	Institute of Electrical and Electronic Engineers
IP	Intellectual Property
OO	Object-Oriented
OOP	Object-Oriented Programming
OVL	Open Verification Library
PSL	Property Specification Language
RTL	Register Transfert Level
SDL	Specification and Description Language
SOC	System On a Chip
TB	Testbench
TCM	Time-Consuming Method (*e*)
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

LEXIQUE

Assertion — Expression d'un comportement attendu d'un modèle qui doit être vérifié (i.e. expression d'une propriété d'un modèle). L'objectif poursuivi par l'utilisation d'une assertion est d'assurer la cohérence entre l'intention d'un concepteur et ce qui est créé.

Banc d'essai — Simulation de l'environnement de travail réel d'un système dans le but de tester son bon fonctionnement.

Constructeur — Fonction membre d'une classe logicielle qui est automatiquement invoquée à chaque fois qu'un objet de cette classe est créé (instancié).

Couverture croisée — Enregistrement de combinaison de valeurs scalaires en un point précis dans le temps à des fins d'analyse de la couverture fonctionnelle.

Couverture d'items — Enregistrement de valeurs scalaires individuelles en un point précis dans le temps à des fins d'analyse de la couverture fonctionnelle. L'objectif de la couverture d'items est de s'assurer que toutes les valeurs intéressantes et importantes d'un item ont été observées lors d'une simulation.

Couverture de transitions	Enregistrement de valeurs scalaires en un point précis dans le temps dans le but de mesurer la présence ou la génération de séquences de valeurs. De façon conceptuelle, la couverture de transitions est semblable à la couverture de chemins dans une machine à états.
Environnement de vérification	Ensemble des modules logiciels utilisés pour la vérification.
Événement	Signal qui permet, par ses différents états, d'indiquer la situation ou l'évolution d'une partie d'un système.
Expression temporelle	Définition déclarative d'un comportement temporel. De façon plus précise, il s'agit de la description de comportements à l'aide de combinaisons d'événements et d'opérateurs temporels.
Fil d'exécution (*thread*)	Composante d'un processus, correspondant à une opération élémentaire effectuée dans un programme, et qui appartient à un seul processus.
Machine à états finis étendus	Structure dans laquelle le calcul est modélisé sous la forme d'une transition d'un état vers un autre parmi un nombre fini d'états. Dans le cas du langage SDL, une machine à états est étendue par l'ajout de variables. Ceci permet d'effectuer des décisions de transition d'états basées sur la valeur associée à une variable de façon ce que l'état suivant ne soit pas seulement déterminé par l'état présent et les valeurs d'entrée de la machine à état.

Machine à états finis Structure dans laquelle le calcul est modélisé sous la forme d'une transition d'un état vers un autre parmi un nombre fini d'états.

Modèle sous vérification Modèle à vérifier.

Moniteur d'assertion Module d'un banc d'essai constitué d'assertions qui vérifient automatiquement le comportement d'un modèle sous vérification.

Pointeur Élément de données (qui est lui-même une unité minimale d'information faisant partie d'un segment de données) indiquant la position d'un autre élément de données.

Processus Instance d'une suite d'instructions nécessaires à l'exécution d'une tâche.

Propriété Attribut comportemental utilisé pour caractériser un modèle. De façon plus formelle, une propriété est une collection de relations logiques et temporelles entre des expressions booléennes subordonnées, des expressions de séquences, et d'autres propriétés qui dans l'ensemble représentent un comportement.

Propriété intellectuelle Bloc de conception dont la réutilisation pour fabriquer de nouveaux circuits est protégée par les règles de la propriété intellectuelle.

Propriété de sûreté — Propriété qui spécifie un invariant pour tous les états d'un modèle. L'invariant n'est pas nécessairement limité à un cycle mais il est limité dans le temps. De façon plus formelle, une propriété de sûreté est une propriété pour laquelle tout chemin violant la propriété possède un préfixe fini de façon à ce que chaque extension du préfixe viole la propriété.

Propriété de vivacité — Propriété qui spécifie une fatalité qui est illimité dans le temps. De façon générale, une propriété de vivacité décrit que quelque chose de bien se produira fatalement. De façon plus formelle, une propriété de vivacité est une propriété pour laquelle tout chemin fini peut être étendu à un chemin qui satisfait la propriété.

Register Transfert Level (RTL) — Niveau d'abstraction associé à la description de modèles matériels où une correspondance exacte cycles par cycles ou états par états existe avec une implantation du modèle au niveau des portes logiques.

Scénario de test (*test case*) — Instructions détaillées qui décrivent les données de base, les procédures et les résultats prévus d'un test en particulier.

Sortie standard — En général, la sortie d'un programme est affichée à la sortie standard qui correspond à l'écran. La plupart des systèmes d'exploitation permettent de rediriger la sortie vers un fichier.

Temps de maintien (*hold time*) — Temps de stabilité requis pour un signal suite à l'activation d'un autre signal.

Temps de préparation (*setup time*)	Temps de stabilité requis pour un signal avant l'activation d'un autre signal.
Temps de validité (*pulse width*)	Temps de stabilité requis pour un signal.
Test	Opération destinée à contrôler la bonne exécution d'un programme dans son ensemble.
Time Consuming Method (TCM)	Fonction utilisée pour synchroniser des processus d'un programme *e* avec des processus ou des événements d'un modèle HDL.
Vérification boîte-blanche	Stratégie de vérification qui est utilisée dans le but de vérifier la structure interne d'un système afin d'assurer la bonne marche de son fonctionnement externe.
Vérification boîte-grise	Stratégie de vérification qui est utilisée dans le but de vérifier des propriétés spécifiques à l'implantation d'un système en contrôlant et en observant celui-ci via ses entrées et ses sorties.
Vérification boîte-noire	Stratégie de vérification qui est utilisée dans le but de vérifier les entrées et les sorties d'un système sans se préoccuper de son fonctionnement interne.
Vérification de bas en haut (*bottom-up verification*)	Stratégie de vérification consistant à vérifier les composants d'un modèle de façon individuelle et à vérifier les fonctionnalités du système en intégrants les composants un à un dans l'environnement de vérification.

Vérification fonctionnelle	Processus permettant de vérifier qu'un modèle possède les fonctionnalités décrites dans la spécification.

LISTE DES ANNEXES

INTRODUCTION

Assurer l'exactitude fonctionnelle d'un modèle de conception matérielle est une étape laborieuse dans tout projet de conception microélectronique. Une implantation d'un modèle ne peut être exacte, sans erreur, par elle-même mais elle peut être exacte relativement à une spécification, une description de ce que l'implantation doit accomplir. Un des grands défis de la vérification matérielle basée sur la simulation est la génération d'un très grand nombre de stimuli de qualité, en déployant un effort minimal, dans le but de découvrir des erreurs qui se seraient glissées dans le modèle à vérifier. Un autre défi relié à la vérification fonctionnelle consiste en l'émulation précise de l'environnement d'un système sous vérification à l'aide d'un banc d'essai. Considérant que la vérification peut consommer jusqu'à 70% de l'effort dédié au développement d'un projet microélectronique, des méthodes efficaces pour accélérer l'implantation d'environnements de vérification ainsi que pour améliorer la qualité du processus de vérification sont requises.

Maximiser l'étendu de la réutilisation des parties d'un banc d'essai à l'intérieur ou même à l'extérieur d'un projet constitue un objectif fondamental pour réduire le temps accordé au processus de vérification. Lorsque des nouveaux composants d'un banc d'essai sont conçus, une approche de conception pour la réutilisation doit être considérée. Concevoir pour la réutilisation requiert l'application de solides concepts et de principes de conception logicielle. Ainsi, la vérification fonctionnelle est devenue une tâche logicielle très complexe.

L'implantation de bancs d'essai requiert l'utilisation de langages de programmation spécialisés. Les langages de description matérielle (*HDL*) comme le VHDL ou Verilog ont amélioré le processus de conception microélectronique en permettant de représenter la fonctionnalité d'un modèle d'implantation à un haut niveau

d'abstraction. Par contre, puisque la vérification fonctionnelle requiert des mécanismes logiciels complexes, les langages *HDL* ne sont pas idéals pour effectuer cette tâche. Certains langages reconnus tel C++ et Java, supportent des structures de données et des structures algorithmiques complexes nécessaires à la vérification fonctionnelle. Par contre, ces langages ne s'interfacent que très difficilement avec des modèles *HDL*. Certaines alternatives sont disponibles au concepteur de bancs d'essai pour contourner ces problèmes. D'une part, il est possible d'utiliser TestBuilder qui consiste en une bibliothèque de classes C++ qui permet le développement de bancs d'essai complexes. D'autre part, l'utilisation de langages de vérification matérielle (*HVL*) tels le langage *e* de Verisity ou OpenVera de Synopsys constituent d'autres alternatives. Ces langages offrent le support logiciel nécessaire à l'implantation de bancs d'essais complexes. Par exemple, l'utilisation du langage *e* permet de générer rapidement des vecteurs de tests grâce au moteur de génération pseudo-aléatoire de l'outil de simulation Specman Elite qui exécute le code *e*. En plus, ce langage possède les mécanismes nécessaires à l'établissement de modèles de couverture fonctionnelle. Pour sa part, Specman Elite possède des mécanismes d'analyse de la couverture fonctionnelle. Aussi, puisque cet outil s'exécute de façon concurrentielle avec un simulateur HDL, la valeur de chaque signal d'un modèle HDL sous vérification peut être échantillonnée et forcée à une valeur précise ce qui rehausse l'observabilité et la contrôlabilité de ce modèle.

Par ailleurs, les langages de vérification permettent aux concepteurs d'implanter des moniteurs d'assertions à l'intérieur des bancs d'essais. Les moniteurs d'assertions permettent de rehausser la qualité du processus de vérification en analysant les comportements du modèle HDL et en rapportant toute violation. Aussi, les moniteurs d'assertions réduisent le temps requis pour mettre au point un modèle puisqu'ils peuvent être utilisés soit comme une technique de vérification boîte-noire, soit comme une technique boîte-blanche pour ainsi accroître la visibilité du modèle sous vérification. Cependant, même si la vérification basée sur des assertions est une méthode efficace pour vérifier des comportements complexes, notre expérience à coder des assertions montre qu'il est très difficile de coder des assertions efficaces. En plus, le processus

d'implantation d'assertions se révèle très sensible aux erreurs. En d'autres termes, il est facile de coder des assertions qui ne détecteront pas d'erreurs alors qu'elles sont supposées en détecter (erreurs fausses-positives), ou encore de coder des assertions qui détecteront de fausses erreurs (erreurs fausses-négatives). Pour ces raisons, le temps nécessaire pour déverminer des assertions est très significatif. Actuellement, aucun langage ne peut assurer la qualité d'un environnement de vérification. Conséquemment, de nouvelles méthodologies de conception de bancs d'essai supportées par ces langages sont tout de même requises.

Parallèlement, un élément clé d'un processus de développement consiste à l'intégration de la conception et de la vérification de manière concurrente. Lorsque la spécification est approuvée, la conception des bancs d'essais doit débuter en même temps que l'étape de développement. De façon idéale, lorsqu'un modèle RTL est prêt à être vérifié, des bancs d'essais sans erreurs devraient aussi être disponibles. La vérification du modèle RTL progresse ainsi beaucoup plus rapidement car le travail n'est pas retardé par la mise au point des bancs d'essai. Une spécification exécutable est un jalon dans l'établissement d'un processus parallèle de conception et de vérification car elle ne permet, en général, aucune ambiguïté dans son interprétation; ce qui n'est pas le cas avec des spécifications écrites en langage naturel car elles sont typiquement incomplètes et ambiguës. Une spécification exécutable peut représenter le comportement d'un système à différents niveaux d'abstraction et est vérifiable par simulation. Plusieurs langages standards sont disponibles pour spécifier le comportement d'un système à un haut niveau d'abstraction. C/C++ et SDL sont des exemples de langage de spécifications exécutables standards. En plus, SystemC, une bibliothèque de conception et de validation basée sur C++, supporte les étapes de conceptualisation ainsi que l'implantation autant matérielle que logicielle. SDL est un langage standard de spécification graphique et textuel. Ce langage est de plus en plus utilisé avec UML spécialement pour spécifier des actions d'un système opérant en temps réel. Ainsi, SDL est un langage approprié pour décrire des applications embarquées.

L'objectif premier de ce travail est de prouver qu'il est possible de simplifier et d'accélérer la création d'assertions réutilisables en langage *e* pour la vérification fonctionnelle par le biais d'un processus automatique de synthèse assistée d'assertions. Ce processus supporte une méthodologie d'implantation de bancs d'essai permettant :

- De définir les assertions à vérifier au niveau d'une spécification SDL exécutable.
- De simplifier l'implantation, dans un banc d'essai, des assertions définies dans la spécification en utilisant un outil permettant leur migration automatique vers des moniteurs d'assertions en langage *e*.
- De maximiser la réutilisation des assertions implantées en langage *e*.

Ce travail a été réalisé en deux étapes importantes. Tout d'abord, un sondage sur les pratiques de vérification de modèles matériels en entreprise a été effectué dans le but d'expliquer la problématique actuelle de la vérification fonctionnelle et de bien positionner le projet dans ce domaine. L'objectif principal du sondage était de répondre à ces deux questions:

1. Comment aborde-t-on et exécute-t-on la vérification fonctionnelle de modèles matériels en entreprise?
2. Existe-t-il une méthode intuitive ou efficace pour vérifier?

Les résultats de ce sondage sont présentés pour fournir une revue des méthodes de vérification utilisées dans l'industrie et les résultats sont comparés aux diverses méthodologies proposées dans la littérature.

Ensuite, une méthodologie d'encapsulation d'assertions dans une spécification SDL est présentée. Les assertions implantées en SDL permettent de définir des comportements complexes au niveau de la spécification sans pour autant coder les assertions à partir de primitives propres aux langages de vérification. Définir une méthodologie d'encapsulation d'assertions dans une spécification standard au niveau système permet de spécifier plus précisément des propriétés d'un système dans une

représentation qui sera utilisée par plusieurs personnes intéressées au développement. Un premier ensemble d'assertions, pour spécifier des comportements du système à haut-niveau, est inséré durant la capture de la spécification. Ensuite, lors du processus de raffinement de la spécification, de nouvelles assertions, pour spécifier des comportements à un niveau plus bas, peuvent être ajoutées. Quatre types d'assertions ont été sélectionnés pour être encapsulées dans la spécification. Plus précisément, la méthodologie présentée supporte des assertions d'exclusion d'états, des assertions de temps de préparation, des assertions de temps de maintien ainsi que des assertions de temps de validité d'un signal. Un outil est ensuite présenté dans le but d'assister le concepteur à la migration automatique des assertions définies dans la spécification vers des moniteurs d'assertions en langage *e*. Ce processus facilite l'implantation des bancs d'essai en utilisant directement l'information contenue dans une spécification pour créer des modules d'un banc d'essai. Des patrons d'implantation d'assertions en langage *e* sont utilisés pour créer les moniteurs d'assertion du banc d'essai. Ainsi, le potentiel de réutilisation des moniteurs d'assertions est optimisé d'abord en implantant des moniteurs génériques puis en les adaptant par la suite aux détails d'implantation spécifiques du système à vérifier. Cette méthodologie permet une séparation claire des aspects d'implantation des assertions, ce qui rehausse la réutilisation.

À l'aide d'un exemple concret, il est démontré qu'avec la méthodologie supportée par l'outil proposé, les concepteurs de système peuvent définir des assertions pour spécifier des comportements tant à haut-niveau qu'à bas niveau, dans une spécification exécutable, sans coder celles-ci à partir de primitives temporelles propres aux langages de vérification. Aussi, les bénéfices de la méthodologie proposée pour résoudre le problème de la conception des moniteurs d'assertions efficaces et réutilisables dans un environnement de vérification en *e* sont démontrés.

En résumé, les contributions originales de ce travail de recherche sont :

- L'établissement d'une base de connaissances sur la pratique de la vérification dans l'industrie par l'analyse des résultats d'un sondage conçu et publié par le groupe de recherche en microélectronique de l'École Polytechnique de Montréal.
- L'établissement d'une preuve de concept portant sur la génération assistée de moniteurs d'assertions à partir d'une spécification exécutable. Ceci implique :
    - L'établissement d'une méthodologie d'encapsulation d'assertions dans une spécification SDL dans le but de rehausser le niveau de détail de la spécification.
    - L'implantation d'un outil de génération assistée de moniteurs d'assertions pour des bancs d'essai.
    - L'établissement d'une méthodologie de partitionnement des aspects d'un environnement de vérification dans le but de rehausser le niveau de réutilisation du code du banc d'essai.

Ce projet a mené à la publication de trois articles. D'abord, (Regimbal, Lemire, Savaria, Bois, Aboulhamid et Baron 2002a) et (Regimbal, Lemire, Savaria, Bois, Aboulhamid et Baron 2002b), présentés aux annexes E et F, exposent une méthodologie de partitionnement par aspects pour rehausser la réutilisation lors de la conception d'environnements de vérification. Enfin, (Lemire, Regimbal, Bois, Savaria, Aboulhamid, Baron 2003) présente la méthodologie de synthèse d'assertions assistée à partir d'une spécification SDL.

Le corps de ce mémoire par article contient quatre chapitres. Il est principalement constitué des textes de deux articles. Le premier article, présenté au chapitre 2, dévoile les résultats et l'analyse d'un sondage sur les méthodes de vérification fonctionnelle en entreprise. Le deuxième article, présenté au chapitre 4, explique la méthodologie de synthèse d'assertions assistée à partir d'une spécification SDL exécutable. Les chapitres 1 et 3 présentent une synthèse de ces deux articles, discutent de leur pertinence dans le contexte de ce projet de recherche et présentent une analyse détaillée des résultats.

CHAPITRE 1

LA VÉRIFICATION FONCTIONNELLE (REVUE DE LITTÉRATURE ET SYNTHÈSE DE L'ARTICLE PRÉSENTÉ AU CHAPITRE 2)

Ce chapitre présente principalement l'article qui fera l'objet du deuxième chapitre de ce mémoire ainsi qu'une discussion générale des résultats qui ont mené à l'élaboration de celui-ci. Il présente aussi les notions et principes de la vérification fonctionnelle qui représente le sujet principal de l'article.

1.1 Formulation du problème de la vérification

L'exactitude d'un code HDL n'est pas une propriété absolue mais une propriété relative. Une description HDL ne peut être exacte, c'est-à-dire sans erreur, d'un point de vue absolu mais elle peut être exacte d'un point de vue relatif à une description de ce que l'usager voudrait qu'elle accomplisse (Bergeron 2003b, Bergeron 2000b, Bening, Foster 2001).

En général, on produit une spécification qui représente une compréhension informelle de ce que l'usager voudrait que l'implantaiton accomplisse. Lorsqu'un modèle est vérifié relativement à une spécification, deux modèles sont fondamentalement comparés. Donc, la vérification peut être définie comme étant l'action de comparer deux modèles. Soit $m*$ qui représente une spécification. Habituellement, $m*$ est implanté dans un langage connu de tous (ex : français, anglais etc.). Aussi, la spécification $m*$ peut être représentée sous forme d'un programme écrit dans un langage quelconque. À partir de la spécification, on génère un autre modèle, m, qui lui représente la source de l'implantation

du design. Dans le domaine de la conception matérielle, ce modèle *m* est une description HDL basée sur la spécification *m**.

Soit maintenant *f* une fonction calculée par un modèle *m*. Il est alors possible d'imaginer l'existence d'une «version exacte» *f** de *f* qui correspond à ce que *m* devrait faire relativement à la spécification *m**. Pour démontrer que *m* est exact, il est nécessaire de prouver que *f* est équivalente à *f**. La connaissance de *f** provient en général de *m**, la spécification. Dans le domaine de la vérification fonctionnelle, on crée un banc d'essai à partir de *m** qui pour chaque valeur d'entrée *x* de *m*, est en mesure de déterminer si *m* calcule une valeur exacte pour *f(x)*, c'est-à-dire si *f(x)* = *f*(x)* (Howden 1987). La figure 1.1 inspirée de (Bergeron 2003b, Bergeron 2000b) montre comment un banc d'essai interagit avec un modèle sous vérification. Le banc d'essai est un programme qui dans le domaine de la vérification matérielle est communément implanté en VHDL (Cohen 1999), Verilog (Bening, Foster 2001), *e* (Verisity 2002, Verisity 2003b) ou OpenVera (Synopsys 2003a, Haque, Khan, Michelson 2001), mais il peut aussi inclure des fichiers de données externes ou des sous-programmes en langage C. Ce programme est utilisé pour créer une séquence d'entrées *x* prédéterminées appliquées au modèle sous vérification *m*. La réponse du modèle aux entrées appliquées est aussi observée. On voit qu'il s'agit d'un système complètement fermé. C'est-à-dire que le système contenant le banc d'essai et le modèle sous vérification ne possède aucune entrée et aucune sortie. Le banc d'essai modélise l'environnement d'opération du modèle sous vérification.

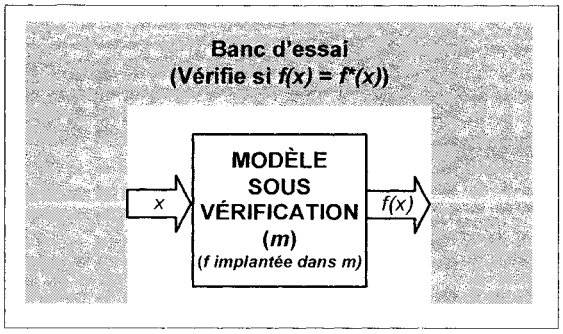


Figure 1.1 - Banc d'essai

1.2 Preuve d'exactitude d'un modèle

Il serait possible de prouver l'exactitude de tout modèle en le vérifiant à l'aide de toutes les valeurs appartenant à son domaine d'entrées. En général, les domaines sont infinis. C'est pour cette raison qu'en pratique, un processus de preuve d'exactitude ne pourra utiliser la finitude des domaines d'entrées. La vérification fonctionnelle permet de contourner ce problème en divisant le problème de la comparaison de modèles en une série de petits problèmes dont il est possible de développer une théorie efficace.

1.3 La vérification fonctionnelle

L'objectif principal de la vérification fonctionnelle est de s'assurer qu'un modèle implante bien les fonctionnalités attendues. Comme il est montré à la figure 1.2, la vérification fonctionnelle compare un modèle, qui constitue une réalisation de la spécification, avec sa spécification (Bergeron 2003b, Bergeron 2000b). Sans vérification fonctionnelle, il faudrait croire, par exemple, que la transformation d'une spécification en un modèle RTL a été effectuée correctement, sans interprétation erronnée de la spécification.

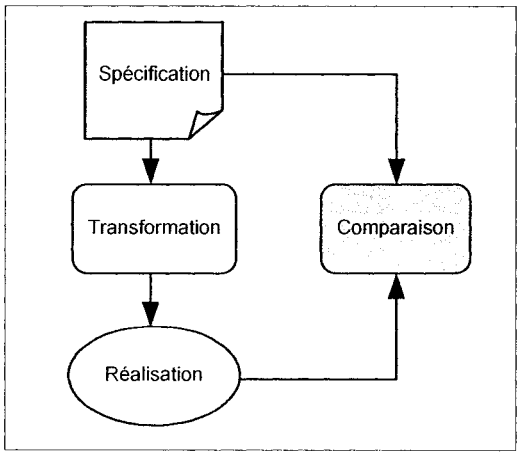


Figure 1.2 - La vérification fonctionnelle

1.4 Limites de la vérification fonctionnelle

Une des difficultés en vérification est de créer un modèle de référence qui capture complètement et correctement ce qu'on veut produire. Aussi, la vérification fonctionnelle pourrait ne pas détecter d'erreur dans un modèle pour les cas suivants (Howden 1987):

- Mauvaise identification des fonctions et des structures. Ceci survient lorsque l'analyse du modèle n'identifie pas toutes les fonctions et les structures utilisées par celui-ci.
- Banc de test incomplet.
- Mauvaise analyse des types de défaillances possibles.

1.5 Détails de soumission de l'article du chapitre 2

L'article présenté au chapitre 2 a pour titre « *A Survey on Current Functional Verification Practice* ». Il a été soumis au magazine *IEEE Design and Test of Computers* le 11 octobre 2002 et porte le numéro de manuscrit 117737. L'article est présentement en processus de recommandation finale par l'éditeur associé du magazine en vue d'une future publication.

1.6 Méthodologie de travail

La période de sondage a officiellement débutée le 4 juin 2002 et s'est terminée le 31 juillet 2002. Une implantation Web fut sélectionnée pour faciliter une publication à grande échelle ainsi que pour favoriser l'analyse automatique des résultats. Une version du sondage en format texte est présentée en annexe A. Le questionnaire était constitué de 45 questions. Les répondants ont été sondés sur les méthodes de vérification utilisées dans leur entreprise, leur opinion sur ce qui constitue un bon processus de vérification, ainsi que leur expérience technique.

Une invitation à répondre au sondage a été distribuée le 4 juin 2002 via le forum de discussion de la Guilde de Vérification (*Verification Guild*) vol. 3, no. 9 (Bergeron 2003a). L'invitation comportait un lien vers le site Web du sondage. À la fin de la période de sondage, 71 répondants avaient complété le questionnaire. Le sondage pouvait être complété de façon anonyme mais plusieurs répondants ont volontairement donné le nom de l'entreprise pour laquelle ils travaillaient. Ainsi, 31 différentes compagnies ont pu être identifiées comme ayant participé au sondage.

1.7 Survol de l'article

Pour établir le cadre d'application ainsi que définir la terminologie de ce projet de recherche, l'article débute par une revue des méthodes actuelles de vérification telles que proposées par la littérature. L'importance du parallélisme entre le déroulement des processus de conception et de vérification est exposée. À partir du moment où une spécification exécutable du modèle à implanter est disponible, la conception du plan de vérification et des bancs d'essai peut s'exécuter en même temps que l'implantation du code RTL du modèle. De façon idéale, lorsqu'une description RTL du modèle est disponible, des bancs d'essais sans défaut sont aussi disponibles. Ceci permet de diminuer grandement le temps de développement du projet.

Une stratégie de vérification optimale basée sur les techniques et technologies actuelles est présentée. Cette stratégie consiste principalement en une intégration de la vérification formelle et de la simulation. Ensuite, les différents niveaux de vérification sont présentés soient la vérification au niveau des blocs fonctionnels, la vérification au niveau de l'intégration des blocs fonctionnels, la vérification au niveau des ASIC, des FPGA et des blocs IP, et la vérification au niveau système.

Ensuite, une analyse exhaustive des résultats du sondage est présentée. Cette analyse offre une perspective générale des pratiques actuelles de vérification des répondants. L'analyse est partitionnée selon les différents sujets abordés dans le sondage soient le profil des répondants, le processus de vérification, les stratégies de vérification,

les méthodes d'auto-vérification, la couverture, la réutilisation ainsi que la vérification formelle. Enfin, une discussion sur l'implication des résultats du sondage est présentée. On y retrouve une comparaison entre les méthodes de vérification fonctionnelle proposées dans la littérature et les méthodes actuelles des répondants.

1.8 Principales conclusions et résultats

Cette recherche a permis d'obtenir des résultats significatifs au niveau de plusieurs sujets reliés à la vérification fonctionnelle. Il est intéressant de remarquer que certaines méthodes de vérification bien documentées dans la littérature ne sont pas ou peu utilisées par les répondants. Aussi, il est possible de remarquer que certaines autres pratiques sont bien établies car elles sont utilisées par la majorité des répondants.

On y voit particulièrement que la vérification fonctionnelle est au centre des préoccupations de l'ensemble des répondants. Aussi, une méthodologie de vérification axée sur la simulation est encore beaucoup plus utilisée qu'une méthodologie de vérification tirant profit à la fois des méthodes formelles et de la simulation. On y voit aussi que la majorité des répondants génèrent les stimuli de leurs bancs d'essai au niveau transactionnel plutôt qu'à un bas niveau d'abstraction. Les langages HDL sont encore très utilisés dans la conception de bancs d'essai. La majorité des répondants utilise une méthodologie de vérification basée sur l'implantation d'assertions (voir chapitre 3 pour plus de détails sur la vérification basée sur des assertions). L'utilisation de la couverture de code ainsi que la couverture fonctionnelle est fréquente. La métrique de couverture de code la plus utilisée est la couverture d'instructions. Au niveau de la vérification formelle, une majorité de répondants utilise la vérification formelle d'équivalence de modèles (*formal equivalence checking*) au profit des autres méthodes formelles. Aussi, un grand nombre de répondants n'utilise ni des méthodes avancées de conception logicielle, ni des méthodes avancées de réutilisation dans l'élaboration de leurs bancs d'essai, et ce malgré le fait que la majorité d'entre eux croit que la conception de bancs d'essai relève principalement du domaine du logiciel.

CHAPITRE 2

A SURVEY ON CURRENT FUNCTIONAL VERIFICATION PRACTICE

Jean-François Lemire[1],
Sébastien Regimbal[1], Guy Bois[1], Yvon Savaria[1] and El Mostapha Aboulhamid[2]
[1] Electrical Engineering Department, École Polytechnique de Montréal,
P.O. Box 6079, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3A7.
{lemire, regimbal, bois, savaria }@grm.polymtl.ca
[2] Department of Computer Science and Operational Research, Université de Montréal,
C.P. 6128, Succ. Centre-ville, Montréal, Québec, Canada, QC H3C 3J7.
aboulham@iro.umontreal.ca

2.1 Abstract

Functional Verification of hardware designs is more than ever treated as a vital part of their development process. The effort required to verify complex designs is increasing at an alarming rate. In the beginning of June 2002, the Groupe de Recherche en Microélectronique of École Polytechnique de Montréal published an academic survey on functional verification methodologies to get a glimpse at today's functional verification practices in the microelectronics industry. The purpose of this paper is to provide a detailed analysis of the survey's results and thus getting a snapshot of the sampled respondents' verification practices. The paper provides an analysis of the state-of-the-art verification methodologies as proposed in the literature. In addition, the survey results are compared to the proposed methodologies and the missing links to a more effective verification practice are identified. Our study led to interesting findings related to various functional verification topics, such as the verification process, formal and

assertion-based verification, coverage, reuse and verification strategies. We noticed that some verification practices are well established, and are used by the majority of our respondents. Also, we discovered that certain verification methods, well documented in the literature, are seldom used by a significant percentage of respondents.

2.2 Introduction

Considering that hardware verification has become the most critical bottleneck in the digital design flow, new approaches and methods are more than ever required. To deal with this issue, several verification methods and tools are currently developed to help verification engineers cope with this difficult task. What was not so long ago an *ad hoc* process is nowadays one of the most interesting fields to integrate new standards and structured practices. To help position our research with respect to current state-of-the-art, we created and published a survey on functional verification (Lemire 2002). Our main objectives were to answer two questions:

1) How is functional verification addressed and performed in industry?

2) Is there an intuitive and/or effective way to perform functional verification?

To establish the framework and terminology of this study, the paper begins with a description of a state-of-the-art verification methodology, as proposed in the literature. It then presents the detailed analysis of the results collected through the survey. This analysis provides a good overall perspective of our respondents' verification practices, which can be compared with current theory to highlight any discrepancy. The survey period began officially on June 4th 2002 and ended on July 31st 2002. We chose a web survey implementation method because of its inherent publication ease and its automatic results analysis possibilities.

The remainder of this paper is organized as follows: We will first discuss, in section 2.3, the theoretical verification flow proposed in the literature. Then, in section 2.4, we will briefly present the sample structure of the survey and, in section 2.5, we will provide a detailed analysis of our findings. In section 2.6, we will discuss the implications of these results in relation to the proposed theoretical flow and our previously defined objectives. Finally, we will summarize our main conclusions.

2.3 Verification flow according to current literature

Nowadays, design and verification methodologies are very important issues. Moreover, cost and time to market are predominant development success factors. This section presents a description of state-of-the-art verification methodologies, as proposed in the literature, to establish the framework and terminology of the survey. The survey results will be compared in section 2.6 with these methodologies.

2.3.1 Development Flow

A key point in a good development process is the integration of concurrent design/verification efforts and methodologies. Figure 2.1, inspired from (Sternberg 2000, Bergeron 2003b, James 1999), highlights the benefits of a concurrent approach. We see that once the specifications are approved, a verification effort begins in parallel with the design effort. The importance of a specification is summarized in (Bening, Foster 2001). The authors explain that the fundamental verification principle is to have a complete specification before RTL implementation to avoid unnecessarily complex and unverifiable designs. Furthermore, an executable specification allows no ambiguity in its interpretation, which is not the case with a natural language specification. This type of specification is directly verifiable through simulation, and hence is a milestone in the top down design of a system. Moreover, an executable specification can be used as a reference to generate the verification plan. Behavioral models are also created so that the simulation testbench can be debugged before the RTL is ready. These behavioral models are also available for verification in a bottom up fashion, where the testbench emulates

design blocks interfaces. They increase the thoroughness of the verification effort, and the speed at which developers can iterate through the process, due to faster simulation turn-around. Finally, once the specifications are approved, system engineers begin to work on the system-level testing and regression test suite.

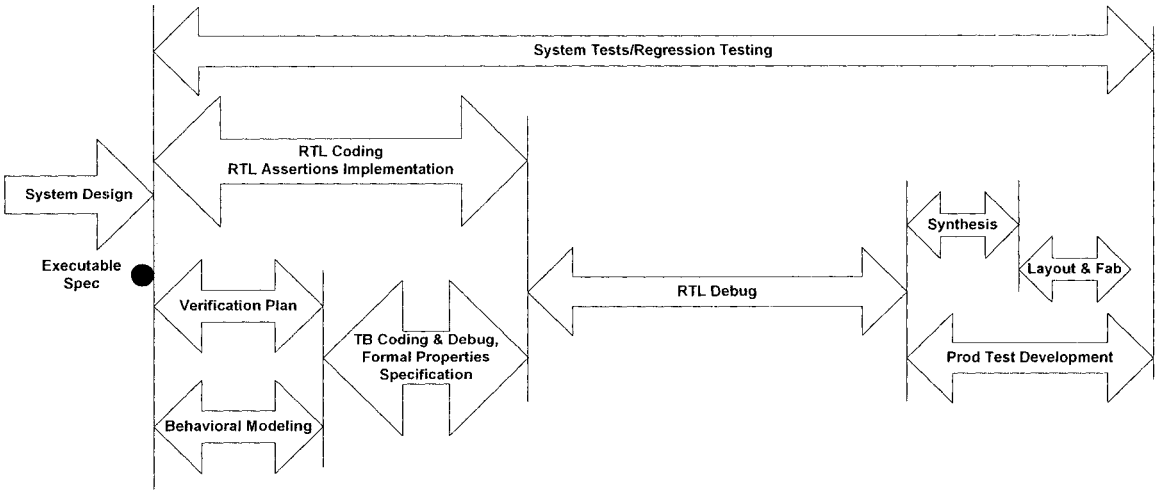


Figure 2.1 - Proposed Development Flow

Ideally, when a RTL description is available for verification, testbenches are already available for simulation. RTL verification progresses faster because the effort is not delayed by the debugging of testbenches. As mentioned earlier, in a bottom-up verification process, only the portion of the design being verified can be represented in RTL. The simulation time is of course reduced if behavioral models represent the remainder of the system.

2.3.2 Formal Techniques

Unfortunately, the traditional simulation approach of verification provides no assurance that all corner cases have been covered. Formal verification techniques have been developed. These techniques use mathematical proof, rather than simulation and test vectors, to provide a higher level of verification confidence on certain properties (Rashinkar, Paterson, Singh 2001, Bening, Foster 2001). Equivalence Checkers is a class of formal tools that mathematically proves the logical equivalence between different

refinements of a design without simulation and test vectors. Model checking is another class of formal verification tools that verifies if an implementation satisfies the properties of a specification. These two approaches should be treated as orthogonal verification process within a design flow because both the verification of circuit equality and the verification of circuit functionality are necessary in any development process (Bening, Foster 2001). To successfully apply model checking to the RT-level requires partitioning the large design into smaller verifiable blocks, and then creating a valid environment description for each partition (a set of constraints to model the block-level environment). Also, the properties to be verified are specified using assertions, which are a claim we make about an event or a sequence of events in a design.

2.3.3 Formal/Simulation-Based Functional Verification

Depending on the different stakeholders in the design and verification flow, there exist two complementary views for specifying properties with the use of assertions (Fitzpatrick, Foster, Marschner, Narain 2002, Foster 2002). A verification engineer needs an expressive means for specifying correct behavior, since his goal is to validate correct system behavior. On the other hand, the design engineer requires a convenient mechanism for expressing lower-level implementation properties, since his focus is on implementing RTL design descriptions using hardware description languages (HDLs). For this reason, both assertions embedded in the RTL code (implementation assertions) and assertions defined outside the RTL code (specification assertions) are necessary. The new Accellera's (Accellera 2003a) SystemVerilog provides constructs to define RTL assertions. Also, Accellera's Open Verification Library (Accellera 2003b) comprises interesting VHDL and Verilog assertions modules that can be instantiated in a HDL code. Assertions defined outside the RTL code are specified by the verification engineers and are well suited for specifying expectations of the design based on the functional intent or global properties that can be verified using simulation or formal techniques. Also, specifying assertions directly in the HDL code provides greater visibility into the design during system verification, which improves simulation-based methodologies and

provides an easier path to formal verification. Hence, as shown in figure 2.1, designers should implement assertions in the HDL code during the RTL coding phase and system architects or verification engineers should concurrently specify formal properties.

2.3.4 Traditional Simulation vs Formal Techniques

The techniques and technologies used when verifying functional behavior of a design are summarized in figure 2.2, where a formal/simulation based functional verification methodology is presented (Bergeron 2000b, Rashinkar, Paterson, Singh 2001, Haque, Khan, Michelson 2001, Bergeron 2003b, Bening, Foster 2001, Cohen 2001a).

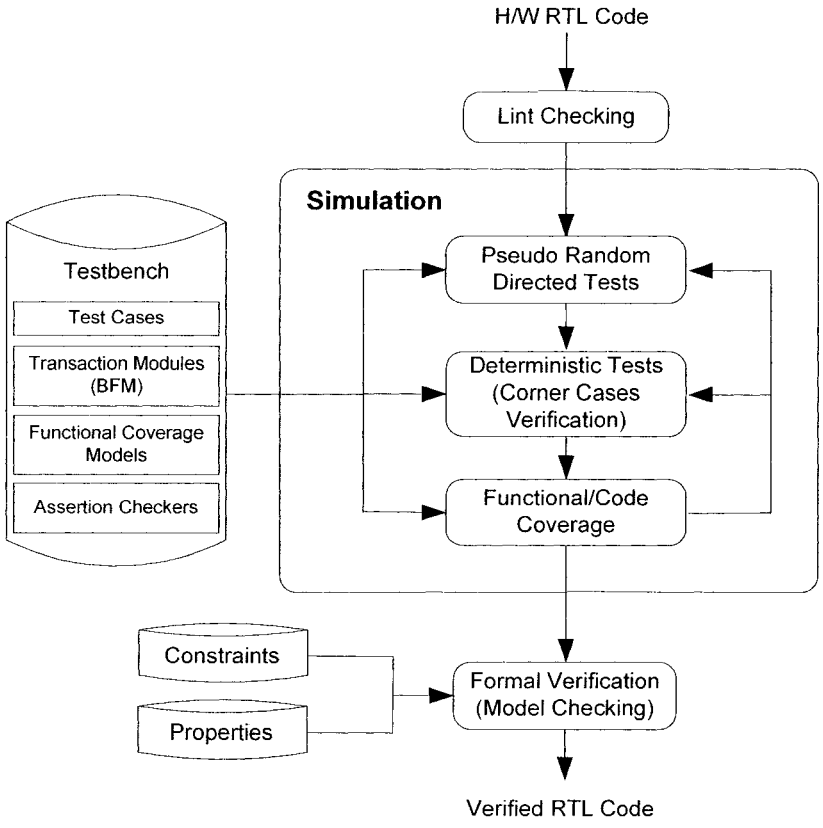


Figure 2.2 - Hardware Formal/Simulation-based verification

The RTL code goes through lint checking to ensure that no syntactical and minimal semantic (synthesis, simulation, formal verification) design violations exist within the

code. Functional simulation of the RTL design uses the system testbenches. A testbench consists of test cases, which represent groups of stimuli that require similar configuration or verification strategy to perform the simulation task. Also, the testbench contains transaction modules (Bus Functional Models) to enable a transaction-based verification methodology. Bus Functional Models (BFMs) enable the verification engineer to specify tests as a sequence of transactions at a high level of abstraction, instead of a set of lower level vectors. They convert high-level transactions into explicit lower-level signals and operations. The testbench also holds functional coverage models to allow functional coverage measurement and assertion checkers to verify design properties dynamically during simulation.

As shown in figure 2.2, the first step of simulation is to apply pseudo-random stimuli to fill the functional coverage models previously defined in the verification plan (Haque, Khan, Michelson 2001, Bergeron 2003b, James, Macionski 2001). This approach enables the design to be exercised under a large number of conditions. Hence, this will create input conditions that were not expected at the time the verification plan was written. If this methodology is properly applied, a more productive feature coverage should be measured in comparison with the traditional directed tests approach (Bergeron 2003b, James, Macionski 2001). The overall quality of the verification process is then enhanced. Corner cases verification (see figure 2.2) refers to the exercise of specific boundary conditions in the design with the use of deterministic tests. Code coverage and functional coverage identify respectively any non-stimulated design code and unverified functional behavior of the design. In other words, code coverage measures how much of the implementation has been exercised and functional coverage measures how much of the original design specification has been exercised (Bergeron 2003b). Functional coverage is essential to support a random-based verification approach since it is used to record which test cases and conditions where automatically created by the testbench pseudo-random stimuli generator. Formal model checking, the last verification phase in figure 2.2, is performed to verify behavioral properties of the blocks and hard-to-reach corners in critical and important design sections. As discussed earlier we see that the

model-checking tool uses constraints and properties of the design as inputs. After the above functional verifications tasks, logical equivalence should be addressed by performing the netlist verification. Afterwards, physical characteristics verification should be addressed by performing timing verification. Finally physical verification and device tests should be addressed to ensure correct chip implementation.

2.3.5 Black-box, White-box and Grey-box Strategies

There are three functional verification complementary approaches: black-box, white-box and grey-box (Bergeron 2000b, Rashinkar, Paterson, Singh 2001, Bergeron 2003b). With a black-box approach, the verification is performed without any knowledge of the internal details of a design. The verification is accomplished through the available interfaces because this approach does not provide insight into the design details. A white-box approach provides full visibility and controllability of the internal structure and implementation of the design. The stimulus for corner cases can be easily generated, and as the verification progresses, results can be easily observed. As results, any discrepancies from the expected behavior can be detected. The grey-box verification approach is a mix between white-box and black-box verification. The purpose of grey-box verification is to exercise significant features specific to the implementation by controlling and observing the design entirely through its top-level interfaces.

2.3.6 Hardware Verification Languages

Testbenches can be created in VHDL, Verilog or C++ languages. Also, the use of Hardware Verification Languages (HVL) addresses the requirements for complex data and algorithmic structures needed for functional verification. The e (Verisity 2003b) and Open Vera (Haque, Khan, Michelson 2001, Synopsys 2003a) languages are examples of HVLs available to verification engineers. Also, Cadence's SystemC Verification Library (Cadence 2003) is a C++ class library that extends C++ into an advanced testbench development language. These languages support a methodology that targets constrained random testing. In addition, both languages possess temporal structures built into them. Legacy testbenches created in VHDL, Verilog, and C/C++ can be used with testbenches

created with the e or Open Vera languages. These languages interface with HW/SW co-verification tools.

2.3.7 Levels of Abstraction

Figure 2.3 presents the different verification levels of abstraction inspired by (Bergeron 2000b, Rashinkar, Paterson, Singh 2001, Haque, Khan, Michelson 2001, Bergeron 2003b, Bening, Foster 2001). The different levels of abstraction presented are organized following a bottom-up verification approach, which is widely used today in many design companies. The first step (front-end acceptance in fig 2.3) is to validate the design data by parsing the files to ensure that their compatibilities with the target tools and that no syntactical and semantic violations exist within the code. Then, depending on the design abstraction level, the design is verified either at a functional block level, at the integration of functional blocks level, at an ASIC, FPGA or IP level, at a system level or at any combination of these levels. If the design is at a higher level of abstraction, it can proceed directly to system-level verification. At the functional block verification level, the individual components are verified in isolation. These functional design blocks are logical partitions. They are created to facilitate the implementation. Each component is tested thoroughly, independently of the environment into which it will be integrated. The interoperation between functional blocks is verified at the integration of functional blocks level. ASICs and FPGAs are physical partitions. They form a natural partition for verification. Also, IPs are reusable components that are intended to be used as-is and unchanged in many different designs. They are usually designed using standardized interfaces. Hence, at the ASIC, FPGA and IP verification level, the basic functionality of the design and the external interconnects are verified. Finally, the goal of verifying at the system level is to test the functionality of the integrated design exhaustively.

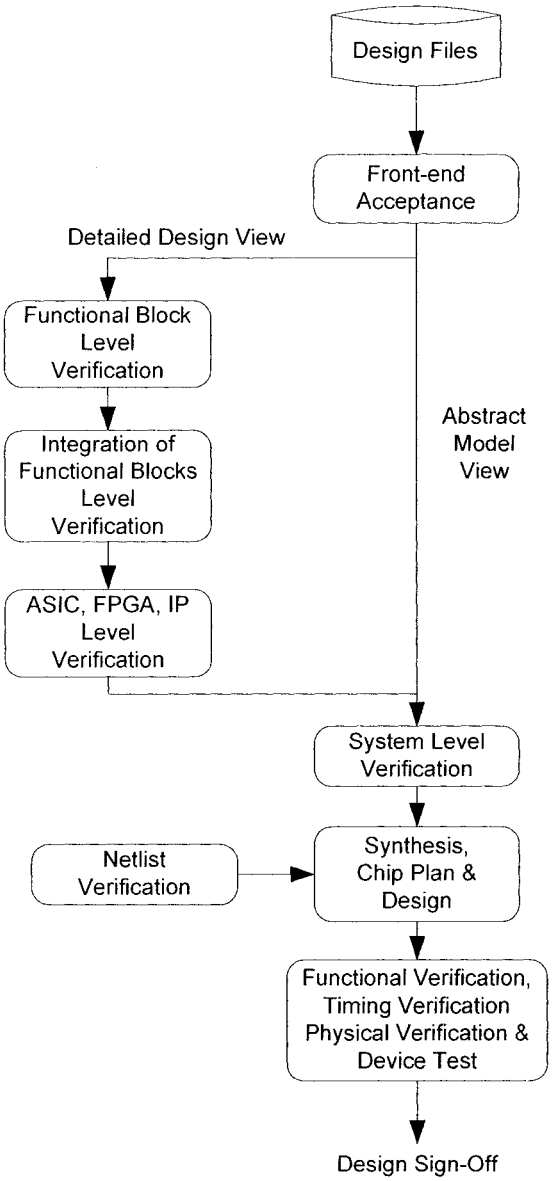


Figure 2.3 - Bottom-up Verification Approach

2.4 Survey Sample Structure

One important point to keep in mind, while analyzing the functional verification survey results presented in the next section, is that at the very best, a sample survey produces close estimates of what the respondents think or do. A sample is a set of

respondents selected from a larger population for the purpose of a survey. We sampled the microelectronics design and verification community by sending personal invitations to complete the survey to personal contacts in different companies, and by posting an invitation on the Verification Guild mailing list (Bergeron 2003a). The sample we chose is not a probabilistic sample because we did not use a systematic selection method that would give everyone in the study population a known chance of being selected. Therefore, in theory, whatever new information is gained through this research applies only to the sample itself and not to the entire verification community (Salant, Dillman 1994). This type of sampling is appropriate for exploratory research such as ours, because the purpose of the survey was to draw statistical conclusions concerning our respondents, and not a larger group of people.

2.5 Results Analysis

By the end of the survey period, 71 respondents had completed the survey. Our questionnaire comprised 45 questions. Respondents were surveyed on the verification methodology they use, their opinions on what constitutes a good verification process, and their technical background. The survey was anonymous, but many respondents have willingly given us the names of the companies they work for. Hence, we were able to identify results from at least 31 different microelectronics companies. The collection of results will be presented according to seven different topics: respondents profile, verification process, verification strategies, coverage, verification reuse, formal and assertion-based verification. Thus, strictly speaking, the following analysis applies only to the people who filled up the questionnaire. The validity of any generalization is influenced by possible bias of the sample set.

2.5.1 Respondents Profile

The majority of our respondents are designers and verification engineers. Of these respondents, 39% have completed a Bachelor's degree, 47% have completed a Masters degree, and 4% have a PhD. We received responses from multimedia,

microprocessor, semiconductors, memory and consulting companies to mention a few. However, it is remarkable that 49% our respondents work in the telecommunication industry (voice and data communication included). We see that 33% have between 0 and 5 years of experience, 21 % have between 6 and 10 years of experience, 24% have between 11 and 15 years of experience, but 13% have more than 15 years of experience.

2.5.2 The Verification Process

The verification process refers to how the verification task is addressed and positioned in the entire development flow. For example, figure 2.1 shows the development flow proposed in the current literature. In this section, we will analyze the verification flow of our respondents and we will present their opinions on the significance of the verification task.

Undoubtedly, functional verification is very important to the vast majority of our respondents. They believe that it will be the case for at least the next year. 58% of our respondents state that between 50% and 69% of the development time in their company is spent on verification. 21% of respondents report spending more than 70% of the development time on verification. 79% of the respondents have designers in their company who perform functional verification tasks and 77% have people in their company specifically dedicated to functional verification.

The verification planning begins for 54% of our respondents at the beginning of the development process. In addition, 11% begin their verification planning during system design, and 23% begin during RTL design. Surprisingly, some companies, 9% of the respondents, do not create verification plans. Also, a significant percentage (21%) does not create structured test plans; i.e. they use *ad hoc* test plans. Finally, 33% of the respondents start their testbench design at the beginning of the development process, 20% during system design and 36% during RTL design. Figure 2.4 summarizes these results.

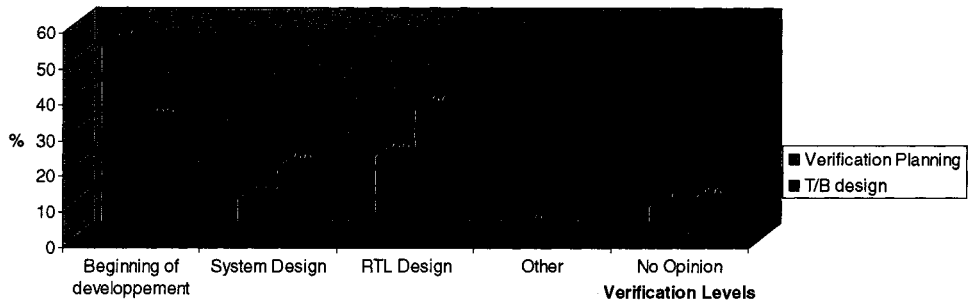


Figure 2.4 - Start of Verification Planning and Testbench Design by Pourcentage of Respondents

2.5.3 Verification Strategies

The verification strategy can be seen as what functional verification methodology is employed. For instance, fig 2.2 presents the ideal verification strategy presented in the current literature to address the verification coverage concerns and to maximize the confidence in the design under verification.

We saw that nearly half of the respondents (45%) believe that testbench design is more of a software design task and only 1% believes that testbench design is a hardware design only. Also, 43% believe that testbench design is as much hardware than software. Approximately half of the respondents (44%) use an object-oriented programming methodology to implement testbenches. When asked about a design for verification methodology, 44% of our respondents said that they do not have a design for verification methodology and a significant percentage (13%) is not familiar with this method. A design for verification methodology refers to the implementation of functional verification specific parts in a design. We found out that a vast majority of respondents (80%) implement Bus Functional Models (BFM) in their testbenches and 6% are not familiar with this concept. 72% perform pseudo random tests during simulation, 77% perform deterministic tests and a significant percentage (69%) performs both types of tests.

69% of the respondents verify at the functional design units level, 69% at integration of functional design units level, 65% at system level and 34% verify at the ASIC, FPGA or IP level. When asked about black-box, grey-box and white-box verification strategies, a majority of respondents (52%) have specified that they use a black-box verification strategy at the system verification level, 39% at the integration of functional blocks level, 31% at the functional unit level and 25% at the ASIC, FPGA or IP level. As for a grey-box verification strategy, 44% of the respondents have specified that they use this technique at the integration of functional blocks level, 34% at the functional unit level, 27% at system level and 17% at the ASIC, FPGA or IP level. A white-box verification strategy seems to be used at lower levels of abstraction since 52% of the respondents use this technique at the functional unit level and 35% at the integration of functional unit level, while only 11% have specified that they use it at the system and ASIC, FPGA or IP level. Figure 2.5 summarizes these results.

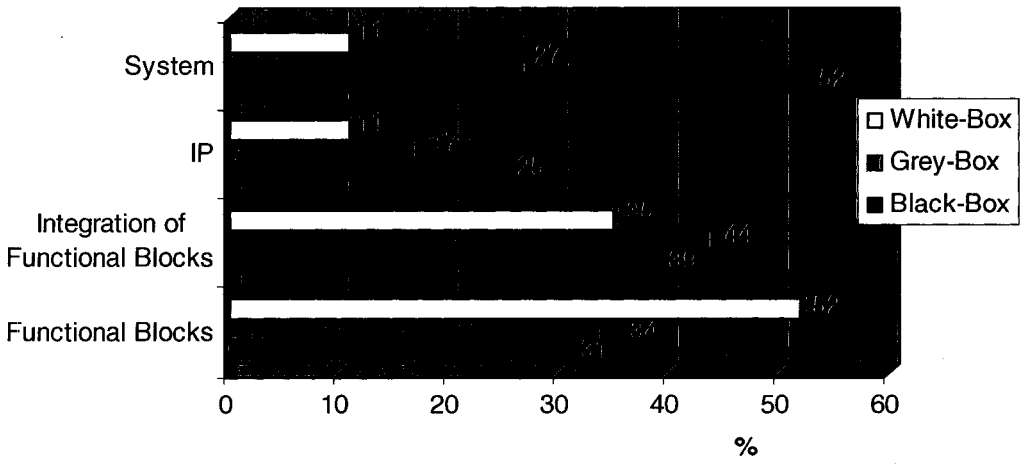


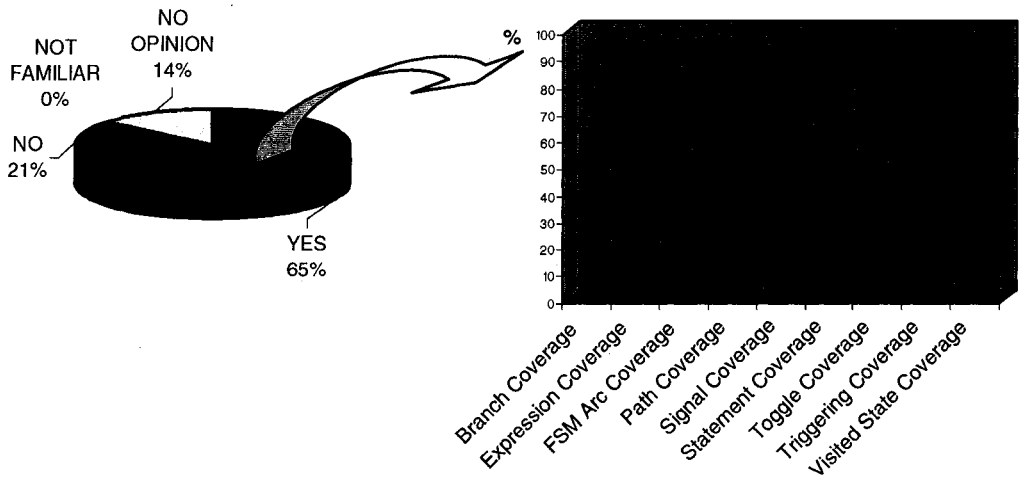
Figure 2.5 - Verification Strategies by Percentage of Respondents

HDL languages are highly used for testbench implementations at each verification level. We see that 46% of the respondents have specified that they use VHDL to implement at least some parts of their testbenches, 66% have specified that they use Verilog and 28% use both languages. However, we cannot state whether these different languages are used in the same project or in different projects. This question remains open.

In addition, 38% of all the respondents have specifically mentioned that they use Hardware Verification Languages (e or OpenVera) to implement parts of their testbenches and 3% respondents use both. Furthermore, 45% of all our respondents have mentioned that they use C/C++, 37% use TCL, and 51% use PERL to implement parts of their verification environment. Finally, it is interesting to note that 8% of all the respondents have mentioned that they use SystemC to implement system-level specifications.

2.5.4 Coverage

An important requirement to complete the verification effort of a design is to reach the complete coverage of the verification plan. Coverage quantifies what fraction of a design has been verified. As mentioned, two general types of coverage can be identified: code coverage and functional coverage (see figure 2.2). With the use of predefined metrics, we can identify which parts, and what fraction of a design code the testbench has exercised. As expected, both code coverage and functional coverage are part of the verification methodology for a majority of respondents (figure 2.6 and figure 2.7).



a) b)

Figure 2.6 - a) Code Coverage Usage and b) Type of Code Coverage by Percentage of Respondents

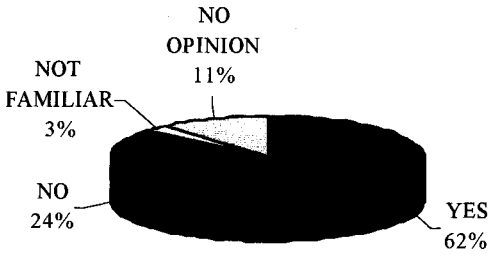


Figure 2.7 - Functional Coverage Usage by Percentage of Respondents

More precisely, 65% of our respondents use code coverage, 62% use functional coverage and 51% use both types of coverage. We have found that statement coverage, which is a metric that shows how many times each design statement was executed, is used by 93% of the people who perform code coverage. The other typical code coverage metrics (VSI Alliance 2001) are not as popular as the statement coverage metric. This is illustrated in figure 2.6. Figure 2.6 a) shows the percentage of respondents using code coverage and the percentage of respondents who do not use code coverage, while b) summarizes the type of code coverage used by the respondents who use code coverage in their

verification process. From figure 2.6 b), it is obvious that statement coverage is a largely used code coverage metric. Finally, figure 2.7 summarizes the percentage of respondents who use functional coverage. Also, the figure shows the percentage of respondents who do not use code coverage and who are not familiar with this methodology.

2.5.5 Reuse

When asked in which way reuse was part of their verification methodology, 54% of the respondents answered that they identify parts of code to reuse when needed, 30% answered that they have an internal module reuse policy, 20% use predefined frameworks from which are based all their testbench and 6% do not reuse code at all. Also, we asked the respondents' opinion on what can be reused in a testbench. We can see that 77% think that Bus Functional Models can be reused, 66% think that stimuli generation modules can be reused, 39% chose test cases, 31% chose assertions, and 27% chose coverage elements. Furthermore, 40% of the respondents believe that a design for reuse approach should be used when they think (without being absolutely sure) that a testbench module will be reused, and 26% prefer to re-design the module the next time it is required.

2.5.6 Formal Verification/Assertion-Based Verification

As mentioned, the scope of formal verification includes equivalence checking and property/model checking. Also, semi-formal verification links simulation and formal verification by using assertions to verify design properties dynamically during simulation. A majority of respondents do not use static model checking formal verification techniques. Only 17% of all our respondents have specified that they use formal model checking (see fig 2.8).

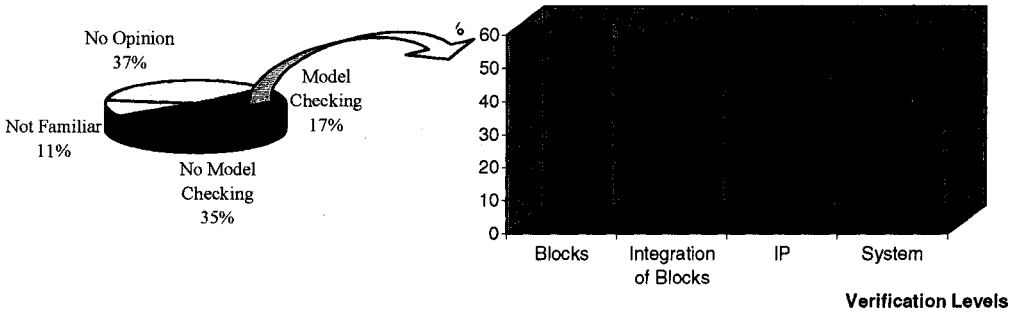


Figure 2.8 - Model Checking Practice and Associated Verification Levels by Percentage of Respondents

On the other hand, 58% of all our respondents have specified that they use formal equivalence checking. In addition, most of them use this formal method at the functional unit verification level, and also at the integration of functional unit level (see figure 2.9).

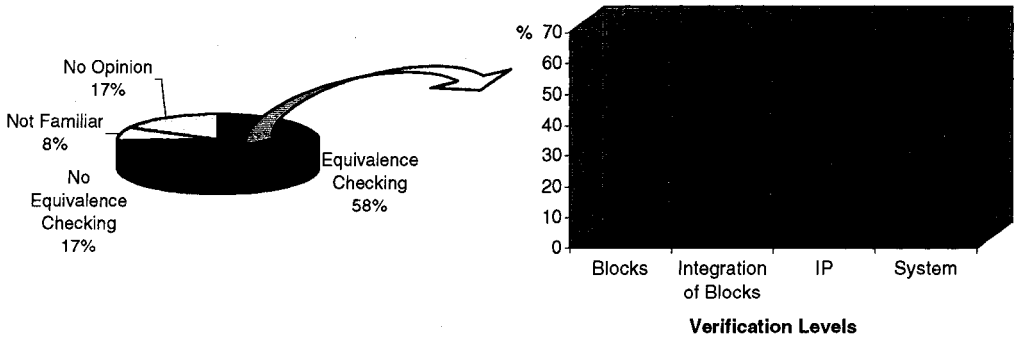


Figure 2.9 - Equivalence Checking Practice and Associated Verification Levels by Percentage of Respondents

As for assertion based-verification, 69% of our respondents use assertions to verify design properties during simulation. Out of those 69%, 90% use assertions to verify designs mostly at the functional unit verification level. Many respondents, among those who use an assertion-based simulation methodology, also use assertions to verify at the integration of functional design unit level (67%) and system verification level (55%),

but only 27% use assertions at the ASIC, FPGA or IP level (see figure 2.10). HDLs (VHDL and Verilog) are the most used languages to implement assertions. As a matter of fact, 41% of the people who use assertions implement VHDL assertions, 43% implement Verilog assertions. HVL assertions (e and OpenVera) are close with a 38% usage, while 20% use other means to implement assertions. 14% of the people who use assertions use both HDL and HVL assertions. Also, 67% of the respondents who use *e* or OpenVera have specified that they code assertions in these languages. The sum of each percentage associated to each assertion language does not equal 100%. This is explained by the fact that some people use more than one language to implement assertions. However, we cannot conclude whether these people use more than one language in the same project.

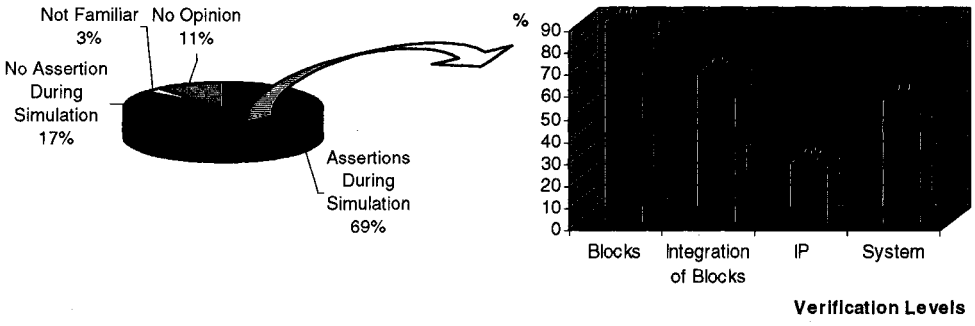


Figure 2.10 - Assertion-Based Verification Practice in Simulation and Associated Verification Levels by Percentage of Respondents

2.6 Survey Results Implications and Discussion

By analyzing the survey results, we have obtained an overall perspective of our sampled respondents' functional verification practice. Natural and intuitive verification methods have emerged, but many respondents do not yet accept some practices. We see

that for the majority of the respondents, functional verification is a very important issue and will still be a major issue for some time.

2.6.1 Verification Planning

By trying to know where people in the industry perform the verification task in their design flow, we saw that verification was generally planned at the beginning of the development process. Hence, we see that planning the verification at the beginning of the development process seems to be an intuitive way to manage a project development. It is remarkable that some companies do not implement verification plans and that a significant percentage does not create organized verification plans. Many authors, (Rashinkar, Paterson, Singh 2001, Haque, Khan, Michelson 2001, Bergeron 2003b, Bening, Foster 2001, Cohen 2001a, Cohen 2001b) to name a few, emphasize the importance of planning the verification and building a verification plan early in the development flow to determine when the verification will be completed with a required degree of confidence. This practice should be applied to every development project to ensure first-time success (Bergeron 2000b, Bergeron 2003b). One important point, as depicted in figure 2.1, is that the verification plan should always start to be written once the functional requirements of the design are specified because the functionalities of the design must be clearly defined before planning the types of test required.

2.6.2 Testbench Design

Testbench design is definitely not a hardware task only, and for the most part, hardware developers think that it has a lot of software design associated with it. A good verification engineer must also be a good behavioral modeler and must break the RTL mindset related to hardware design (Bergeron 2000b, Bergeron 2003b). The structure of a synhthesizeable RTL model is more than often dictated by the limitations of a synthesis tool, but verification engineers must use software engineering techniques to structure and encapsulate their behavioral code, in order to maximize the maintainability. Considering the coding style required by a synthesizeable subset, a testbench written in behavioral VHDL, Verilog, e or OpenVera does not face the same constraints. Data abstraction

techniques, object-oriented programming and even aspect-oriented programming (Bergeron 2003b, Hollander, Morley, Noy 2000, Regimbal, Lemire, Savaria, Bois, Aboulhamid, Baron 2002a) are methodologies that should be applied in testbench design. Nevertheless, we can see that many respondents do not yet adopt object-oriented testbench design (nearly half of our respondents use it). The use of Hardware Verification Languages (HVLs) such as OpenVERA (Symopsys 2003a) and e (Verisity 2003b) plays a good part in the establishment of object-oriented testbenches implementation methodologies. It seems that there is no standard moment to begin testbench design for our sampled respondents. The theoretical flow proposed by figure 2.1 suggests that testbenches design should begin during RTL design, but after having implemented high-level behavioral models of the design. Figure 2.11 shows the testbench coding and debug flow according to the results collected through the survey. We saw that 33% of respondents begin their testbench design at the beginning of the development process, as shown with testbench coding arrow point 1 in figure 2.11. 20% and 36% of respondents begin their testbench design respectively during the system-level modeling phase and during RTL design (see testbench coding arrows segments 2 and 3 respectively in figure 2.11).

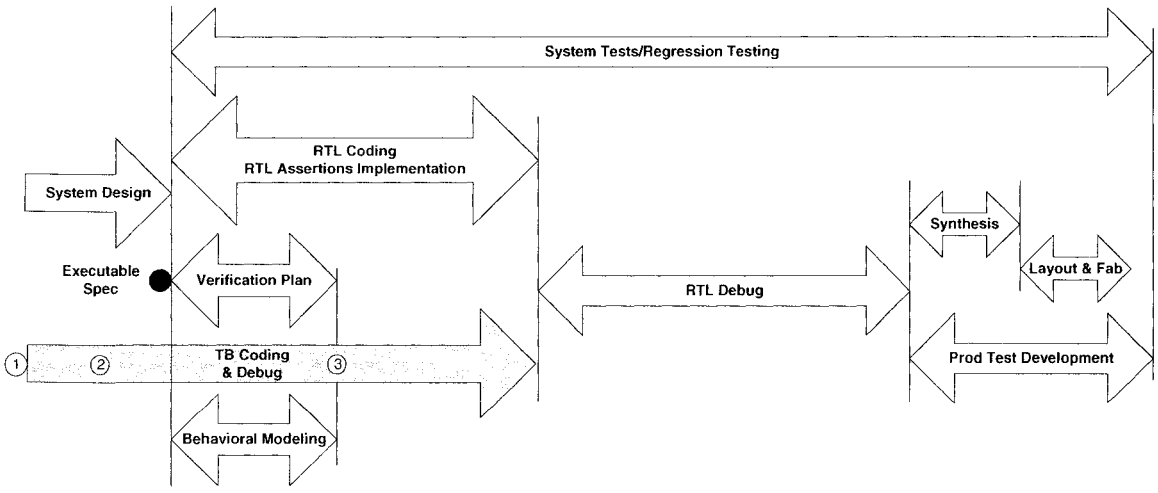


Figure 2.11 - Testbench Coding Flow According to Survey Results

We did not question the respondents about the behavioral modeling phase. Hence, we cannot state whether people use this technique. Also, we cannot state whether people of segment 3 (see figure 2.11) begin testbench coding after the behavioral modeling phase or not, and similarly, after the verification plan creation or not. A verification plan should always be written prior to the coding of new testbench modules (when no testbench module reuse is possible). The verification plan should provide vital information for testbench designers such as the granularity of the verification effort, which components to reuse, the verification strategies and, most importantly, coverage goals to define the level of confidence in the design under verification. Behavioral models are very beneficial to a development project because they enable the debugging of the testbenches before the RTL version of the design is ready. The testbench stimuli generators can then be quickly exercised to ensure that they generate interesting scenarios. For obvious reasons, the design of these behavioral models should begin when all the design functionalities have been defined.

Looking on how people assess testbench implementation, we see that HDL languages are still the preferred testbenches implementation languages, but a significant number of respondents use Hardware Verification Languages. Even if a good verification methodology does not only rely on the verification language used, HDL languages lack important features necessary to a modern verification process. HVL languages help to raise the level of abstraction of a testbench and they have the important features necessary to interact with a hardware design.

The use of Bus Functional Models is well adopted and seems to be a natural way to emulate system behavior at the device under verification boundaries during simulation. In addition, people understand that the great advantage of using Bus Functional Models is that they can reuse them through the entire verification flow, since the majority specified that Bus Functional Models have the best reuse potential over all other testbench elements. Additionally, a significant percentage does not yet adopt a design for verification methodology, and many respondents are not familiar with this methodology.

Since functional verification requires more effort than design, it is reasonable to provide additional design effort to simplify or enhance the quality of the verification task. An additional effort for designers to enhance the quality of the verification task is to code assertions directly in their RTL designs (see figure 2.1) (Bening, Foster 2001, Fitzpatrick, Foster, Marschner, Narain 2002, Foster 2002). This practice helps to provide a better white-box bottom-up verification practice, which improves simulation-based verification and provides a seamless path to formal verification.

2.6.3 Verification Reuse

The most popular and intuitive reuse method seems to be the identification of parts of code to reuse when needed, as opposed to the use of predefined implementation frameworks or the use of a company's standard reuse policy. Also, people think that they would rather design a testbench module for reuse than re-design the module the next time it is needed, even if they are not sure it will effectively be reused. Consequently, we can say that according to our respondents, a testbench design for reuse methodology has become an intuitive means to handle reuse problems.

2.6.4 Formal Verification vs Simulation

If we compare the survey results with the flow proposed in figure 2.2, we see that simulation seems to be a more natural and popular means to verify as opposed to model checking formal technique, even though a mixed simulation/formal method approach to verification is proposed in the literature. However, the use of assertions for simulation is well accepted by the surveyed community. The more intuitive use of assertions seems to be at the functional unit level. More respondents implement VHDL and/or Verilog assertions than any other assertions implementation methods but many use Hardware Verification Languages assertions. One reason that model checking is not yet widely used as a verification tool could be because earlier versions of model checker tools were difficult to run. The property specifications were not intuitive, and designers had to learn special-purpose specification languages. The Accellera organization (Accellera 2003a) is currently working on the standardization of a Property Specification Language (PSL)

based on IBM's Sugar language (IBM 2003). This standard language will ease the interoperability between the multitudes of EDA tools on the market. Another reason why model checking is not widely used is that the model checkers currently available in the industry have capacity restrictions. The size and complexity of the designs that a model checker can handle depend on the property and type of system being verified (Rashinkar, Paterson, Singh 2001). Formal model checking is effective for verifying control-intensive designs, but not for datapath-intensive designs. The designs containing datapaths typically have very large and deep state spaces, and the verification of properties on such designs can be expensive in memory and processor time. Thus, our sample set of respondents, which has a significant proportion of telecommunication engineers, may justify the inclination to simulation methodology. Furthermore, the addition of a formal verification methodology to a verification flow has been proven to be beneficial. Formal model checking does not eliminate the need for simulation but rather supplements it. Simulation is still the best tool for covering a broad range of functionalities and for eliminating the easy-to-find and some not-so-easy-to-find bugs (Bergeron 2003b). Formal model checking puts the finishing touch on the hard-to-reach corners in critical and important design sections. Also, formal model checking can be used to prove the correctness of the design for uncovered simulation cases. Functional coverage metrics collected from simulation can be used to identify conditions that remain to be verified. In contrast, formal equivalence checking seems to be the only widely used formal verification method. This formal method seems to be well adopted to verify the equivalence of two different views of a design as in the gate-level netlists verification versus its RTL functional model.

2.7 Conclusion

This paper presented a detailed analysis of the functional verification survey performed by the Groupe de Recherche en Microélectronique of École Polytechnique de Montréal. The goal of this project was to get an overall perspective of our respondents' current industrial verification practices, and to find what are the most effective and

intuitive methodologies to perform verification. The survey results were compared to methodologies proposed in the literature. We saw that a simulation-based verification is still the preferred method over a mixed formal and simulation approach. Also, almost every respondent agrees that testbenches creation is in part a software design task and not only a hardware design task, but also, many still do not use well proven software development methods to implement testbenches such as object-oriented implementations or design frameworks. The majority of our respondents is well aware that reuse plays an important part in a successful design process, but many do not use advanced reuse methods such as designing for reuse or, as mentioned earlier, design frameworks.

Finally, it would be of interest to further investigate how many designers use different languages within the same project to perform verification. It would also be interesting to know the opinion of the design/verification community on many topics such as the future of formal verification and assertion-based simulation, the use of Hardware Verification Languages and the possible increase of their testbenches performance and quality of their verification by using those HVLs.

2.8 Acknowledgements

We would like to thank all the respondents for their time and their willingness to share their functional verification methodologies with us. This research is partly supported by Micronet, the Natural Sciences and Engineering Research Council of Canada and by PMC-Sierra. The authors are also grateful to André Baron for his guidance and constructive criticisms and to Janick Bergeron for allowing us to diffuse our survey through the verification Guild.

CHAPITRE 3

LA VÉRIFICATION BASÉE SUR LES ASSERTIONS (REVUE DE LITTÉRATURE ET SYNTHÈSE DE L'ARTICLE PRÉSENTÉ AU CHAPITRE 4)

Ce chapitre présente l'article qui fera l'objet du quatrième chapitre de ce mémoire. De plus, les notions nécessaires à la compréhension de l'article sont présentées ainsi qu'une discussion générale des résultats qui ont mené à l'élaboration de celui-ci.

3.1 La vérification basée sur les assertions

Une assertion est une expression d'un comportement attendu d'un modèle qui doit être vérifié. L'objectif poursuivi par l'utilisation d'une assertion est d'assurer la cohérence entre l'intention d'un concepteur et ce qui est créé (Bening, Foster 2001). On utilise donc une assertion afin d'exprimer une propriété d'un modèle à vérifier. On définit une propriété comme étant un comportement général utilisé pour caractériser un modèle de conception. Les assertions sont utilisées lors d'une simulation ou lors d'un processus de vérification formelle. Ainsi, les assertions sont utilisées en vérification fonctionnelle afin de vérifier un comportement d'un modèle en conformité avec une spécification.

L'idée d'utiliser une assertion à des fins de vérification n'est pas nouvelle. Par exemple, l'utilisation d'assertions pour vérifier des valeurs calculées est un concept bien établi dans le domaine du génie du logiciel. Les assertions sont alors employées pour décrire des relations entre des valeurs de variables en un point précis dans un programme. La figure 3.1 montre un exemple d'une assertion typique au domaine du logiciel implantée dans un programme C++ (Deitel, Deitel 2001).

```
Employee::Employee( const char *first, const char *last )

   firstName = new char[strlen( first ) + 1];
   assert( firstName != 0 );
   strcpy ( firstName, first );
   ...
```

Figure 3.1 - Assertion typique en C++

Cet exemple montre l'utilisation d'une assertion utilisée à l'intérieur d'une fonction constructeur d'une classe nommée employé (*Employee*). Le constructeur alloue dynamiquement de l'espace mémoire pour le prénom de l'employé et utilise la fonction « *strcpy* » pour copier le prénom dans l'objet. La ligne de code contenant la macro-instruction « *assert* » constitue une assertion. La macro-instruction « *assert* » vérifie si une condition est vraie ou fausse. Si le résultat de l'évaluation de la condition est faux, alors l'assertion transmet un message d'erreur et appelle une fonction pour terminer l'exécution du programme.

Tel que présenté à la figure 3.1, l'assertion représente un outil de déverminage logiciel utile pour vérifier si une variable possède une valeur adéquate. Ainsi, dans cet exemple en C++, l'assertion détermine si l'opérateur « *new* » a rempli adéquatement la requête d'allocation dynamique de la mémoire. L'assertion examine le pointeur « *firstName* » dans le but de déterminer s'il n'égale pas 0. Si la condition testée dans l'assertion est vraie, le programme continue sans interruption. Par contre, si la condition testée dans l'assertion est fausse, un message d'erreur contenant le numéro de la ligne de code, la condition vérifiée ainsi que le nom du fichier contenant l'assertion est généré et le programme se termine.

Dans le domaine de la vérification de modèles de conception matérielle, la vérification d'assertions est reconnue comme une méthode efficace pour atteindre les objectifs de vérification. Une assertion bâtie à l'aide de l'instruction « *assert* » en VHDL représente un exemple simple d'assertion utilisée pour la vérification de modèles matériels. L'instruction « *assert* » en VHDL est présentement le seul mécanisme de

définition d'assertion inclus dans ce langage. La figure 3.2 montre une assertion en VHDL où les entrées « set » et « reset » d'un loquet S/R sont testées. L'assertion vérifie que les entrées « s » et « r » ne sont pas égales à 1 au même moment dans le but de d'éviter que le loquet ne tombe dans un état indésirable.

```
assert NOT ( (s = '1') AND (r = '1') )
   report "Set and Reset are both 1"
   severity ERROR;
```

Figure 3.2 - Exemple d'assertion en VHDL

L'instruction d'assertion vérifie la condition d'opération normale du loquet S/R et aucun message d'erreur n'est rapporté pendant la période où la condition est vraie. Par contre, si la condition vérifiée est fausse, l'instruction de message d'erreur optionnelle « *report* » est rapportée à la sortie standard. Si aucune expression d'erreur n'est spécifiée, le message d'erreur par défaut « *Assertion Violation* » sera généré. Aussi, l'instruction optionnelle de sévérité d'erreur « *severity* » spécifie un niveau de sévérité associé à l'assertion. Les quatre niveaux de sévérité sont, en ordre du moins important au plus important, « *NOTE* », « *WARNING* », « *ERROR* » et « *FAILURE* ». Par contre, l'action du simulateur associé à chaque niveau de sévérité dépend du simulateur utilisé. Le niveau de sévérité par défaut, s'il n'est pas spécifié, est « *ERROR* ».

Lorsque utilisée dans une architecture VHDL à l'extérieur d'un processus (*process*), l'assertion VHDL représente une déclaration d'un processus passif (i.e. le processus n'effectue aucune assignation de valeur à un signal) contenant l'instruction d'assertion. L'assertion VHDL peut aussi être utilisée à l'intérieur d'un bloc de code procédural dans un processus. (Cohen 1999) discute en détail de l'assertion en VHDL. Par contre, Verilog ne possède aucun équivalent sémantique à l'assertion VHDL mais la spécification de System Verilog 3.1 (Accellera 2003c) approuvée par Accellera (Accellera 2003a) possède une instruction d'assertion.

La définition détaillée d'une assertion possède généralement ces catégories syntaxiques (Foster, Flake, Fitzpatrick 2002):

- Identificateur : Une assertion peut posséder une étiquette d'identification pour faciliter l'entretien du code.
- Mécanisme d'arrêt : Une assertion peut posséder un mécanisme d'arrêt pour stopper son opération durant la vérification.
- Expression d'évaluation : Une assertion possède une expression d'évaluation qui représente ce qui est évalué pour déterminer si l'assertion réussit ou échoue.
- Événement d'échantillonnage : L'événement d'échantillonnage (voir défition d'un événement à la section 3.1.1) permet une évaluation de l'*expression d'évaluation* de l'assertion seulement au moment où celui-ci se produit.
- Niveau de sévérité : Une assertion peut posséder un niveau de sévérité qui spécifie la façon dont un outil gère l'échec de l'évaluation de celle-ci.
- Action : Une assertion peut contenir des instructions à être exécutées soit à la suite d'un succès ou soit à la suite d'un échec de son évaluation.

Il est important de mentionner que ces catégories syntaxiques ne sont généralement pas toutes implantées dans les langages supportant la définition d'assertions.

3.1.1 Evénements

L'expression d'évaluation d'une assertion peut contenir des événements. Les événements peuvent être définis comme des comportements désirables dont la réalisation est nécessaire au moment de la vérification. Les événements sont utilisés pour capturer soient des actions internes ou externes spécifiques au modèle à vérifier, des changements de valeurs de signaux ou des états. Par exemple, des événements pourraient être utilisés

pour spécifier une assertion complexe comme « Après A, B, C, on s'attend à ce que D soit vrai ». A, B, C sont tous des événements et D est le comportement à vérifier.

En général, les événements sont classés comme étant statiques ou temporels. Un événement statique représente une combinaison unique de signaux en un temps précis tandis qu'un événement temporel représente une séquence unique d'événements ou de transitions d'états sur une période de temps définie. Un exemple d'événement statique A pourrait être la réalisation d'une condition où un FIFO est plein lorsqu'il y a une requête d'écriture dans celui-ci. Parallèlement, un exemple d'événement temporel A pourrait être la réalisation d'un événement statique X suivi d'un événement statique Y un cycle d'horloge plus tard. Les langages HVL possèdent de solides structures pour définir des événements statiques et temporels complexes.

3.1.2 Expression temporelles

Plusieurs langages offrent un ensemble d'opérateurs temporels qu'un concepteur peut utiliser pour implanter des expressions temporelles. Une expression temporelle est une combinaison d'événements et d'opérateurs temporels qui décrivent un comportement particulier. De puissantes structures sont disponibles dans les langages supportant les expressions temporelles afin d'implanter des assertions complexes. Elles permettent à des propriétés comme par exemple « une requête doit toujours être suivie d'un accusé de réception au cycle d'horloge suivant » d'être vérifiées directement en utilisant la simulation ou la vérification formelle. La section 3.5.2 présente l'algèbre temporelle du langage *e*.

3.1.3 Moniteurs d'assertions

Les moniteurs sont des modules fonctionnels d'un banc d'essai qui vérifient automatiquement le comportement du modèle sous vérification. Il existe deux types de moniteurs : des moniteurs de vérification et des moniteurs d'enregistrement (Haque, Khan, Michelson 2001). Les moniteurs d'enregistrements enregistrent simplement l'activité dans le modèle sous vérification, habituellement pour des besoins de

déverminage. Les moniteurs de vérification connaissent la façon dont le modèle devrait se comporter, étant donnés des vecteurs d'entrée comme stimuli, et rapportent les erreurs de façon appropriée. Les moniteurs d'assertions sont des moniteurs de vérification implantés à partir d'assertions. L'article présenté au chapitre 4 discute plus en détail des moniteurs d'assertions.

3.1.4 Avantages des assertions utilisées lors de simulations

Les assertions aident à automatiser le processus d'exécution des scénarios d'essais lors de simulations. Si des assertions ne sont pas utilisées, il est habituellement difficile de prouver qu'un scénario de test exerce une propriété spécifique au fur et à mesure qu'un modèle évolue.

Les assertions sont aussi utiles lorsqu'elles sont impliquées dans un processus de vérification de bas en haut (*buttom-up*). Lorsque de nouveaux blocs d'un banc d'essai sont intégrés dans un environnement de vérification, les assertions peuvent vérifier les modules et les interfaces entre les blocs pour prévenir toute violation aux protocoles de communication. En plus, chaque assertion associée à un bloc de conception du modèle à vérifier demeure valide et utilisable peu importe si le modèle est vérifié de façon isolée ou faisant partie d'un système plus large. Ceci est une application adéquate du «principe de conservation de l'information utile » (Bening, Foster 2001) et ceci simplifie grandement l'intégration des propriétés intellectuelles (*IP*).

3.2 Classes d'assertions

Les assertions sont classées selon la manière dont elles sont évaluées. En plus, les assertions sont classées selon l'évaluation temporelle des propriétés qu'elles vérifient. Cette section décrit les différentes classes d'assertions.

3.2.1 Assertions procédurales et déclaratives

Une assertion peut-être procédurale ou déclarative (Fitzpatrick, Foster, Marschner, Narain 2002). D'abord, une assertion procédurale est décrite dans le contexte

de l'exécution d'un processus ou d'un ensemble d'instructions séquentielles comme par exemple les processus (*process*) VHDL ou les blocs « *always* » de Verilog. Ce type d'assertion est activé uniquement lorsque la ligne de code contenant l'assertion est exécutée de façon procédurale. Ainsi, l'assertion est implantée dans la logique d'un modèle et sera évaluée selon les chemins parcourus par le modèle à travers un ensemble d'instructions séquentielles. L'assertion présentée à la figure 3.1 est une assertion procédurale. L'assertion VHDL présentée à la figure 3.2 peut aussi être une assertion procédurale si elle est insérée dans un segment de code VHDL procédural.

Ensuite, les assertions déclaratives surveillent continuellement l'expression de l'assertion. Ce type d'assertion existe dans le contexte structurel d'un modèle. En d'autres mots, l'assertion est évaluée en même temps que tous les autres éléments structurels du modèle ou de l'environnement de vérification. Les langages HVL ainsi que les langages de vérification formelle possèdent des instructions solides pour bâtir des assertions concurrentes. Une assertion déclarative est toujours active. Ainsi, à l'opposé de l'assertion procédurale qui est seulement active lorsqu'un chemin spécifique est exécuté dans un code, l'assertion déclarative est toujours sous vérification. L'exemple d'assertion en VHDL présentée à la figure 3.2 peut être une assertion déclarative si elle est placée à l'extérieur d'un processus VHDL. De cette façon, l'assertion représente un processus concurrent contenant une instruction, soit l'assertion en question.

Il est important d'implanter les deux types d'assertions, procédurales et déclaratives, dans un modèle de conception. (Fitzpatrick, Foster, Marschner, Narain 2002) présentent un exemple d'une machine à états finis dans un bloc de code « always » de Verilog qui contient des instructions procédurales. Pour spécifier des assertions concernant les opérations effectuées dans un état particulier, le concepteur peut préférer insérer des assertions procédurales directement dans le code de cet état particulier. Par contre, pour spécifier des assertions concernant les transitions d'états observables à l'extérieur de la machine à états, le concepteur peut préférer ajouter des assertions déclaratives à l'intérieur du module dans lequel la machine à état est implantée.

3.2.2 Assertions de propriétés de sûreté (invariants) et de vivacité

Les assertions peuvent être classifiées selon le type d'évaluation temporelle requise pour vérifier les propriétés qu'elles spécifient. (Bening, Foster 2001) mentionne deux types d'évaluations temporelles de propriétés: propriétés de sûreté et propriétés de vivacité. Une propriété de sûreté est aussi appelée un invariant. De façon générale, une propriété de sûreté spécifie qu'en tout temps quelque chose de mauvais ne devrait pas se produire. De façon plus formelle, une propriété de sûreté est une propriété pour laquelle tout chemin violant la propriété possède un préfixe fini de façon à ce que chaque extension du préfixe viole la propriété. Une propriété de vivacité spécifie une fatalité qui est illimité dans le temps. De façon générale, une propriété de vivacité décrit que quelque chose de bien se produira fatalement. De façon plus formelle, une propriété de vivacité est une propriété pour laquelle tout chemin fini peut être étendu à un chemin qui satisfait la propriété.

3.3 Méthodes d'implantation d'assertions

Tel qu'introduit dans l'article du chapitre 2, dans le domaine de la vérification fonctionnelle de modèles matériels, il existe deux visions complémentaires reliées à l'implantation d'assertions selon les différentes personnes impliquées dans le processus de conception et de vérification (Fitzpatrick, Foster, Marschner, Narain 2002). Un ingénieur en vérification a besoin d'outils pour exprimer correctement des comportements d'un modèle. Par contre, un ingénieur en conception a besoin d'un mécanisme permettant d'exprimer des propriétés d'implantation bas-niveau puisqu'il doit se concentrer sur la description de modèles RTL en utilisant des langages de description matériels. L'ingénieur en conception peut implanter des assertions déclaratives ou procédurales en les intégrant au modèle de conception. Par contre, pour les ingénieurs en vérification, l'utilisation d'assertions déclaratives est naturelle afin de capturer des comportements à un haut niveau tels que les comportements aux interfaces de IP. Les ingénieurs en vérification n'implanteront probablement pas d'assertions déclaratives et procédurales dans un modèle RTL. Par exemple, le choix pour un concepteur

d'implanter une machine à états comme une machine à un bit d'état actif (*one-hot*) plutôt qu'un autre type est sans importance pour l'ingénieur en vérification tant et aussi longtemps que la machine à état démontre un comportement fonctionnellement correct.

L'approche la plus efficace consiste en une combinaison d'assertions à haut-niveau pour vérifier le système en entier et des assertions bas-niveau pour procurer une meilleure observabilité du modèle sous vérification. Les assertions codées directement dans le code RTL simplifient l'intégration des blocs à bas-niveau. De façon similaire, les assertions à haut-niveau spécifiées à l'extérieur du modèle RTL, dans l'environnement de vérification, facilitent la réutilisation des composants de vérification.

Selon (Andrews 2002, Maxfield 2002), il est possible de distinguer cinq méthodes d'implantation d'assertions :

- Assertions RTL déclaratives en utilisant des moniteurs VHDL ou Verilog
- Assertions RTL procédurales en utilisant l'instruction « assert »
- Assertions décrites en langage formel ou HVL
- Assertions incluses dans des pseudo-commentaires
- Analyse de l'historique de simulation

3.3.1 Assertions RTL déclaratives

Les assertions déclaratives dans un code RTL sont instanciées dans le modèle et interagissent de façon concurrente avec les autres composants du modèle (voir section 3.2.1). La méthode la plus commune pour implanter des assertions déclaratives dans un code RTL consiste en l'utilisation des assertions de l'*Open Verification Librairy* (OVL) (Accellera 2003b). L'OVL est une bibliothèque d'assertions VHDL et Verilog, distribuée gratuitement, pouvant être instanciées dans un modèle. Pour exprimer des propriétés temporelles au niveau RTL, le concepteur doit construire des machines à états pour capturer le comportement temporel à l'intérieur de son code. L'OVL procure une

manière cohérente de spécifier des assertions vérifiant des propriétés de sûreté et de vivacité dans un code RTL.

3.3.2 Assertions RTL procédurales

Les assertions procédurales RTL sont spécifiées en utilisant des instructions procédurales plutôt que des instances de modules appartenant à une bibliothèque.

3.3.3 Assertions décrites en langage formel ou HVL

Le chapitre 3.6 présente en détail les macanismes d'implantation d'assertions du langage *e* qui est considéré comme un langage HVL.

3.3.4 Assertions incluses dans des pseudo-commentaires

Une autre méthode d'implantation d'assertions consiste en l'intégration d'assertions à l'intérieur de pseudo-commentaires. Dans le domaine de la vérification matérielle, la méthode de pseudo-commentaires la plus utilisée consiste à intégrer des assertions à l'intérieur de commentaires dans le code RTL. Des outils de vérification peuvent ensuite lire les commentaires en utilisant un analyseur syntaxique particulier. La compagnie *0-In Design Automation* (0-in 2003) propose un outil d'assertion qui utilise des pseudo-commentaires. Cet outil produit un équivalent Verilog RTL pour chaque assertion de façon à ce qu'elle puisse être simulée à l'aide d'un simulateur standard.

3.3.5 Analyse de l'historique de simulation

L'analyse de l'historique de simulation se démarque grandement des autres méthodes mentionnées ci-haut puisqu'au lieu d'être vérifiées lors de la simulation, ces types d'assertions sont vérifiés après une simulation. Cette approche est similaire à la méthode de vérification des fichiers de résultats de simulation. La compagnie *TransEDA* (TransEDA 2003) a développé un outil qui peut lire un fichier de résultats de simulation en format VCD ou FSDB. Ensuite, l'outil lit un ensemble d'assertions spécifiées en langage Perl ou Sugar et analyse le fichier de résultats pour extraire l'information requise pour chaque assertion. Les avantages principaux de cette méthode sont que le simulateur

n'a besoin d'aucun changement, aucune extension de langage n'est nécessaire ou aucun changement dans les fichiers de conception n'est nécessaire.

3.4 SDL

SDL (*Specification and Description Language*) (ITU-T 1999, Doldi 2001) est un langage de programmation haut-niveau orienté-objet. SDL possède une grammaire et une représentation graphique standard. Ceci fournit la possibilité de compiler le code, à l'aide d'outils spécialisés, en langages de plus bas niveau tels que C/C++. Ainsi, SDL peut être traduit en une application exécutable sans avoir recours à une étape de codage manuel. SDL était à la base utilisé pour spécifier des systèmes de télécommunication mais il est maintenant utilisé pour décrire toutes formes d'applications embarquées.

Les systèmes décrits en SDL consistent en plusieurs processus concurrents communicant entre eux via des signaux. Chaque processus est décrit par une machine à états finis étendue. Un exemple d'une spécification SDL est présenté dans l'article du chapitre 4 (section 4.6).

Le choix de SDL comme langage de spécification pour ce projet est justifié par le fait qu'il constitue est un langage exécutable standard possédant une représentation graphique qui est aussi standard. Ceci qui facilite l'échange d'information entre outils et personnes impliquées dans un projet de développement.

3.5 Le langage *e*

Cette section présente les avantages reliés à l'utilisation du langage *e* pour la vérification fonctionnelle. Elle présente aussi les structures du langage *e* permettant la définition d'assertions et la définition d'expressions temporelles.

Les méthodologies de vérification ont évolué ces dernières années de simples bancs d'essai en langage HDL, permettant les tests dirigés, à des bancs d'essai complètement automatiques permettant la génération de stimuli pseudo-aléatoires. Avec

la complexité des projets actuels, l'automatisation des bancs d'essai est donc devenue primordiale pour maximiser la productivité du processus de vérification. Le langage *e* (Verisity 2002, 2003b) fournit tous les outils nécessaires pour construire un banc d'essai auto-vérifiant orienté-objet. Présentement, seul l'outil Specman Elite™ de Verisity (Verisity 2003a, 2003b) supporte le langage *e*. Ce langage peut être utilisé pour effectuer les fonctions suivantes dans un environnement de vérification :

- Génération automatique de stimuli : Des vecteurs de tests pseudo-aléatoires sont générés automatiquement à partir de contraintes de génération établies par l'usager.
- Stimulation du modèle sous vérification : Suite à leur génération, les vecteurs de tests doivent être injectés dans le modèle sous vérification. Le langage *e* possède une interface pour simulateur et il possède aussi les mécanismes nécessaires pour stimuler un modèle HDL.
- Cueillette des signaux aux sorties du modèle sous vérification suite à l'injection de stimuli : Puisque les sorties du modèle sous vérification doivent être recueillies et vérifiées, le langage *e* fournit une interface de simulation qui permet de recevoir des données d'un modèle HDL.
- Vérification des données provenant du modèle sous vérification suite à leur réception : La vérification de données peut s'établir aux sorties du modèle ou à l'intérieur de celui-ci. L'utilisation de moniteurs d'assertions permet une visibilité accrue du modèle ainsi qu'une vérification plus rigoureuse s'approchant des méthodes formelles.
- Implantation de modèles de couverture fonctionnelle : Les résultats de la couverture fonctionnelle permettent de vérifier si les objectifs du plan de vérification on été atteints. Le langage *e* possède trois types de couverture de base : la couverture d'items, la couverture de transitions ainsi que la couverture croisée (*cross coverage*).

Le choix du langage *e* comme langage de vérification est justifié par la collaboration de la compagnie PMC Sierra au projet qui utilise ce langage. En plus, ce langage est largement utilisé dans l'industrie.

3.5.1 Définition d'événements en langage *e*

Un événement (*event*) constitue un attribut d'une classe *e*. La définition d'un événement associe un nom d'événement à une expression temporelle (voir section 3.5.2). La figure 3.3 présente la syntaxe d'une définition d'événement en langage *e*.

```
event nom_d'événement [is [only] expression_temporelle]
```

Figure 3.3 - Syntaxe d'un événement *e*

L'expression temporelle utilisée lors de la definition d'un événemnt peut être la réalisation d'un autre événement (@nom_d'événement – voir section 3.5.2) ou une combinaison d'événements et d'opérateurs temporels. L'évaluation d'une expression temporelle associée à la définition d'un événement s'amorce lorsque débute la simulation. À chaque succès de l'évaluation de l'expression temporelle, l'événement sera émis. Si l'évaluation de l'expression temporelle échoue alors aucune action n'est accomplie. Des exemples de définitions d'événements *e* sont présentés à la section 4.6 ainsi qu'à l'annexe D.

3.5.2 Expressions temporelles du langage *e*

Des expressions temporelles sont utilisées dans les structures du langage *e* suivantes :

- Attributs « *event* » d'une classe (voir section 3.5.1).
- Définition d'attributs « *expect* » d'une classe (voir section 3.5.3)
- Actions « *sync* » et « *wait* » à l'intérieur de méthodes TCM (Verisity 2002)

Les points suivants résument les notions importantes se rattachant à l'évaluation d'expressions temporelles en langage *e* :

- Chaque expression temporelle possède un événement d'échantillonnage.
- L'évaluation d'une expression temporelle réussit, échoue ou demeure ouverte pour chaque réalisation de l'événement d'échantillonnage.
- La période entre chaque événement d'échantillonnage est appelée période d'échantillonnage.

Le contexte d'utilisation d'une expression temporelle détermine le moment où l'évaluation de l'expression temporelle débute. En général, une nouvelle évaluation débute à chaque réalisation de l'événement d'échantillonnage. Le tableau 3.1 présente une description des opérateurs temporels du langage *e*.

Tableau 3.1 - Opérateurs temporels du langage *e*

Nom de l'opérateur	Exemple d'utilisation	Decription
Inversion	**not** *ET*	L'évaluation de l'expression temporelle « *not* » réussit si l'évaluation de la sous-expression temporelle (*ET*) échoue lors de la période d'échatillonnage. Ainsi « *not ET* » réussie à chaque réalisation de l'événement d'échantillonnage si *ET* échoue.
Échec	**fail** *ET*	L'évaluation d'une expression « *fail* » réussit chaque fois que l'expression temporelle (*ET*) échoue. Si l'expression temporelle (*ET*) possède des interprétations multiples (par exemple *fail(ET1 or ET2)*), l'évaluation de l'expression réussit si et seulement si toutes les interprétations échouent.

Et	*ET1* **and** *ET2*	L'évaluation de l'expression temporelle réussit lorsque l'évaluation des deux expressions temporelles (*ET1* et *ET2*) débute au cours de la même période d'échantillonnage et réussit au cours de la même période d'échantillonnage.
Ou	*ET1* **or** *ET2*	L'évaluation de l'expression temporelle « *or* » réussit losrque l'une ou l'autre des deux expressions temporelles (*ET1* et *ET2*) réussit au cours de la même période d'échantillonnage.
Séquence	{*ET1* ; *ET2*}	L'opérateur de séquence « ; » évalue une série d'expressions temporelles (*ET*) à chaque réalisation d'un événement d'échantillonnage.
Contrôle de succès	**eventually** *ET*	Cet opérateur est utilisé pour spécifier que l'expression temporelle (*ET*) doit réussir en un point dans le temps non-spécifique avant que la simulation se termine.
Répétitions fixes	[*nombre*]**ET*	La répétition d'une expression temporelle (*ET*) est fréquemment utilisée pour décrire des comportements temporels cycliques ou périodiques. L'opérateur de répétitions fixes « [*nombre*] * » spécifie un nombre fixe de réalisations (*nombre*) d'une même expression temporelle.

Premier succès suite à des répétitions variables	{[exp1..exp2]*_ET1_; _ET2_}	L'opérateur temporel du premier succès suite à des répétitions variables est valide seulement lorsqu'utilisé dans une séquence ({_ET_; _ET_}). L'évaluation de l'expression de premier succès suite à des répétitions variables réussit lors du premier succès de l'expression temporelle _ET2_ entre les bornes de répétition (_exp1_ et _exp2_) spécifiées pour l'expression temporelle _ET1_.
Nombre variable de succès consécutifs	{~[exp1..exp2]*_ET_}	L'opérateur temporel du nombre variable de succès consécutifs est utilisé pour spécifier un nombre de fois variable où l'évaluation d'une expression temporelle doit réussir. _Exp1_ représente la borne inférieure du nombre de répétitions et _exp2_ représente la borne supérieure.
Dépendance	_ET1_ => _ET2_	L'opérateur de dépendance est utilisé pour spécifier que le succès de l'évaluation d'une expression temporelle (_ET2_) dépend du succès de l'évaluation d'une autre expression temporelle (_ET1_). L'expression de dépendance _ET1_ => _ET2_ est égale à ((fail _ET1_) or {_ET1_; _ET2_}).
Délai	**wait delay**(<_délai_sim_>);	L'évaluation de l'expression temporelle de délai réussit après un certain temps de simulation (_délai_sim_). Une procédure de rappel de l'outil Specman est effectuée par le simulateur lorsque le temps de simulation spécifié s'est écoulé.
Événement d'échantil-lonnage	_ET_ @ _nom_d'événement_	Cet opérateur est utilisé pour spécifier un événement d'échantillonnage pour une expression temporelle (_ET_).

Réalisation d'un événement	*@nom_d'événement*	Un événement peut être utilisé pour exprimer la forme la plus simple d'une expression temporelle *e*. L'évaluation de l'expression temporelle *@nom_d'événement* réussit à chaque réalisation de l'événement *nom_d'événement.*
Cycle	**cycle** @event_name	L'opérateur « *cycle* » est utilisé pour représenter un cycle de l'événement d'échantillonnage.
Expression booléenne	**true**(<*exp*>) *@nom_d'événement*	L'évaluation de l'expression temporelle contenant l'opérateur « *true* » réussit à chaque moment où l'expression « *exp* » est vraie.
Front d'un signal	**rise/fall/change** (<*exp*>)*@nom_ d'événement*	Les opérateurs *rise*, *fall*, *change*, détectent un changement dans la valeur échatillonnée d'un signal HDL (*exp*) lors de la réalisation d'un événement d'échantillonnage (*nom_d'événement*).

3.5.3 Assertions en langage *e*

L'attribut « *expect* » d'une classe *e* associe le nom d'une règle comportementale à une expression temporelle. Ceci permet de créer des assertions. La figure 3.4 présente la syntaxe d'une assertion en langage *e* implantée à l'aide de l'attribut de classe « *expect* ».

```
expect [identificateur is [only]] expression_temporelle
            [else dut_error(message_d'erreur)]
```

Figure 3.4 – Syntaxe d'une assertion en langage *e*

La définition d'une assertions « *expect* » prévoit un champs d'identification facultatif (*identificateur*). L'évaluation d'une expression temporelle (*expression_temporelle*) associée à l'attribut « *expect* » s'amorce lorsque débute la simulation. Ensuite, une nouvelle évaluation de l'expression temporelle débute à chaque réalisation de l'événement d'échantillonnage. À chaque échec de l'évaluation de l'expression temporelle, l'instruction « *else dut_error* » est exécutée et la simulation se termine. Un message d'erreur (*message_d'erreur*) est affiché à la sortie standard. Rien ne se produit si l'évaluation d'une assertion « *expect* » réussit. Des exemples d'assertions en langage *e* sont présentés à la section 4.6 ainsi qu'à l'annexe D

3.6 Méthodologie de recherche

Les premiers travaux reliés à ce projet de recherche ont consisté à développer une méthodologie de partitionnement pour les environnements de vérification. Le partitionnement proposé est basé sur le principe de séparation des différents aspects d'un environnement de vérification et il est implanté selon les principes de la programmation orientée-aspect (Kickales, Lamping, Menghekar, Meada, Videira Lopes, Loingtier, Irwin 1997). Les résultats de ces travaux ont été présentés dans deux conférences internationales (Regimbal, Lemire, Savaria, Bois, Aboulhamid, Baron 2002a, 2002b présentés aux annexes E et F). La section 3.8 de ce document présente un résumé des principales conclusions de ces travaux. Les résultats de ces travaux serviront aussi à l'élaboration d'une méthodologie d'implantation de moniteurs d'assertions, présentée au prochain chapitre, pour favoriser la réutilisation.

La méthodologie proposée dans l'article du chapitre 4 vise premièrement à favoriser l'établissement d'assertions fonctionnelles au niveau d'une spécification SDL exécutable. En définissant une méthodologie d'encapsulation d'assertions au niveau d'une spécification système standard, il est possible de définir plus en détail des comportements dans une représentation qui sera utilisée par plusieurs personnes impliquées dans le développement d'un projet. Quatre types d'assertions ont été

sélectionnés pour être encapsulées dans une spécification SDL (voir section 4.5). Un premier type d'assertions, assertions d'exclusion d'états, dédié à la spécification des comportements du système à haut-niveau est intégré au moment de la saisie de la spécification. Aussi, lors du processus de raffinement de la spécification, de nouvelles assertions peuvent être ajoutées pour spécifier des comportements à bas-niveau. Les assertions de bas-niveau qui ont été retenus afin d'être encapsulées dans la spécification SDL sont des assertions de temps de préparation (*set-up time*), des assertions de temps de maintien (*hold-time*) ainsi que des assertions de temps de validité (*pulse width*). Une méthode d'implantation d'assertion à l'aide de pseudo-commentaires SDL a été retenue.

Ensuite, un outil a été développé afin d'effectuer l'étape visant à assister le concepteur de bancs d'essai dans la migration automatique des assertions définies dans la spécification vers des assertions en langage *e* (voir sections 4.5 et 4.6). De façon plus précise, des patrons d'implantation d'assertions en langage *e* ont été définis et implantés dans l'outil pour créer automatiquement les moniteurs d'assertion du banc d'essai. Une méthodologie visant l'implantation des assertions pour la réutilisation a été développée et consiste à implanter des moniteurs d'assertions génériques et à adapter ces moniteurs génériques aux détails d'implantation du modèle HDL (voir section 4.4). Cette méthodologie favorisant la réutilisation est basée sur une méthodologie de partitionnement des aspects reliés à l'implantation de bancs d'essai telle que présentée à la section 3.8 ainsi qu'aux annexes E et F.

3.7 Syntaxe des assertions SDL proposées

Cette section présente la syntaxe des quatre types d'assertions SDL proposées soient des assertions d'exclusions d'états, des assertions de temps de prépapration, des assertions de maintien et des assertions de temps de validité. Les sections 4.5 et 4.6 présentent en détail le contexte d'utilisation de ces assertions ainsi que des exemples d'application. La description de la syntaxe de chaque assertion contient des mots en

caractères gras qui représentent les mots clés de l'outil de synthèse d'assertion proposé et les mots en italique représentent des champs qui devront être défnis par l'usager.

3.7.1 Assertion d'exclusion d'états

La figure 3.5 présente la syntaxe d'une assertion d'exclusion d'état.

```
check: illegal with nom_de_processus.nom_d'état;
```

Figure 3.5 - Syntaxe de l'assertion SDL d'exclusion d'états

Selon la syntaxe proposée, la définition d'une assertions SDL débute toujours par le mot-clé « *check :* ». Puisqu'une assertion d'exclusion d'état SDL est toujours liée à un état SDL précis, l'assertion doit spécifier l'état qui sera mutuellement exclusif à cet état auquel l'assertion est associée (voir section 4.5). Ainsi les champs « *nom_de_processus* » et « *nom_d'état* » spécifient respectivement le processus SDL dans lequel se trouve l'état mutuellement exclusif à l'état auquel l'assertion est associée ainsi que le nom de cet état.

3.7.2 Assertion de temps de préparation

La figure 3.6 présente la syntaxe d'une assertion de temps de préparation.

```
check: setup time to nom_de_signal = temps ns;
```

Figure 3.6 - Syntaxe de l'assertion SDL de temps de préparation

L'assertion de temps de préparation est toujours liée à un signal SDL particulier (voir section 4.5). L'assertion doit donc spécifier le temps de préparation (« *temps* ») de ce signal en nanosecondes par rapport à un signal de référence (« *nom_de_signal* »).

3.7.3 Assertion de temps de maintien

La figure 3.7 présente la syntaxe d'une assertion de temps de maintien.

```
check: hold time from nom_de_signal = temps ns;
```

Figure 3.7 - Syntaxe de l'assertion SDL de temps de maintien

L'assertion de temps de maintien est toujours liée à un signal SDL particulier (voir section 4.5). L'assertion doit donc spécifier le temps de maintien (« *temps* ») de ce signal en nanosecondes par rapport à un signal de référence (« *nom_de_signal* »).

3.7.4 Assertion de temps de validité

La figure 3.8 présente la syntaxe d'une assertion de temps de validité.

```
check: pulse width = temps ns;
```

Figure 3.8 - Syntaxe de l'assertion SDL de temps de validité

L'assertion de temps de validité est toujours liée à un signal SDL particulier (voir section 4.5). L'assertion doit donc spécifier le temps de validité (« *temps* ») de ce signal en nanosecondes.

3.8 Méthodologie de partitionnement d'environnements de vérification

Dans le domaine du génie logiciel, la réutilisation est reconnue comme étant primordiale afin d'améliorer la productivité des processus de développement (Presman 2001). Dans ce domaine, la majorité des approches pour la réutilisation sont basées sur une méthodologie de conception orientée-objet. La programmation orientée-objet (Presman 2001, Whitemore, Dearth 1999) diminue le temps de développement et augmente la qualité des logiciels en fournissant un support de programmation haut-

niveau. Les projets de développements microélectroniques actuels requièrent l'implantation de bancs d'essai complexes.

Un des grands défis dans le domaine de la vérification consiste à maximiser la réutilisation du code à l'intérieur d'un même projet ainsi que pour d'autres projets. Lorsque les ingénieurs en vérification créent de nouveaux modules de bancs d'essai, une approche de conception pour la réutilisation doit être considérée. Ainsi, concevoir des bancs d'essai pour la réutilisation requiert l'application de solides principes de conception logicielle.

Une solution possible pour augmenter le potentiel de réutilisation et d'entretien d'un logiciel consiste à utiliser une méthodologie de partitionnement des aspects. Le partitionnement des aspects d'un logiciel est la base de la programmation orientée-aspect introduite par (Kiczales, Lamping, Menghekar, Meada, Videira Lopes, Loingtier, Irwin 1997). (AOSD 2003) présente plus de détails sur le paradigme ainsi que sur le mouvement de la programmation orientée-aspect.

L'analyse orientée-objet organise un système en une claire hiérarchie d'objets. Un aspect définit un comportement qui affecte plusieurs objets d'une implantation logicielle. Lorsque certaines fonctionnalités équivalentes sont définies à l'intérieur de plusieurs objets d'un système, on dit que ces comportements entrecroisent la modularité du système. Une méthodologie orientée-aspect résout ce problème de modularité. Si toutes les fonctionnalités communes des objets sont regroupées dans des éléments de programmation que l'on nomme aspects, il est ainsi possible d'isoler ces fonctionnalités équivalentes. Le potentiel de réutilisation est grandement amélioré car cette approche force l'isolation des aspects d'un logiciel pour ainsi empêcher qu'ils ne soient dispersés dans tout le code.

La programmation orientée-aspect est un mécanisme puissant lorsque appliquée à la vérification fonctionnelle. En effet, elle permet de modifier les classes d'objets originales d'un banc d'essai sans modifier le code source original. La vérification

fonctionnelle est concernée par différents problèmes à différents moments. Chaque scénario d'essai applique différentes contraintes à l'environnement de vérification. Ainsi, chaque scénario d'essai représente une exploitation différente du même environnement de vérification. Un scénario d'essai ne devrait pas nécessairement être forcé de s'exécuter avec un autre scénario pour ainsi courir le risque d'une interférence. Les langages orientés-aspects, comme le langage *e*, offre un mécanisme bien défini pour ajouter des déclarations et insérer ou remplacer du code depuis l'extérieur d'une classe sans modifier l'implantation originale de cette classe. (Bergeron 2003b) discute de l'application de la programmation orientée-aspect au domaine de la vérification fonctionnelle.

De plus, les ingénieurs en vérification peuvent bénéficier d'une méthodologie de partitionnement des aspects lorsqu'ils ont à changer une propriété d'un modèle qui affecte plusieurs blocs fonctionnels de l'environnement de vérification. Il est souvent difficile pour les ingénieurs en vérification de retrouver chaque instance d'une propriété à modifier à l'intérieur de milliers de lignes de code. Si ces caractéristiques ne sont pas bien retrouvées et implantées, ceci peut introduire des erreurs. Partitionner un banc d'essai en différents aspects aide à regrouper chaque fonctionnalité semblable. Ainsi, cette méthodologie facilite la tâche d'entretien du banc d'essai.

Tel que déjà mentionné, (Regimbal, Lemire, Savaria, Bois, Aboulhamid, Baron 2002a, 2002b) présentent les détails d'une méthodologie de partitionnement des aspects d'un banc d'essai qui a été développée au début de ce travail de recherche. Ces deux articles sont présentés aux annexes E et F. Le tableau 3.1 résume les cinq catégories d'aspects qui ont été définis pour toute implantation de bancs d'essai.

Tableau 3.2 - Catégories d'aspects pour un banc d'essai

Catégorie d'aspect	Définition
Base	Définition du coeur de chaque classe de l'environnement de vérification.

HDL	Définition des connexions et des interactions entre les classes et le modèle sous vérification.
Glue Logic	Définition des connexions et des interactions entre les classes de l'environnement de vérification.
Coverage	Définition des modèles de couvertures de l'environnement de vérification.
Configuration	Définition de chaque scénario d'essai nécessaire à l'atteinte des objectifs du plan de vérification. Les scénarios d'essai consistent en des contraintes appliquées à l'environnement de vérification.

La catégorisation des aspects de base représente le cœur de la méthodologie de partitionnement des aspects. Lors de l'analyse orientée-objet, un certain ensemble de classes a été défini comme étant essentiel pour définir le cœur de l'environnement de vérification. Les aspects de base regroupent la définition de toutes ces classes. Chaque aspect appartenant aux autres catégories prolonge ces classes de base. Ces prolongements sont implantés en ajoutant des fonctionnalités (aspects) aux classes de base dans d'autres fichiers de l'environnement (se référer aux annexes E et F pour plus de détails ainsi que pour des exemples d'application).

3.9 Détails de soumission de l'article du chapitre 4

L'article présenté au chapitre 4 a pour titre «*Methodology for Assertion Checkers Synthesis from an Executable Specification*». Il a été soumis au magasine *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* le 15 avril 2003 et porte le numéro de manuscrit 854. L'article est présentement en processus de recommendation finale par l'éditeur associé du magazine en vue d'une future publication.

3.10 Survol de l'article

L'article constituant le prochain chapitre débute par une introduction où les objectifs du travail sont présentés. Ensuite, les fondements de la vérification basée sur les assertions sont exposés. La définition d'une assertion, les différents types d'assertions

ainsi que les méthodes d'implantation d'assertions sont présentés. En plus, un exemple de moniteur d'assertions en langage *e* est donné.

La quatrième section de l'article présente la méthodologie d'implantation de moniteurs d'assertions développée pour favoriser la réutilisation. Une méthodologie de partitionnement des aspects reliés à l'implantation de moniteurs d'assertions est présentée.

La cinquième section de l'article expose la méthodologie proposée. Un schéma de principe présente la méthodologie d'encapsulation d'assertions dans une spécification SDL ainsi que les principes de fonctionnement de l'outil de synthèse des assertions en langage *e*. Les quatre types d'assertions pouvant être encapsulées dans une spécification SDL sont aussi présentés.

La sixième section présente un exemple d'application de la méthodologie. Une partie de la spécification SDL d'un commutateur ATM est présenté. Une version détaillée de la spécification SDL du commutateur est présentée à l'annexe C. Des assertions sont ajoutées à la spécification à deux niveaux d'abstraction. L'interface graphique de l'outil de synthèse de moniteurs d'assertion est présentée. Ensuite, des exemples de code *e* généré par l'outil sont présentés pour illustrer les résultats de l'application des patrons d'implantation d'assertions. Ces patrons sont présentés en détail à l'annexe B. Aussi, les exemples de code *e* générés par l'outil sont présentés en détail à l'annexe D.

La septième section de l'article présente les expériences réalisées afin d'évaluer la méthodologie. Deux expériences ont été effectuées sur deux modèles HDL différents soit le commutateur ATM ainsi qu'une mémoire TCAM. Des assertions ont été définies pour chaque modèle et deux ingénieurs ont codé ces assertions manuellement. Ensuite, les mêmes ingénieurs ont utilisé l'outil de synthèse afin d'implanter les mêmes assertions. Les temps nécessaires aux deux types d'implantations (manuelle et automatique) sont présentés.

Pour terminer, une discussion sur les résultats ainsi qu'une conclusion sont présentées respectivement à la huitième et à la neuvième section.

3.11 Principales conclusions et résultats

L'article du prochain chapitre présente l'intérêt d'encapsuler des assertions dans une spécification exécutable. En effet, cette méthodologie permet de spécifier des comportements particuliers dans une description à un haut niveau d'abstraction. Aussi, ceci permet de fournir un moyen simple de spécifier des comportements complexes. Une spécification comportant des propriétés bien spécifiées profite aux personnes impliquées dans la vérification d'un modèle que ce soit pour la simulation ou pour la vérification formelle. Si l'outil de synthèse d'assertions est utilisé pour spécifier des assertions au niveau d'une spécification, on peut ainsi abstraire l'implantation des quatre types d'assertions sélectionnées en langage *e*.

Trois aspects importants de l'implantation de moniteurs d'assertions en langage *e* ont été identifiés soient, l'implantation des assertions constituants les moniteurs, la connexion des moniteurs d'assertions à l'intérieur du banc d'essai et la mise en correspondance des assertions aux différents signaux du modèle sous vérification. Ainsi, l'implantation d'assertions au niveau HVL proposée est partitionnée de façon à séparer d'une part, l'implantation générique des moniteurs d'assertions et d'une autre part, l'implantation de ce qui est spécifique au modèle sous vérification. Donc, comme il a été démontré, un code plus facile à entretenir, plus portable et plus adaptable est généré.

De façon expérimentale, il a été montré que l'étape d'implantation manuelle et de déverminage d'assertions temporelles complexes en langage de vérification représente une étape importante du processus d'implantation d'assertion. La méthodologie a été testée sur deux types de modèles HDL différents soient un commutateur ATM et une mémoire TCAM. Un ensemble d'assertions différentes a été implanté manuellement pour chaque modèle et ce même ensemble d'assertions a été implanté à l'aide de la méthodologie proposée. Il est montré qu'une fois les assertions bien définies dans la

spécification SDL, le temps requis par les ingénieurs qui ont participé à l'expérimentation pour implanter ces moniteurs d'assertions est d'au moins 20 fois inférieur lorsque l'outil proposé est utilisé comparativement au temps requis pour une implantation manuelle.

CHAPITRE 4

METHODOLOGY FOR ASSERTION CHECKERS SYNTHESIS FROM AN EXECUTABLE SPECIFICATION

Jean-François Lemire, École Polytechnique de Montréal, Montréal, QC
Sébastien Regimbal, École Polytechnique de Montréal, Montréal, QC
Guy Bois, École Polytechnique de Montréal, Montréal, QC
Yvon Savaria, École Polytechnique de Montréal, Montréal, QC
El Mostapha Aboulhamid, Université de Montréal, Montréal, QC
André Baron, PMC-Sierra, Montréal, QC

4.1 Abstract

According to the literature, assertion-based verification is likely to be the next breakthrough to address the verification of complex systems-on-chip (SoC's). Assertions provide a formal structure to verification, which is nowadays needed to avoid building testbenches in an *ad hoc* fashion. Nevertheless, our experience with coding assertions shows that it is difficult to code efficient assertions. Moreover, the assertions implementation process is very error-prone. In this paper, we propose a method to accelerate and optimize the e language assertions implementation process. First, we show how we can use an executable specification language such as SDL to encapsulate some assertions. We then show how our methodology assists the designer in the automatic migration of assertions defined in this executable specification towards *e* assertion checker modules. We show how the created assertion checkers are optimized for reuse. With a concrete example, we demonstrate the benefits of the proposed tool to

address the problem of designing efficient and reusable assertion checker modules in an e verification environment.

4.2 Introduction

One of today's greatest verification challenges is to provide new methodologies and more automation to reduce the time required to create functional verification environments and to perform the verification task. What was not so long ago an *ad hoc* process is nowadays one of the most interesting fields to integrate new standards and structured practices. A design implementation cannot be correct, without errors, by itself, but it can be correct in accordance with a specification, a description of what it is suppose to accomplish. Functional verification that verifies design correctness, in relation to the specification, is viewed as a major difficulty to designing increasingly complex Systems On Chips (SOC's).

Nowadays, a complex sequence of tools and techniques are required to reduce the number of design errors with respect to the specification. Several authors (Bening, Foster 2001, Bergeron 2000b, 2003b, Haque, Khan, Michelson 2001, Rashinkar, Paterson, Singh 2001) provide comprehensive discussions of the functional verification problem. A key point in a good development process is the integration of concurrent design/verification efforts. Some authors (Bergeron 2000b, 2003b, James 1999, Sternberg 2000) also highlight the benefits of a design/verification concurrent approach. We see that once the specifications are approved, a verification effort begins in parallel with the design effort. Ideally, when a RTL description is available for verification, bug-free testbenches are already available. RTL verification progresses faster, because the effort is not delayed by the debugging of testbenches.

An executable specification is an important milestone in the concurrent top down design and verification of a system, because it allows no ambiguity in its interpretation, which is not the case with a natural language specification, which is typically incomplete and ambiguous. Several standard languages are available to specify a system behavior at

a high level of abstraction. System designers can refine this high-level system model description to represent lower-level details as the system design evolves towards implementation. C/C++, SystemC and SDL are examples of standard executable specification languages. SystemC (VA Software/Open SystemC 2003) is a standard design and verification language built in C++ that spans from concept to implementation in hardware and software. As well, SDL (Specification and Description Language) (ITU-T 1999, Doldi 2001) is a formal graphical and textual system specification standard language. This specification language is increasingly used with UML (Unified Modeling Language) (Object Management Group 2001), especially to describe actions that are part of a real-time and detailed design. Hence, SDL is an executable specification language suitable to describe embedded applications.

Software reuse is known to be the most important issue for improving the productivity of software development processes. Because of the actual complexity of microelectronics designs, implementing a testbench for functional verification has become a complex software task. When a verification engineer creates new testbench components, a design for reuse approach should be considered. In other words, it is important to maximize the extent to which parts of testbench code can be reused in the same development project and in other projects. Designing testbenches for reuse requires the verification engineer to apply solid software design concepts and principles. The use of Hardware Verification Languages (HVL), such as Verisity's *e* (Verisity 2002, 2003b) and Synopsys OpenVera (Synopsys 2003, Haque, Khan, Michelson 2001) and also the use of functional verification specific C++ class library like TestBuilder (Cadence 2003), improves the verification process by addressing the reuse and information hiding requirements of complex functional verification software development. In addition, these languages enable designers to efficiently implement assertion checker modules in their testbenches. Many believe that assertion-based verification is the next breakthrough that will help to formalize the verification process and will enable engineers to continue to design and verify larger and more complex systems. That is why Accellera standards organization selected IBM's Sugar assertions language, in April 2002, as a basis for their

new standard Property Specification Language (PSL) (Accellera 2003c). Sugar promises a way to write assertions that are interoperable between formal model checkers and simulators, and that work with both VHDL and Verilog.

In this paper, we propose a methodology to accelerate and optimize the *e* language assertions implementation process. First, we show how we encapsulate assertions in a SDL specification. Since SDL offers a standard graphical notation, it enables system designers to visually design models instead of using only textual notations. Many people involved in the design, verification, management and marketing of a project can use this graphical design representation. Also, SDL provides graphical structuring features, extended finite state machines and communication through signals that are not available in languages such as C++ or Java. By defining a methodology to encapsulate assertions in a standard system-level specification language, such as SDL, we are able to further define system properties in a design representation that will be used by many development stakeholders. Thus, an SDL assertion subset enabling the definition of complex assertions in a simple style is presented. With the help of the proposed SDL assertion subset, system designers are thus able to specify complex behavior at the specification level, without coding assertions from temporal primitives specific to HVL languages. Next, we show how our methodology assists the designer in the automatic migration of assertions defined in the specification towards *e* assertion checker modules. This process is useful to verification engineers, because it eases the testbench implementation task by directly using information defined in the specification to create testbench checker modules. The proposed tool uses built-in assertion implementation patterns to create the *e* assertion checkers. Thus, bug-free testbench assertions can be quickly created.

We have selected four types of assertions to encapsulate in an SDL specification. A first set of assertions (to validate correct system behavior) is inserted during the specification capture, and during the refinement process, new assertions (for lower-levels) can be added. Also, we show how the created assertion checkers are optimized

for reuse, first by implementing generic checkers, and then by customizing these generic checkers with specific design implementation details.

The remainder of the paper is organized as follows. Section 4.3 gives some background information on assertion-based verification. Section 4.4, describes our methodology to implement assertion checkers for reuse. In section 4.5, we present a method to encapsulate assertions in a SDL specification. We also give an overview of the tool that automatically migrates assertions defined in SDL into e testbench assertion checker modules optimized for reuse. In section 4.6, we present our methodology by applying it to a concrete example, the verification of an ATM switch. In section 4.7, we present an evaluation of the methodology. In section 4.8, we discuss the proposed methodology and we suggest new avenues to investigate. Finally, section 4.9 presents a conclusion.

4.3 Assertion-Based Verification

Assertions are a means to express a design property to be dynamically verified in a simulation or formal verification. A property is a general behavioral attribute used to characterize a design. Assertion-based verification is a method to verify a group of design properties with the use of assertions. From a syntactical point of view, comprehensive assertion definitions usually possess the following features (Foster, Flake, Fitzpatrick 2002): an identifier, a reset (which is a mechanism that prevents the assertion from unduly firing during verification), a sampling event, an evaluation expression, a severity level and some actions that may be executed, either for an evaluation success as well as an evaluation failure. It is remarkable that these features are not all implemented in every language supporting assertions.

Assertions can be procedural or declarative (Fitzpatrick, Foster, Marschner, Narain 2002). A procedural assertion is described within the context of an executing process or set of sequential statements such as the VHDL process or a Verilog always block. Declarative assertions continuously monitor the assertion expression. In addition,

assertions may be classified by the type of temporal evaluation the assertion expression requires. We distinguish two general types of temporal properties verified by assertions: safety properties and liveness properties. Safety properties are properties that must evaluate to true for all sample points of time. Liveness properties specify an eventuality that is unbounded in time.

In the hardware functional verification domain, there exist two complementary views for implementing assertions depending on the different stakeholders in the design and verification flow (Fitzpatrick, Foster, Marschner, Narain 2002). A verification engineer needs an expressive means for specifying correct behavior, since his goal is to validate correct system behavior. On the other hand, the design engineer requires a convenient mechanism for expressing lower-level implementation properties, since his focus is on implementing RTL design descriptions using hardware description languages (HDLs).

We can distinguish five assertions implementation methods (Andrews 2002, Maxfield 2002) : 1) RTL Declarative Assertions using a library of VHDL or Verilog monitor modules, 2) RTL Procedural Assertions using VHDL or Verilog assert construct, 3) formal property languages, 4) pseudo-comment directives, and 5) post-processing simulation history. The most common RTL declarative assertions implementation method is through the use of the Open verification Library (OVL) (Accellera 2003b). The OVL is an open assertion monitor library of VHDL and Verilog modules that can be instantiated into a design. The VHDL and the new SystemVerilog (Accellera 2003d) assert construct is used to implement procedural assertions. The Sugar language (IBM 2003) supported by Accellera, Verisity's *e* language (Verisity 2002, 2003b) and Synopsys' OpenVera Assertions (OVA) (Synopsys 2003a) are all examples of formal property languages. Nowadays, the most common pseudo-comment directives method, in hardware verification, consists of embedding assertions into RTL code comments. Verification tools can then read the comments using a special parser. O-In Design Automation (0-In 2003) proposes an assertion tool that use pseudo-comment directives.

This tool outputs a Verilog RTL equivalent for each assertion so it can be simulated in a standard simulator.

Monitors are functional modules in a testbench that automatically check the behavior of the design under test. There are two types of monitors: checking and logging monitors. Logging monitors simply log the activity in the design usually for debugging purposes. Checking monitors know how the design should behave, given an input stimulus, and flag errors appropriately. Assertion checkers are checking monitors implemented with assertions. The assertion checking monitor-based verification methodology has emerged as a technique for unifying traditional simulation and formal verification. An example of an assertion checker as a testbench functional module is presented in Figure 4.1. Figure 4.2 presents an example of how an assertion checker is implemented using the *e* language. We see in Figure 4.2 that the expect construct defines an assertion as a member of a testbench class (unit) called *protocol_monitor*. Two testbench events (*ready_asserted and data_valid*) are also defined as members of class *protocol_monitor*. These events are tied to the ready and valid design signals (not shown in the figure) and the presented assertion is built by specifying a temporal relation between the two events. The assertion states that a valid signal must rise one clock cycle after a ready signal is asserted and the sampling event is the clock event (*@clk*). Assertion checkers can improve simulation quality by monitoring how the HDL design behaves and by reporting any violation. Also, assertion checkers reduce debug time since they can be used either as black-box or white-box verification components (see Figure 4.1), to provide an increased visibility of the design under test.

Even if assertion-based verification is a powerful method to verify complex systems-on-chip (SoC's), our experience with coding HVL assertions shows that efficient assertions are difficult to code and the process of implementing them is very error-prone. Expressing complex design behavior with the use of temporal constructs provided by HVL languages is not trivial. The time necessary to debug effective assertions can be quite significant. More specifically, it is easy to code assertions that will create false

positive errors, meaning that the assertions will not detect errors when they are supposed to. Also it is easy to code assertions that will create false negative errors, meaning that they will fire for no good reason.

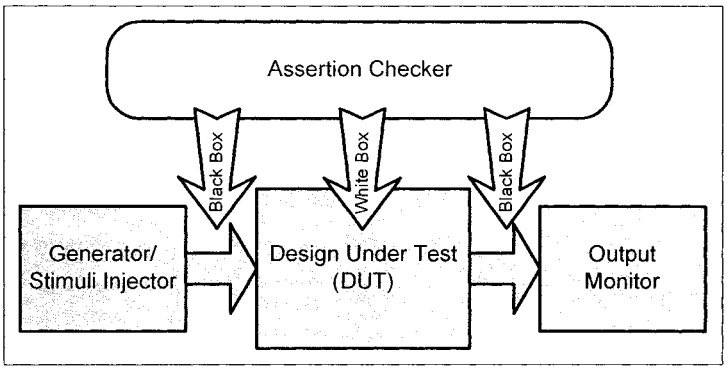


Figure 4.1 - Assertion Checker Functional View

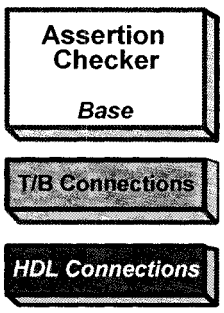
```
unit protocol_monitor {
    ...
    event ready_asserted;
    event data_valid;
    event clk;

    expect @ready_asserted => @data_valid @clk
       else dut_error("valid did not rise 1 cycle
                       after ready assertion");
    ...
};
```

Figure 4.2 - *e* Assertion Checker Example

4.4 Implementing Assertions for Reuse

An automatic testbench code generation process requires a good code implementation methodology. It is well known that reuse plays an important role in any software design methodology. One solution that we have proposed to increase the reuse potential of testbench code is to use a specific testbench code aspect partitioning methodology based on the Aspect-Oriented Programming paradigm (Regimbal, Lemire,

Savaria, Bois, Aboulhamid, Baron 2002a, 2002b). This methodology allows an efficient separation of verification concerns when constructing testbenches, which increases the reuse potential. The main concerns related to the creation of assertion checkers in a HVL testbench are: (1) implementing the assertion checker modules, (2) connecting the assertion checker modules with other testbench elements, and (3) connecting the assertions to internal design signals or variables. Therefore, as shown in Figure 4.3(a), if we isolate these three assertion implementation concerns in the testbench assertion checkers class code, we can then increase the assertion checkers reuse potential due to the improved modularity of the verification environment. We first implement a basic abstract checker class (see Figure 4.3(a)). We then extend this abstract checker by adding code to connect the checker in the testbech and to connect the checker's assertions to HDL design signals. Moreover, if these implementation concerns are placed in separate testbench code files (see Figure 4.3(b)), this partitioning structure provides an efficient support to implement typical assertion checker modifications, such as: adapting the assertion checkers to new designs (portability), extending the assertion checkers to include new assertions as the verification environment evolves (maintainability), and modifying the assertion checkers to support design changes (customizability). More implementation details will be found in section 4.6 where we present a concrete example.



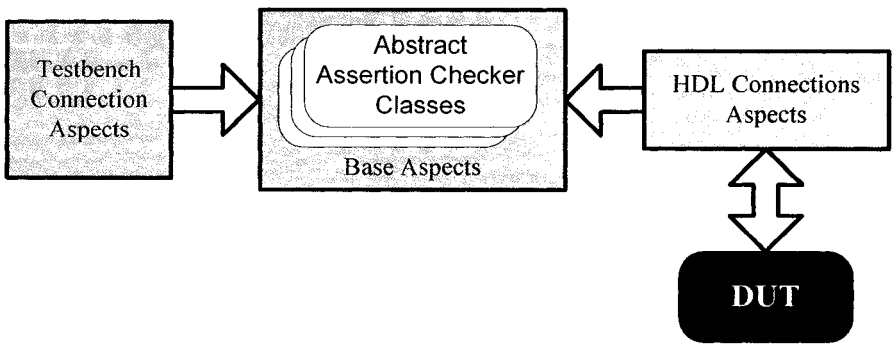


Figure 4.3(a) **Figure 4.3(b)**

Figure 4.3 - Assertion Checker Testbench Partitioning

4.5 The Proposed Methodology

In an ideal design and verification methodology, the system designer begins by creating a high-level specification to define fundamental characteristics of the design (such as communication protocols or complex arbitration schemes). This process is important to ensure specification consistency checking before implementing the design. In the communication engineering domain, the Specification and Description Language (SDL) (ITU-T 1999) has emerged as a graphical and textual standard to specify the overall communication structure of a system. A standard executable language is ideal for a high-level specification of the design functionality. This enables the system designer to create unambiguous specifications that can be verified using simulation. Also, the verification engineer needs this information to create testbenches. Moreover, the use of a standard representation technique to create the high-level specification of a design is required in our methodology. Although SDL possesses standard graphical and textual notations, it is convenient to use the graphical notation as a specification implementation means and then use an SDL model authoring tool to translate the specification to a textual notation. Since the implementation of an efficient automatic assertion generation tool requires that this tool be given standard inputs, the SDL textual version of the specification is more suitable to use as input, since it is relatively easy to extract relevant information from a text file.

Figure 4.4 shows a diagram of the methodology supported by the proposed tool. Assertions are first encapsulated in an SDL specification. The user must choose the assertions from the proposed SDL assertion set. An SDL pseudo-comment directives approach is used to define and configure assertions. Since assertions are placed into design comments, they will not interfere with the design code. A first set of assertions, to specify high-level system behavior, is inserted during the specification capture. At the specification refinement stage, new assertions can be added to specify lower-level design properties. Next, our methodology assists the designer in the automatic migration of the assertions defined in the specification towards *e* assertion checker modules.

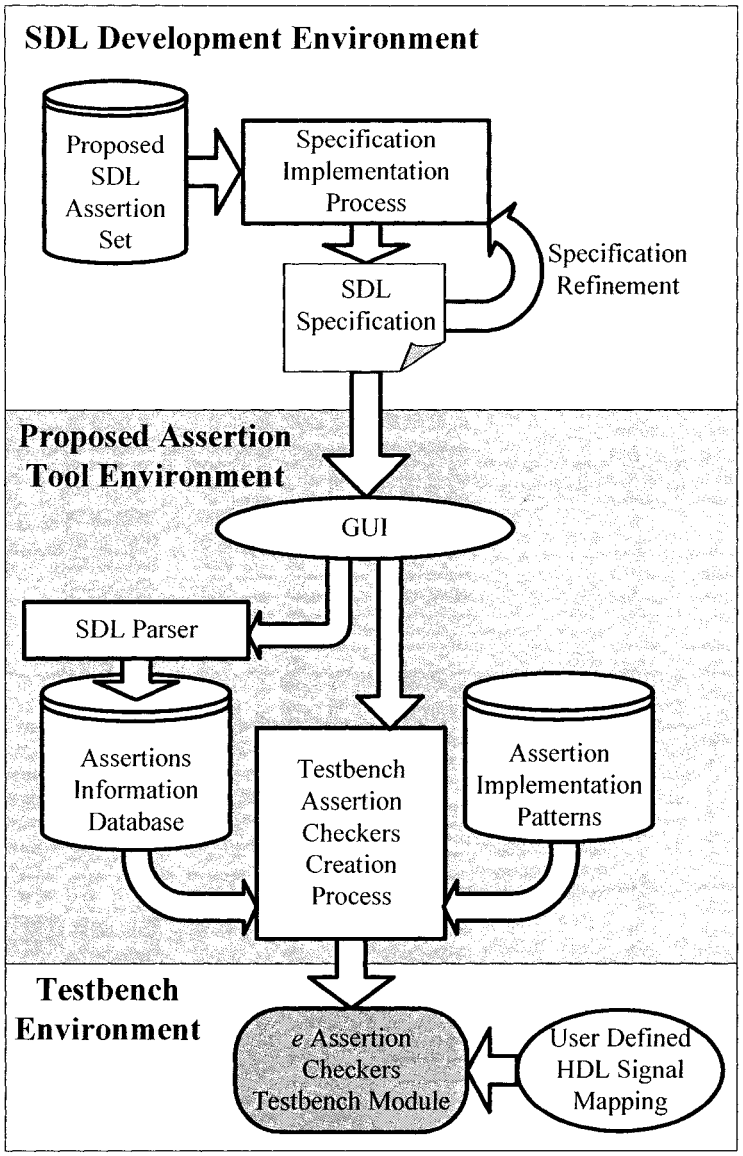


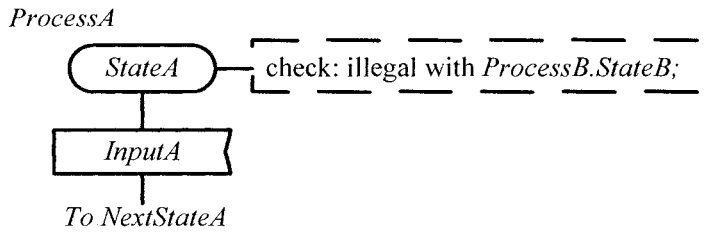
Figure 4.4 - Assertion Tool Overview

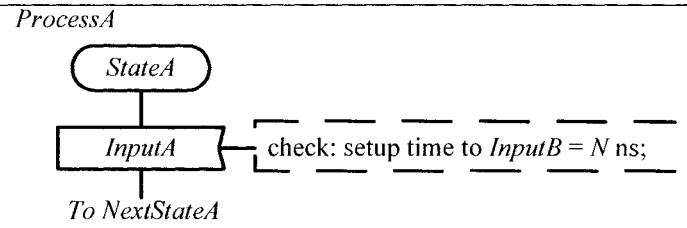
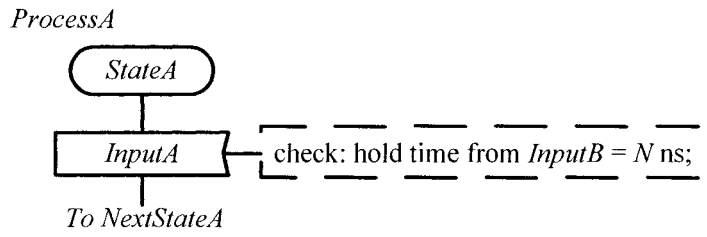
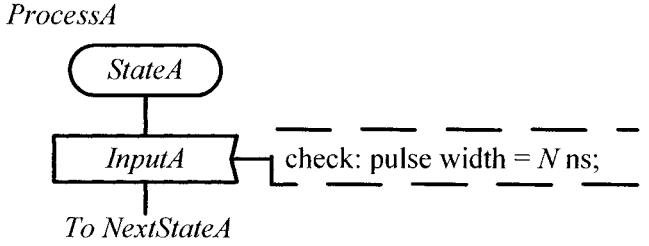
The SDL specification is first parsed to create an assertion information database. Then, the automatic assertion checker creation process uses built-in assertion implementation patterns together with the information extracted from the SDL specification to create *e* assertion checkers. As explained in section 4.4 of this paper, the created *e* assertion checkers are optimized for reuse, first by implementing generic checkers and then by customizing these generic checkers with specific design

implementation details. To finalize the checker implementation, the user must specify the HDL signals to which the created assertions are related.

We have selected four types of assertions, which are supported by the tool, to encapsulate in an SDL specification. As a first experiment, we have used this set of assertions to prove the concept of automatic assertion migration from an executable specification to HVL testbench assertions. Since our approach is flexible, we can add other types of assertions to define a comprehensive specification language assertions set. An *e* implementation pattern has been built inside the tool for each assertion. Table 4.1 shows the supported assertions and how they are specified in SDL. Each SDL assertion example shows an SDL state transition (from *StateA* to *NextStateA*) inside an SDL process (*process A*). The state transition occurs when the process receives the signal *InputA*. An assertion is attached to either a state or an input signal. Each assertion begins with the keyword "check". The SDL parser recognizes as an assertion definition the statements that follow the "check" keyword. The high-level State Exclusion assertion specifies illegal states in which two concurrent design processes cannot be at the same time. For example, the State Exclusion assertion in Table 4.1 specifies that the design cannot be at the same time in StateA from ProcessA and in StateB from ProcessB. Low-level assertions specify signals timing behavior (setup time, hold time and pulse width). For example, the setup time assertion specifies that *InputA* must be stable *N* ns before the activation of *InputB*.

Tableau 4.1 - SDL Assertion Set

Assertion Type	SDL Implementation Form
High-Level Assertion:	
State Exclusion Assertion	

Low-Level Assertions:	
Setup Time Assertion	
Hold Time Assertion	
Pulse Width Assertion	

4.6 Application Example

In this section, we present our methodology by applying it to a concrete example: the verification of an ATM switch. The specification of this design is available to the public for The Verification Guild Project (Bergeron 2003a). Specifically, the design is a Quad ATM user-to-network interface and forwarding node (SQUAT). UNI ATM cells are received on four input interfaces and are reformatted as NNI cells, then routed to the appropriate output interfaces. The rewriting and forwarding information is maintained by the host processor in an internal register file. The total number of ATM cells dropped due to output interface congestion is continuously reported in an internal status register.

Figure 4.5 shows a portion of the system SDL specification (see also annex C).

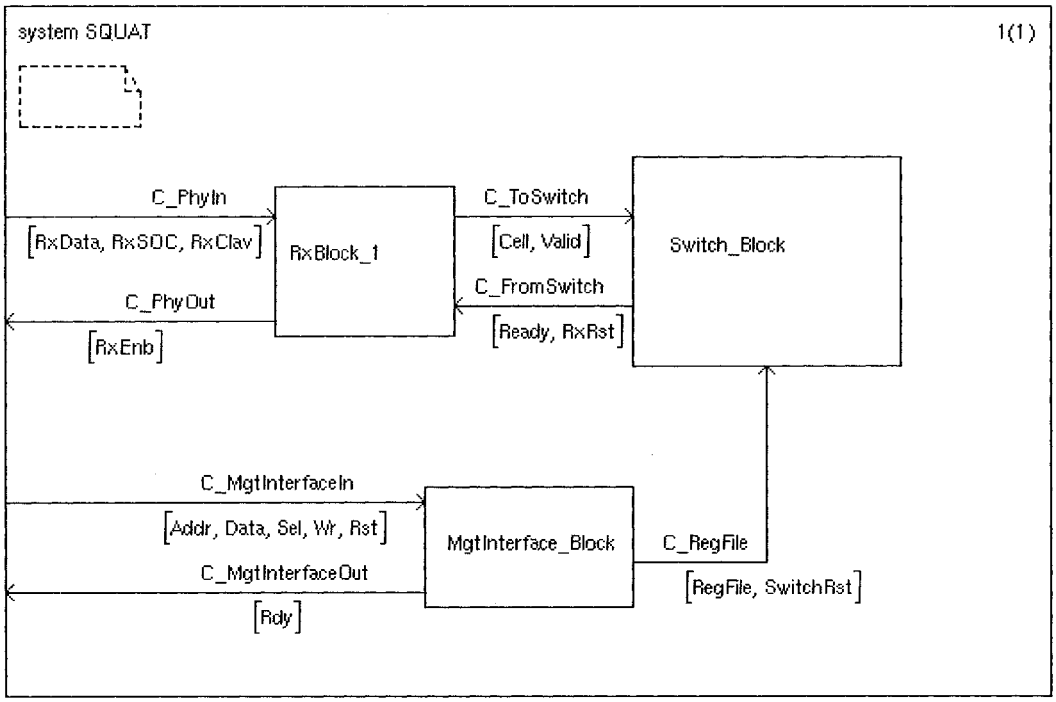


Figure 4.5 - ATM Switch SDL System Example

The example shows three different design blocks. Each block contains processes and each process contains an extended finite state machine. The state machines communicate by exchanging signals, placed in between brackets, through channels represented by arrows. The ATM Switch block (*Switch_Block*) is the core of the system. The cell rewriting and forwarding operations are performed within this functional block. The Management Interface block (*MgtInterface_Block*) receives the register file data from the host processor. Finally, the system possesses four ATM cell-receive interfaces and four ATM cell-transmit interfaces. To simplify the example, only one ATM cell-receive interface (*RXBlock_1*) is showed and we deliberately omitted the four ATM cell-transmit interfaces.

A handshake protocol has been implemented between the ATM *RxBlock_1* and the *Switch_Block* for the transfer of an ATM cell. The *Switch_Block* signals to the

interface *RxBlock_1* that it is ready to receive a new ATM cell by transmitting a ready signal carrying the value TRUE (*ready(TRUE)*). Then, the *Switch_Block* waits until the *RxBlock_1* has received a complete cell. The *RxBlock_1* transmits a *Valid(TRUE)* signal as soon as it has received a complete cell. Then, the *Switch_Block* transmits a *Ready(FALSE)* signal to the *RxBlock_1* to enable the reception of the cell contained in the *RxBlock_1*. The *RxBlock_1* responds by transmitting a *Valid(FALSE)* signal so that a cell can be transferred from the *RxBlock_1* to the *Switch_Block*. When the cell transfer is completed, the *Switch_Block* signals to the interface that it is ready to receive the next cell by transmitting a *Ready(TRUE)* signal. *RxBlock_1* is then ready to receive another new ATM cell from the physical layer and the *Switch_Block* is ready to process its cell. Figure 4.6 shows a state transition of an SDL state machine defined in *RxBlock_1*. This state machine fragment implements part of the previously discussed handshake protocol. Precisely, the *RxBlock_1* waits for a *Ready* signal from the *Switch_Block* carrying a TRUE value before enabling the reception (*RxEnb(TRUE)*) of a new ATM cell from the physical layer.

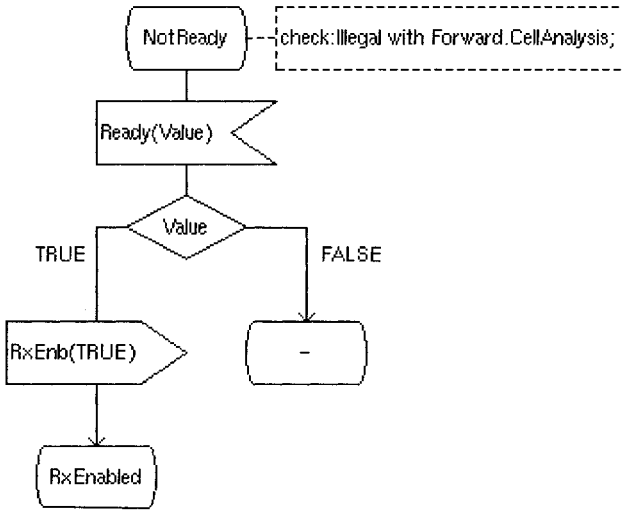


Figure 4.6 - State Assertion Example

According to the proposed methodology, a system designer would define assertions in the system SDL specification. The state exclusion assertion, which is currently supported by the tool, could be implemented for instance during the initial specification definition phase. For example, we have implemented a state exclusion assertion in Figure 4.6, which states that the *Switch_Block* cannot be analyzing a received cell if the *RxBlock_1*, where the cell came from, is still *NotReady* to receive another new cell. In other words, the *Switch_Block* cannot be in its *CellAnalysis* state (defined in the *Forward* process) if the *RxBlock_1* is still in its *NotReady* state. When the design is refined to the point of specifying low-level details, lower-level assertions can be defined in the specification. For example, we may want to specify setup-time and hold-time assertions in the *Mgt_Interface_Block* to further define the switch's register file properties. Figure 4.7 shows a subset of a SDL state machine, which specifies that once the *Mgt_Interface_Block* is selected and the address (*Addr(Address)*) is received, it falls into the *Addressed* state. We have defined a setup-time assertion and a hold-time assertion corresponding to the address signal low-level timing behavior. Precisely, the address setup-time to the memory write signal activation is 10 ns and the address hold-time from the memory write signal deactivation is 4 ns.

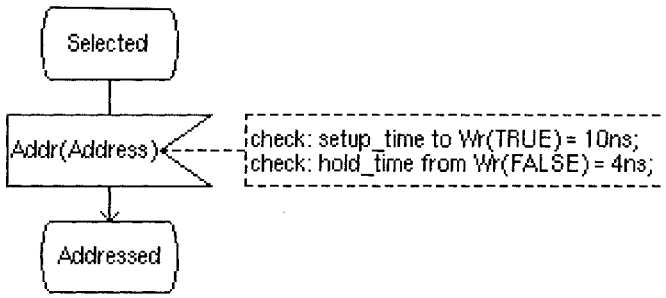


Figure 4.7 - Timing Assertions Example

During the testbench design process, the assertions defined in the SDL specification can be migrated into an *e* testbench assertion checker module by using the proposed HVL assertion implementation tool. First, the user specifies an SDL model and

runs the parser to extract any relevant information. Then, the *e* assertion checker module is created by first specifying the name of the checkers and then by running the assertion checker automatic creation process (see figure 4.8). The tool creates the assertion checker modules in accordance with the partitioning methodology presented in section 2.

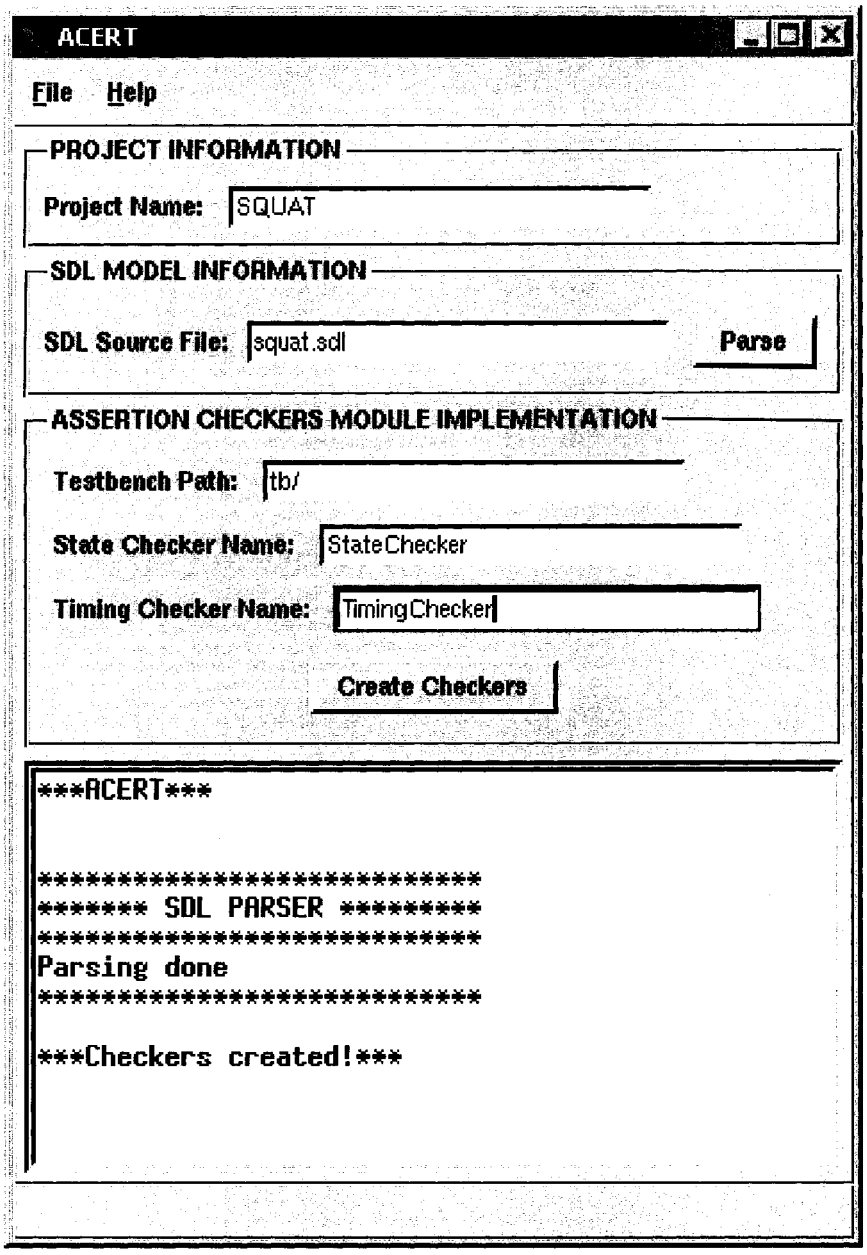


Figure 4.8 - Assertion Tool GUI

Figure 4.9 presents an *e* code example created by the tool for the state exclusion assertion of Figure 4.6.

```
1   state_machine() @sys.any is
2   {
3   all of
4   {
5      state machine RxState {
6         ...
7         NotReady => RxEnabled
8         {
9             check_machine();
10            wait @Ready_TRUE;
11        };
12         ...
13     };
14
15     state machine ForwardState {
16     ...
17     };
18  };
19  };
20  check_machine() is
21  {
22     check that (not (RxState == NotReady
23         and ForwardState == CellAnalysis));
24  };
```

Figure 4.9 - State Exclusion Assertion Base Implementation

Specifically, a portion of the base checker class (see figures 4.3(a) and 4.3(b)) named *StateChecker* is presented. The StateChecker's *state_machine()* method (lines 1 to 19) and *check_machine()* method (lines 20 to 24) are automatically generated. Since the *state_machine()* method is executed at every occurrence of the Specman sys.any event (line 1), the method is called a TCM (Time Consuming Method). In this TCM example, two *e* state machines (line 5 and line 15) are created to replicate the behavior of the two SDL state machines affected by the State Exclusion Assertion. The check_machine() method contains the State Exclusion Assertion implementation. The two state machines instruction blocks of the *state_machine()* TCM are executed in

parallel because of the « *all of* » instruction on line 3. Only an *e* version of the state transition of Figure 4.6 is presented (line 7 to line 11). Line 7 specifies that when the state machine of the *RxBlock_1* is in state *NotReady*, the next state should be *RxEnabled*. The current state of the state machine is kept in the state variable *RxState* (see line 5). Before going into the state *RxEnabled*, the instructions of lines 9 and 10 must be executed. The check_machine() method is called (line 9) to verify the state exclusion assertion. On line 10, the execution of the state machine is stopped until the occurrence of the *Ready_TRUE* abstract event, which corresponds to the SDL *Ready* signal input carrying a *Value* that is *TRUE*, in Figure 4.6. On line 22, an *e* assertion is implemented to specify that both state machines cannot be at the same time respectively in the *NotReady* state and in the *CellAnalysis* state. If this condition is not respected, the assertion will fire and the simulation will be stopped.

Figure 4.10 shows the implementation of another generic checker containing the two timing assertions defined in Figure 4.7, which are the address setup time to the write signal assertion and the address hold time to the write signal deassertion.

```
1  unit TimingChecker {
2
3      event Wr_TRUE;
4      event Wr_FALSE;
5      event Addr;
6      event Addr_setup;
7      event Addr_hold;
8
9      expect Address_Setup is
10        @Addr => {[..]*not @Wr_TRUE; @Addr_setup}
11        else dut_error("Address setup time to Wr_TRUE violation");
12
13     expect Address_Hold is
14        @Wr_FALSE => {[..]*not @Addr; @Addr_hold}
15        else dut_error("Address hold time from Wr_FALSE violation");
16 };
```

Figure 4.10 - Timing Assertions Base Implementation

Events *Write_TRUE*, *Write_FALSE* and *Address* are automatically created based on the SDL assertions and the signals attached to it. Also, events Addr_setup and Addr_hold are created because they are essential to implement the setup time assertion and the hold time assertion. As shown in Figure 4.11, the VHDL address signal (addr) must be asserted at least 10 ns before the assertion of the write signal (wr_n). The *Addr* event marks the change in the design address signal, the *Addr_setup* event marks the end of the setup time period and the *Wr_TRUE* event marks the fall of the design write signal. The *Address_Setup* assertion of Figure 4.10 (lines 9 to 11) is created to fire if the *Wr_TRUE* event occurs between the *Addr* and the *Addr_setup* events. Finally, the *Address_Hold* hold time assertion is created in the same manner (lines 13 to 15).

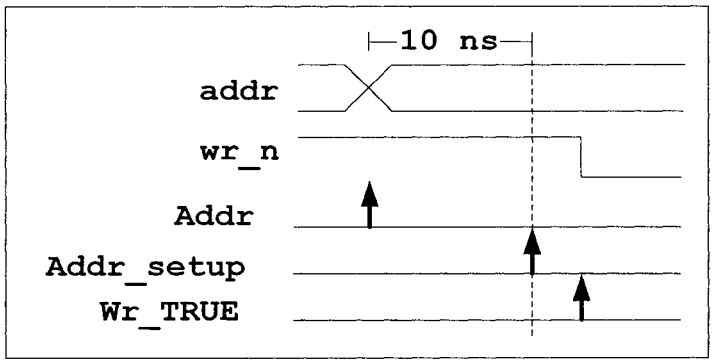


Figure 4.11 - Address Setup-Time Assertion Timing Diagram

According to figure 4.3(b), both checker connections in the testbench and to the design are implemented separately from the base class definition. Figure 4.12 shows two *StateChecker* class extensions (line 1 and line 7) dedicated to connecting the checker to the design. The first extension defines the event *Ready_TRUE* on line 4 as a rise, a fall or a change in the HDL computed signal name assigned to the *Ready_TRUE_s* string data member declared on line 2. This event was first encountered in line 10 of Figure 4.9. The user must specify if a rise, a fall, or a change is required by keeping only one of the three generated options on line 5. The second extension binds the *Ready_TRUE_s string* used to define the event *Ready_TRUE* on line 5 to a HDL signal name (line 9). The user must specify the HDL signal name to be placed between the generated quotation marks.

The use of a string-type data member to represent computed HDL signal names is recommended, because it enhances the maintainability and portability of the testbench by detaching signal mapping from core event definitions. Hence, if a modification in the design signal names occurs, the user will only have to modify the string mapping of the second extension (Van Der Schoot 2001). Finally, a *StateChecker* object must be instantiated in the testbench to perform the testbench connection requirements presented in Figure 4.3(b). The HDL signal mapping of the checker presented in Figure 4.10 is also created in the same manner.

```
1   extend StateChecker{
2      Ready_TRUE_s: string;
3      -- User must specify rise, fall or change.
4      event Ready_TRUE is only
5         Rise | fall | change('(Ready_TRUE_s)')@sim;
6   };
7   extend StateChecker{
8         --User must specify "HDL signal names"
9         keep soft Ready_TRUE_s == "";
10  };
```

Figure 4.12 - State Exclusion Assertion Checker Design Connections

4.7 Evaluation of the Methodology

We evaluated the proposed methodology by performing experiments on two real-world HDL designs: a Quad ATM user-to-network interface and forwarding node (SQUAT) presented in the application example section of this paper and a TCAM (Ternary Content Addressable Memory). First, an SDL specification has been developed for each design model and assertions were integrated in these specifications in accordance with the methodology presented in this paper. *e* language testbenches were implemented to verify the two designs. Then, assertion checkers where created and connected to the testbenches. The evaluation consisted in creating the *e* language assertion checkers manually. Afterwards, the same person has created the checkers automatically with the help of the proposed tool. The time required for these different implementations was recorded. The purpose of these experiments was to compare the

time required to manually generate the assertion checkers and the time required to implement the assertion checker with the help of the prototype tool that was implemented.

For the verification of the ATM switch design, seven assertions were defined in the SDL specification to verify different design properties. More specifically, the high-level state exclusion assertion of Figure 4.6 was implemented as well as six timing assertions including the two assertions presented in Figure 4.7. The timing assertions were specifically implemented to verify the communication protocol between the physical layer and the design's management interface. A verification engineer with four months of experience with coding *e* language testbenches has implemented the manual and automatic version of the design's assertion checkers. The results of this experiment are summarized in Figure 4.13. A total of eight hours were required to manually implement, test and debug the ATM switch assertions. Precisely three hours were required to implement and debug the state exclusion assertion and five hours were required to implement and debug the six timing assertions. Twenty minutes were required to implement the assertions (without bug) with the help of the assertions synthesis tool. The majority of these twenty minutes were spent completing the events definition and mapping the assertions to the design signals as shown in Figure 4.12.

We also applied our methodology to verify a TCAM design. The TCAM is a type of memory that accelerates any application requiring fast searches on a database, list or pattern. Moreover, in a CAM, the desired information is compared in one clock cycle against the entire list of the entries pre-stored in the memory array. The conventional CAMs are binary, where each bit of the incoming data is compared with the same position bit of the stored data. The result of the search is the address of the memory location where the first match is found. This output is usually used to address an auxiliary RAM where an associated data is stored. The ternary CAM strongly reduces the required memory size especially in Internet applications, where the longest prefix match has to be performed. In the comparison phase for each bit, three values are

searched: 0, 1, don't care. For this verification example, 16 state exclusion assertions were implemented to verify the overall functionality of the TCAM. A verification engineer with no significant experience with the *e* language performed the manual and automatic assertions implementation. A total of 11.5 hours were required to manually implement the 16 assertions whereas 30 minutes were required to implement the assertion checker by using the assertions synthesis tool. It is important to note that we cannot make direct comparison between the two verification projects, since different numbers and types of assertions were created for each design.

Figure 4.13 illustrates the significant reduction in the time necessary to implement assertion checker modules, for the proposed assertion subset, with the proposed tool/methodology.

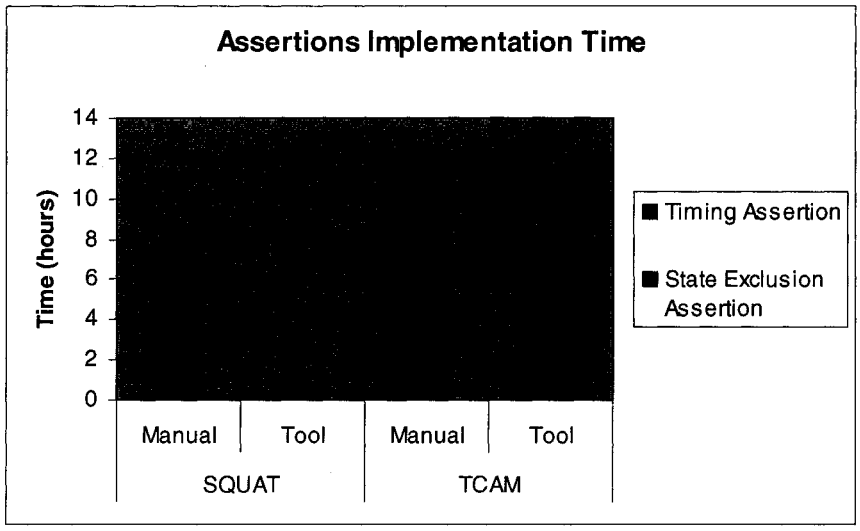


Figure 4.13 - Assertions Implementation Time

4.8 Discussion

The proposed methodology aims at improving the verification process by automating a subset of the assertion implementation process, and by proposing an assertion implementation methodology, which maximizes reuse of the testbench code. Following a comprehensive testbench partitioning methodology helps taking another step

towards increasing the kinds of verification concerns that can be captured cleanly within the source code. Implementing generic checkers creates a reusable code core that is not tied to any specific design implementation. Customizing the generic checkers separately from the core code allows a better modularity that eases the maintenance task. Furthermore, customizing the core checkers definitions adds code to the verification environment that has a low reuse potential. In other words, mapping design signals to previously defined events adds code that is really specific to the design being verified. Nevertheless, there is an automation perspective for these different implementation concerns, by developing an automatic code generation process. What is needed is a standard design representation that is used by several different development stakeholders. Specifying assertions in this design representation enables a better understanding of the design behavior and helps increase the efficiency of the verification task by clearly defining design properties to verify. A standard graphical specification entry means, such as SDL, helps to bring a specification to a level easily understandable by many people with different backgrounds. Furthermore, having a means to implement complex assertions at a graphical specification level without using complex temporal constructs specific to an HVL or a formal language does not make the specification more difficult to understand.

The use of a tool to automatically generate *e* assertion checker modules based on assertions defined in the executable specification facilitates testbench implementation. Thus, the assertion checkers presented as examples in this paper are relatively complex and, as presented in figure 4.13, require a significant amount of time to debug if coded manually as opposed to their SDL equivalent form. Therefore, if good assertion implementation patterns are defined, error-free assertion checkers can be quickly created. This efficiently reduces the assertion debugging time. Moreover, we have designed the tool to accommodate portability issues. Hence, the tool could be modified to support assertions generation in any HVL language. Only implementation patterns would then have to be modified to accommodate specific language implementation semantic and syntax.

4.9 Conclusion

In this paper, we have presented a methodology that targets the acceleration and optimization of the *e* language assertions implementation process. The proposed methodology helps create efficient and reusable assertion checker modules in HVL functional verification testbenches. On the basis of the results obtained so far, we have discussed the limitations of the proposed model and we have proposed interesting features that could be added. We believe that the idea of encapsulating high-level and low-level assertions in an executable specification and automating the migration of these assertions in a HVL or formal language is useful, beneficial and is worth further developments. The implemented tool could obviously benefit from an automatic design signal mapping process. The tool could also benefit from a more comprehensive specification language assertions set. In other words, we could add many more assertions to the current set of four that were implemented. For example, SDL state machine path assertions and SDL signal value constraints could be interesting additions to the model.

4.10 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 Elite software by Verisity and a Tau software license by Telelogic. This project also benefits from the technical guidance of PMC-Sierra. Finally, we acknowledge the contribution of Kevin Peterson, from École Polytechnique de Montréal, who has developed the TCAM testbench used in this research.

DISCUSSION GÉNÉRALE ET CONCLUSION

Les résultats du sondage sur la vérification en entreprise démontrent clairement que la vérification est au cœur des préoccupations de l'industrie. La vérification est en constante évolution et le sondage représente une vision du domaine au mois de juin et juillet 2002 pour l'échantillon choisi et non pour l'ensemble de la population en vérification. Les résultats de ce sondage ont permis d'établir une base de connaissance des pratiques en industrie et ont principalement servi d'outil de base pour introduire la vérification aux personnes nouvellement impliquées dans le domaine ainsi qu'aux étudiants dans les cours de conception microélectronique. On peut remarquer que certaines pratiques sont bien établies. Par exemple, la couverture d'instructions, la vérification formelle d'équivalence, l'utilisation d'assertions pour vérifier des blocs fonctionnels de modèles HDL ainsi que l'utilisation de modèles fonctionnels de bus pour relever le niveau d'abstraction des bancs d'essai sont toutes des méthodes très répandues. Aussi, plusieurs utilisent la couverture fonctionnelle pour obtenir une mesure des progrès du processus de vérification. Finalement, même si l'implantation de banc d'essais en langage HDL est très répandue, un nombre significatif utilise des langages HVL.

L'analyse des résultats démontre clairement qu'un fort pourcentage de répondants favorise la simulation au profit des méthodes formelles. Puisqu'un fort pourcentage de répondants provient du domaine des télécommunications, le profil des répondants peut justifier une inclinaison vers les méthodes de simulation. Il serait intéressant de savoir s'il existe des domaines où la vérification formelle est plus utilisée.

Il serait intéressant de refaire un autre sondage et de comparer l'évolution des méthodes et des approches des répondants. Il serait tout aussi intéressant de chercher à

développer plus amplement l'espace d'échantillonnage pour rejoindre une plus grande variété d'entreprises. Il serait aussi avantageux de concentrer l'échantillonnage spécifiquement sur des types d'entreprises en particulier. De cette façon, une analyse des méthodes de vérification en fonction des domaines d'application pourrait être réalisée. Il serait profitable de chercher à comprendre qu'est-ce qui motive les compagnies à opter pour leur méthodologie de vérification actuelle. En plus, si une autre version du sondage devait être faite, il est évident qu'il serait pertinent d'aborder d'autres sujets. Par exemple, il serait d'actualité de discuter plus en détails de la génération pseudo-aléatoire de vecteurs de test dans des environnements de vérification par la définition de contrainte de génération. Présentement, Accellera, l'organisation qui veille à l'application de standard dans l'industrie, se concentre à développer et à standardiser un langage d'assertions nommé PSL *(Property Specification Language)*. Cet effort de standardisation procurera l'infrastructure standard nécessaire à l'utilisation d'assertions à l'intérieur des processus de conception et de vérification. Quelques simulateurs et outils de vérification ont déjà commencé à supporter ce langage. Ainsi, il serait intéressant de savoir si l'avènement de nouveaux standard dans le domaine de la vérification basée sur les assertions changera drastiquement les méthodes de vérification des répondants.

La tendance la plus importante pour les années à venir consistera à pousser la conception et la vérification à des niveaux d'abstraction plus élevés. Les nouvelles méthodologies qui permettront ceci aideront à augmenter de façon significative la productivité. Ainsi, la méthodologie proposée dans le second article de ce mémoire visait à proposer une preuve de concept sur la possibilité d'améliorer le processus d'implantation de moniteurs d'assertions. Elle permet la définition d'assertions au niveau d'une spécification haut-niveau et assiste, à l'aide d'un outil, la migration des assertions vers des moniteurs d'assertions à l'intérieur de bancs d'essai. Il est montré qu'il est possible de créer des patrons d'implantation d'assertions complexes en langage de vérification et d'automatiser la création de ces assertions basée sur ce qui a été défini au niveau de la spécification. Élever la définition d'assertions au niveau de la spécification permet non seulement d'ajouter une simplification des comportements

spécifiés mais aussi de définir plus spécifiquement et plus complètement certains comportements ou propriétés d'un modèle. Une spécification comportant des propriétés bien spécifiées profite aux personnes impliquées dans la vérification d'un modèle que ce soit pour la simulation ou pour la vérification formelle.

Il a aussi été démontré que l'étape d'implantation manuelle et de déverminage d'assertions temporelles complexes en langage de vérification représente une étape qui demande beaucoup de temps. Il est alors important d'ajouter de l'automatisation dans le processus de création d'assertions. L'importance de l'utilisation de patrons dans le but de concevoir des systèmes complexes est reconnue dans plusieurs domaines et particulièrement dans le domaine du logiciel. Si de bons patrons d'implantation d'assertions sont définis, c'est-à-dire qu'ils décrivent des solutions aux problèmes d'implantation d'assertions qui surviennent, des moniteurs d'assertions peuvent être créés rapidement et sans erreur si les patrons servent de base à l'implantation automatique de ceux-ci. L'implantation d'assertions au niveau HVL proposée est partitionnée de façon à séparer d'une part, l'implantation générique des moniteurs d'assertions et d'autre part, l'implantation de ce qui est spécifique au modèle sous vérification. Ainsi, comme il a été démontré, un code plus réutilisable est généré.

Dans une perspective future, il serait intéressant de développer plus largement la bibliothèque de patrons et d'assertions pouvant être implantés dans une spécification SDL. Par exemple, des assertions sur les chemins empruntés par une machine à états ou des contraintes sur les valeurs portées par les signaux pourraient être des ajouts intéressants au modèle déjà proposé. En plus, l'association entre les événements constituants les assertions et les signaux d'un modèle à vérifier constitue un obstacle sérieux à une perspective d'automatisation totale de la génération d'assertions. Une façon de contourner le problème pourrait être la génération de l'association des signaux du modèle à vérifier à un niveau supérieur à celui du code HVL. Par exemple, l'association des événements aux signaux du modèle à vérifier pourrait s'effectuer au

niveau de l'interface graphique de l'outil. Ceci assisterait l'usager de façon plus efficace et pratique.

Dans le cas où un standard dans le domaine des assertions se démarquerait, cet outil de synthèse automatique d'assertions demeurerait valide si quelques modifications mineures sont apportées puisqu'il a été conçu pour être portable. En effet, seulement les patrons d'implantation devraient être modifiés pour supporter la nouvelle syntaxe et sémantique du nouveau langage. La structure principale de l'outil, c'est-à-dire les modules d'analyse syntaxique de SDL, les structures de la base de donnée d'assertions ainsi que les fondements du processus de création automatique ne nécessiterait aucune modification. Ainsi, les patrons pourraient être modifiés pour supporter le nouveau langage standard d'assertions d'Accellera, PSL (*Property Specification Language*) lorsque les outils de vérification le supporteront adéquatement.

BIBLIOGRAPHIE

[1] 0-IN DESIGN AUTOMATION, INC. 2003. 0-In Design Automation Products & Solutions. In *0-In Assertion-Based Verification (ABV) Increases Verification Productivity*. [En ligne]. http://www.0-in.com/products.html (Page consultée le 31 mars 2003).

[2] ACCELLERA ORGANIZATION, INC. 2001. Accellera Completes Initial Draft of Verilog Enhancements, Acccepts Contributions From Member Companies. *In Accellera* [En ligne]. http://www.accellera.org/press11.html (Page consultée le 31 mars 2002).

[3] ACCELLERA ORGANIZATION, INC. 2003. Accellera, About Us. *In Accellera* [En ligne]. http://www.accellera.org (Page consultée le 31 mars 2003).

[4] ACCELLERA ORGANIZATION, INC. 2003. Open Verification Library. *Verificationlib.org* [En ligne]. http://www.verificationlib.org/ (Page consultée le 02 avril 2003).

[5] ACCELLERA ORGANIZATION, INC. 2003. Property Specification Language Reference Manual Version 1.0. [En ligne]. Napa, CA: Accellera. 124p. http://www.eda.org/vfv/docs/psl_lrm-1.0.pdf (Page consultée le 3 avril 2003).

[6] ACCELLERA ORGANIZATION, INC. 2003. *SystemVerilog 3.1 Draft 2 - Accellera's Extension to Verilog*. [En ligne]. Napa, CA: Accellera. 248 p. http://www.sutherland-hdl.com/download/SystemVerilog_3.1_draft2.pdf (Page consultée le 31 mars 2003).

[7] ANDERSON, Thomas L. 2000. «Using VCS with white-box verification techniques». *10th Annual Synopsys Users Group Conference, March 2000, San Jose, CA*. [En ligne]. San Jose, CA: Synopsys, Inc. http://www.0-in.com/resources_conference_paper.html (page consultée le 31 mars 2003).

[8] ANDREWS, Jason. 2002. « Introduction to Assertions on Axis' RCC Platform Assertion Processor ». *Axis Systems White Paper*. [En ligne]. http://www.axiscorp.com/pdf/assertion_processor.pdf (page consultée le 31 mars 2003).

[9] AOSD STEERING COMMITTEE. 2003. Aspect-Oriented Software Development. *aosd.net*. [En ligne]. http://www.aosd.net/ (Page consultée le 3 avril 2003).

[10] BARTLEY, Mike. 2001. « Verification - it's all about confidence ». *Synopsys Users Group Conference, March 12-13, 2001, Munich, Germany*. [En ligne]. Munich, Allemagne: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/eurosnug/eurosnug2001/bartley3_final.pdf (page consultée le 31 mars 2003).

[11] BENING, Lionel, FOSTER, Harry. 2001. *Principles of Verifiable RTL Design*. 2e ed. Norwell, MA: Kluwer Academic Publishers. 281p.

[12] BERGERON, Janick. 2000. « The case for eVCs ». *Verisity Users Group 2000 (VUG)*. [En ligne]. San Jose, CA : Verisity Design, Inc. http://www.qualis.com (Page consultée le 31 mars 2003).

[13] BERGERON, Janick. 2000. *Writing Testbenches: Functional Verification of HDL Models*. Boston, MA: Kluwer Academic Publishers. 384p.

[14] BERGERON, Janick. 2003. Verification Guild Mailing List. *Janick Bergeron's Home Page*. [En ligne]. http://www.janick.bergeron.com/guild/default.htm (page consultée le 31 mars 2003).

[15] BERGERON, Janick. 2003. *Writing Testbenches: Functional Verification of HDL Models*. 2nd ed. Boston, MA: Kluwer Academic Publishers. 475p.

[16] CADENCE DESIGN SYSTEMS, INC. 2003. TestBuilder. *Cadence TestBuilder*. [En ligne]. http://www.testbuilder.net/ (Page consultée le 1 avril 2003).

[17] CO-DESIGN AUTOMATION, INC. 2002. *SUPERLOG Design Assertion Subset Revision 1.8*. [En ligne]. Co-Design Automation, inc. http://www.eda.org/assertion/docs/SUPERLOG_DAS_1.8_Accellera_final.pdf (Page consultée le 1 avril 2003).

[18] COHEN, Ben. 1999. *VHDL Coding Styles and Methodologies*. 2nd ed. Norwell, MA: Kluwer Academic Publishers. 453p.

[19] COHEN, Ben. 2001. *Component Design by Example... a step-by-step process using VHDL with UART as vehicle*. VhdlCohen Publishing. 284p.

[20] COHEN, Ben. 2001. « Component Verification by Example ». *11th Annual Synopsys Users Group Conference, March 2001, San Jose, CA* . [En ligne]. San Jose: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/sjsnug/cohen_final.pdf (Page consultée le 1 avril 2003).

[21] DEITEL, H. M., DEITEL, P. J. 2001. *C++ how to program*. 3rd ed. New Jersey: Prentice Hall. 1168p.

[22] DOLDI, Laurent. 2001. *SDL Illustrated - Visually design executable models*. Toulouse, France: Éd. Laurent Doldi. 270p.

[23] ECLIPSE.ORG CONSORTIUM. 2003. AspectJ Project. *Eclipse.org*. [En ligne]. http://www.eclipse.org/aspectj (Page consultée le 3 avril 2003).

[24] FENTON, Norman E. 1995. *Software metrics, a rigorous approach*. London: International Thomson Computer Press ed. 656p.

[25] FITZPATRICK, Tom, FOSTER, Harry, MARSCHNER Erich, NARAIN, Prakash. 2002. « Introduction to Accellera's assertion efforts ». *EEDesign*. [En ligne]. http://www.eedesign.com/features/exclusive/OEG20020602S0001 (Page consultée le 2 avril 2003).

[26] FOSTER, Harry, FLAKE, Peter, FITZPATRICK, Tom. 2002. « Adding SUPERLOG Design Assertion Extensions to System Verilog ». *The 11th Annual International HDL Conference & Exhibition Proceedings*. San Jose, CA. 117-124.

[27] FOSTER, Harry. 2002. Improving Verification Through Property Specification. *Design and Reuse*. [En ligne]. http://www.us.design-reuse.com/news/news2841.html (Page consultée le 2 avril 2003).

[28] FORTE DESIGN SYSTEMS. 2003. The Convergence of Design and Verification. *Forte Design Systems- Strength in Verification, Strength in Design*. [En ligne]. http://www.forteds.com/ (Page consultée le 2 avril 2003).

[29] GAMMA, Erich, HELM, Richard, JOHNSON, Ralphn, VLISSIDES, John. 2000. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley. 395p.

[30] GOERING, Richard. 2002. « Vendors Join Push for Assertion Standards ». *EETimes*. [En ligne]. http://www.eetimes.com/story/oeg20020311s0026 (Page consultée le 2 avril 2002).

[31] GOERING, Richard. 2002. « Asserting trio of standards ». *EEdesign*. [En ligne]. http://www.eedesign.com/story/OEG20020318S0014 (Page consultée le 2 avril 2003).

[32] HAQUE, Faisal I., KHAN, Khizar A., MICHELSON, Jonathan. 2001. *The Art of Verification with Vera*. Fremont, California: Verification Central. 452p.

[33] HOLLANDER, Yoav, MORLEY, Matthew, NOY, Amos. 2000. « The e Language: a Fresh Separation of Concerns ». *Proceedings of Technology of Object-Oriented Languages and Systems Conference*. Zurich, Switzerland: IEEE Computer Society. 41.

[34] HOWDEN, William E. 1987. *Functional program testing and analysis*. McGraw-Hill. 171p.

[35] IBM. 2003. Sugar. *Sugar: Homepage*. [En ligne]. http://www.haifa.il.ibm.com/projects/verification/sugar/index.html (Page consultée le 2 avril 2003).

[36] INFOPOLL, INC. 2001. How to Conduct a Successful Web Survey. *Infopoll.com*. [En ligne]. http://www.infopoll.com/support/resources/howto/conduct.a.web.survey.htm (Page consultée le 2 avril 2003).

[37] INFOPOLL, INC. 2001. How to Write a Good Survey. *Infopoll.com*. [En ligne]. http://www.infopoll.com/support/resources/howto/write.a.good.survey.htm (Page consultée le 2 avril 2003).

[38] INFOPOLL, INC. 2001. Three Commonly-Made Mistakes When Selecting Question Types. *Infopoll.com*. [En ligne]. http://www.infopoll.com/support/resources/tips/q.mistakes.htm (Page consultée le 2 avril 2003).

[39] INFOPOLL, INC. 2001. Top Tips to Increase Response Rate. *Infopoll.com*. [En ligne]. http://www.infopoll.com/support/resources/tips/howto.increase.response.rate.htm (Page consultée le 2 avril 2003).

[40] ITU-T. 1999. *Specification and DescriptionLlanguage (SDL)*. International Telecom Union (ITU). 246p. Telecommunication Standardization Sector of ITU Series Z: Languages and General Software Aspects for Telecommunication Systems, Geneva, Switzerland, Recommandation Z-100.

[41] JAMES, Peet. 1999. « A recipe for multi-million gate ASIC verification ». *Synopsys Users Group Conference*. [En ligne]. Boston, MA: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/snug/bostonsnug99_papers/james_final.pdf (Page consultée le 2 avril 2003).

[42] JAMES, Pete, MACIONSKI, Chris. 2001. « Shotgun e An Eight-Step Approach to Experience Random Verification ». *Verisity Users Group 2001 (VUG)*. [En ligne]. http://www.qualis.com/ (Page consultée le 2 avril 2003).

[43] KARTALOPOULOS, Stamatios V. 1999. *Understanding SONET/SDH and ATM*. Piscataway, NJ: IEEE Press Understanding Science & Technology Series. 257p.

[44] KICZALES, Gregor, LAMPING, John, MENGHEKAR, Anurag, MEADA, Chris, VIDEIRA LOPES, Cristina, LOINGTIER, Jean-Marc, IRWIN, John. 1997. « Aspect-Oriented Programming ». *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. [En ligne]. Finland. Springer-Verlag. http://citeseer.nj.nec.com/cache/papers/cs/16616/http:zSzzSzwww.parc.xerox.comzSzcslzSzgroupszSzsdazSzpublicationszSzpaperszSzKiczales-ECOOP97zSzfor-web.pdf/kiczales97aspectoriented.pdf (Page consultée le 2 avril 2003).

[45] LAHTI, Gregg D., WILSON, Tim L. 1999. « Designing procedural-based behavioral bus functional models for high performance verification ». *Synopsys Users Group Conference (SNUG)*. [En ligne]. San Jose, CA: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/snug/snug99_papers/Lahti_Final.pdf (Page consultée le 2 avril 2003).

[46] LEMIRE, Jean-François, REGIMBAL, Sébastien, BOIS, Guy, SAVARIA Yvon, ABOULHAMID, El Mostapha, BARON, André. 2003. « Implementing *e* Assertion Checkers from an SDL Executable Specification ». *12th International Design and Verification Conference and Exhibition Proceedings (DVCon 2003)*. San Jose, CA: Omnipress. Omnipro-CD.

[47] LEMIRE, Jean-François. 2002. Functional Verification Survey. *Functional Verification Survey Page*. [En ligne]. http://www.grm.polymtl.ca/~lemire/survey (page consultée le 2 avril 2003).

[48] MAMMERI, Zoubir. 2000. *SDL - Modélisation de protocoles et systèmes réactifs*. Paris: Hermes Science Publications. 624p.

[49] MAXFIELD, Clive. 2002. « Walking the assertion maze ». *EEDesign*. [En ligne]. http://www.eedesign.com/story/OEG20020815S0035 (Page consultée le 2 avril 2003).

[50] MILLER, Sandra Kay. 2001. « Aspect-oriented programming takes aim at software complexity ». *IEEE Computer Magazine*. 34:4. 18-21.

[51] MURPHY, Gail C., WALKER, Robert J., BANIASSAD, Elisa L. A. 1999. « Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming ». *IEEE Transsactions on Software Engineering*. 25:4. 438-455.

[52] OBJECT MANAGEMENT GROUP, INC. 2001. *OMG Unified Modeling Language Specification Version 1.4.* [En ligne]. Cupertino, CA: Rational Software Corporation. 582p. UML V1.4 beta R1.

[53] PIETROSKE, Dan. 2001. « VHDL verification partitioning with Verilog-like flexibility. Synopsys ». *Synopsys Users Group Conference, Europe 2001 (SNUG).* [En ligne]. Munich, Germany: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/eurosnug/eurosnug2001/pietroske_final.pdf (Page consultée le 2 avril 2003).

[54] PRESMAN, R. S. 2001. *Software Engineering: A Practitioner's Approach.* New-York, NY: McGraw-Hill. 860p.

[55] RASHINKAR, Prakash, PATERSON, Peter, SINGH, Leena. 2001. *System-on-a-Chip Verification Methodology and Techniques.* Norwell, MA: Kluwer Academic Publishers. 372p.

[56] REGIMBAL, Sébastien, LEMIRE, Jean-François, SAVARIA Yvon, BOIS, Guy, ABOULHAMID, El Mostapha, BARON, André. 2002. « Applying Aspect-Oriented Programming to Hardware Verification with e ». *Proceedings of HDL Conference 2002.* San Jose, CA: Accellera Organization, Inc. 68-75.

[57] REGIMBAL, Sébastien, LEMIRE, Jean-François, SAVARIA Yvon, BOIS, Guy, ABOULHAMID, El Mostapha, BARON, André. 2002. « Aspect partitioning for Hardware Verification Reuse ». *Proceedings of the International Workshop on System on Chip for Real-time Applications.* Banff, Canada: U. of Calgary. 49-58.

[58] SALANT, Priscilla, DILLMAN, Don A. 1994. *How to conduct your own survey.* Hoboken, NJ: John Wiley & Sons, Inc. 256p.

[59] STERNBERG, Lewis. 2000. « Getting It Right: AMS Design and Verification Strategies ». *Analog & Mixed-Signal Application Conference*. [En ligne]. San Jose, CA. http://www.qualis.com/cgi-bin/qualis/libObject.pl?object=mb017 (Page consultée le 2 avril 2003).

[60] SYNOPSYS, INC. 2003. OpenVera. *OpenVera Website.* [En ligne]. http://www.open-vera.com (Page consultée le 2 avril 2003).

[61] SYNOPSYS, INC. 2003. Synopsys Product and Solutions - VERA Testbench Automation. *Synopsys*. [En ligne]. http://www.synopsys.com/products/vera/vera.html (Page consultée le 2 avril 2003).

[62] TRANSEDA. 2003. VN-Property DX. TransEDA. [En ligne]. http://www.transeda.com/ (Page consultée le 3 avril 2003).

[63] VA SOFTWARE CORPORATION AND OPEN SYSTEMC INITIATIVE. 2003. Welcome to the SystemC Community. *SystemC*. [En ligne]. http://www.systemc.org (Page consultée le 2 avril 2003).

[64] VAN DER SCHOOT, Hans. 2001. « Signal Mapping in e ». *Club Verification - Verisity User's Group*. Ottawa, Canada: Verisity.

[65] VERISITY DESIGN, INC. 2002. *e Language Reference*. Verisity Design, Inc. Version 4.0.2. 1026p.

[66] VERISITY DESIGN, INC. 2003. Spec-Based Verification - A New Methodology for Functional Verification of Systems/ASICs. *Verisity White Papers*. [En ligne]. http://www.verisity.com/resources/whitepaper/specbased.html (Page consultée le 2 avril 2003).

[67] VERISITY DESIGN, INC. 2003. Welcome to Verisity. *Verisity*. [En ligne]. http://www.verisity.com (Page consultée le 2 avril 2003).

[68] VERPLEX SYSTEMS, INC. 2003. Verplex. *Formal verification and equivalence checking from Verplex Systems, Inc.* [En ligne]. http://www.verplex.com (Page consultée le 2 avril 2003).

[69] VIEGA, John, VOAS, Jeffrey. 2000. « Can Aspect-Oriented Programming Lead to More Reliable Software ». *IEEE Software Magazine*. [En ligne]. 17: 6. 19-21. http://www.computer.org/software/so2000/s6toc.htm (Page consultée le 2 avril 2003).

[70] VSI ALLIANCE, INC. 2001. *Taxonomy of Functional Verification for Virtual Component Development and Integration.* [En ligne]. Los Gatos, CA: VSI Alliance, inc. - Functional Verification Working Group. 38p. VER 1 1.2. http://www.vsi.org/library/specs/ver111.pdf (Page consultée le 3 avril 2003).

[71] WALKER, Robert J., BANIASSAD, Elisa L. A., MURPHY, Gail C. 1999. « An initial Assessment of aspect-oriented programming ». *Proceedings of the 21st International Conference on Software Engineering (ICSE'99).* [En ligne]. Los Angeles, CA: ICSE'99. 120-130. http://www.cs.ubc.ca/labs/se/papers/1999/icse99-aop.pdf (Page consutée le 3 avril 2003).

[72] WHITEMORE, Paul, DEARTH, Glenn. 1999. « Object-Oriented Approach to Verification ». *Synopsys Users Group Conference (SNUG).* [En ligne]. Boston, MA: Synopsys, Inc. http://solvnet.synopsys.com/wwwauth/wwwauth:8000/news/pubs/snug/bostonsnug99_papers/whittemore_final.pdf (Page consultée le 3 avril 2003).

ANNEXES

ANNEXE A

Questionnaire du sondage sur la vérification fonctionnelle

Please provide the answer that best matches your current verification situation.

Q1 **How important is functional verification?**

1 VERY IMPORTANT
2 IMPORTANT
3 UNDECIDED OR UNSURE
4 SLIGHTLY IMPORTANT
5 NOT IMPORTANT

Q2 **How important is functional verification for your company?**

1 VERY IMPORTANT
2 IMPORTANT
3 UNDECIDED OR UNSURE
4 SLIGHTLY IMPORTANT
5 NOT IMPORTANT

Q3 **One year from now, how important will functional verification be to your company?**

1 VERY IMPORTANT
2 IMPORTANT
3 UNDECIDED OR UNSURE
4 SLIGHTLY IMPORTANT
5 NOT IMPORTANT

Q4 **In your company, what percentage of the development process is dedicated to functional verification?**

_____%

Q5 **Does your company have people specifically dedicated to functional verification?**

1 YES
2 NO

If yes, what percentage of the development team is specifically dedicated to functional verification?

_____%

Q6 **Do designers in your company perform functional verification?**

1 YES
2 NO

If yes, what percentage of a typical designer's project development effort is dedicated to functional verification in your organization?

_____%

Q7 **Do you believe that testbench design is more of a hardware design or of a software design?**

1 HARDWARE ONLY
2 MOSTLY HARDWARE
3 HARDWARE AND SOFTWARE
4 MOSTLY SOFTWARE
5 SOFTWARE ONLY

Q8 **At which frequency does each type of error happen?**

(Please circle your answer.)

1	SPECIFICATION ERRORS	VERY FREQUENTLY	FREQUENTLY	SOMETIMES	NEVER
2	SPECIFICATION INTERPRETATION ERRORS	VERY FREQUENTLY	FREQUENTLY	SOMETIMES	NEVER
3	DESIGN IMPLEMENTATION ERRORS	VERY FREQUENTLY	FREQUENTLY	SOMETIMES	NEVER
4	TEST BENCH ERRORS	VERY FREQUENTLY	FREQUENTLY	SOMETIMES	NEVER

Q9 **How important are the consequences of each type of error?**

(Please circle your answer.)

1	SPECIFICATION ERRORS	VERY IMPORTANT	IMPORTANT	SLIGHTLY IMPORTANT	NOT IMPORTANT
2	SPECIFICATION INTERPRETATION ERRORS	VERY IMPORTANT	IMPORTANT	SLIGHTLY IMPORTANT	NOT IMPORTANT
3	DESIGN IMPLEMENTATION ERRORS	VERY IMPORTANT	IMPORTANT	SLIGHTLY IMPORTANT	NOT IMPORTANT
4	TEST BENCH ERRORS	VERY IMPORTANT	IMPORTANT	SLIGHTLY IMPORTANT	NOT IMPORTANT

Q10 **In your company's development process, when does the verification planning begin?**

1 AT THE BEGINNING OF THE DEVELOPMENT PROCESS
2 DURING SYSTEM DESIGN
3 DURING RTL DESIGN
4 NO VERIFICATION PLANNING IS ACCOMPLISHED
5 OTHER (PLEASE SPECIFY) _____

Q11 **How are non-specified elements in the design specification treated?**
(Circle the number(s) for your response)

1 THEY ARE TREATED AS "DON'T CARE"
2 THEY TRIGGER THE GENERATION OF A NEW REVISION OF THE SPECIFICATION
3 THE SPECIFICATION INTERPRETATION IS LEFT TO THE DESIGNER
4 OTHER (PLEASE SPECIFY) _____

Q12 **Does your company create verification plans?**

1 YES
2 NO
3 MAYBE/I DO NOT KNOW

If yes, are verification plans built in an organized or in an ad hoc fashion?

1 ORGANIZED
2 AD HOC

If you use a verification plan, who does participate in its creation?
(Circle the number(s) for your response)

1 ENGINEERS DEDICATED TO VERIFICATION
2 JUNIOR DESIGNERS
3 SENIOR DESIGNERS
4 PROJECT LEADER
5 TECHNICAL PROJECT MANAGER
6 NON-TECHNICAL PROJECT MANAGER

Q13 **In your company's development process, when do the testbench designs begin?**

1 AT THE BEGINNING OF THE DEVELOPMENT PROCESS
2 DURING SYSTEM DESIGN
3 DURING RTL DESIGN
4 OTHER (PLEASE SPECIFY) _____

Q14 **At which level of granularity do you verify?**

(Circle the number(s) for your response)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN UNITS
3 IP
4 SYSTEM
5 I DO NOT DO VERIFICATION
6 I DON'T KNOW

Q15 **Do you implement bus functional models (BFM) in your testbenches?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, at which verification level(s) do you use bus functional models?

(Circle the number(s) for your response)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN UNITS
3 IP
4 SYSTEM
5 I DON'T KNOW

Q16 **At which level(s) do you use each verification approach**

(Please circle your answer(s).)

1	BLACK BOX	UNIT	INTEGRATION	IP	SYSTEM	NONE	I DON'T KNOW
2	WHITE BOX	UNIT	INTEGRATION	IP	SYSTEM	NONE	I DON'T KNOW
3	GREY BOX	UNIT	INTEGRATION	IP	SYSTEM	NONE	I DON'T KNOW

Q17 **At which verification level do you use each simulation tool?**

(Please circle your answer(s).)

1	MODELSIM® BY MODEL TECHNOLOGY	UNIT	INTEGRATION	IP	SYSTEM	NONE
2	COCENTRIC® PRODUCT FAMILY BY SYNOPSYS	UNIT	INTEGRATION	IP	SYSTEM	NONE
3	SCIROCCO™ BY SYNOPSYS	UNIT	INTEGRATION	IP	SYSTEM	NONE
4	VCS™ BY SYNOPSYS	UNIT	INTEGRATION	IP	SYSTEM	NONE
5	SOFTWARE FAMILY NC BY CADENCE	UNIT	INTEGRATION	IP	SYSTEM	NONE
6	SEAMLESS™ BY MENTOR GRAPHICS	UNIT	INTEGRATION	IP	SYSTEM	NONE
7	MATLAB™	UNIT	INTEGRATION	IP	SYSTEM	NONE
8	OTHER (PLEASE SPECIFY) _____	UNIT	INTEGRATION	IP	SYSTEM	NONE

Q18 **At which verification level do you use each type of formal verification?**

(Please circle your answer(s).)

1	EQUIVALENCE CHECKING	UNIT	INTEGRATION	IP	SYSTEM	NONE	NOT FAMILIAR
2	PROPERTY/ MODEL CHECKING	UNIT	INTEGRATION	IP	SYSTEM	NONE	NOT FAMILIAR
3	THEOREM PROVING	UNIT	INTEGRATION	IP	SYSTEM	NONE	NOT FAMILIAR

If you do formal verification, which formal verification tools do you use for each level?

1 UNITS : _____

2 INTEGRATION : _____

3 IP : _____

4 SYSTEM : _____

Q19 **Do you perform semi-formal verification?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, at which verification level do you perform semi-formal verification and which semi-formal tool(s) do you use for each selected verification level?
(Circle the number(s) for your response)

		Tools
1	FUNCTIONAL DESIGN UNITS	_____
2	INTEGRATION OF FUNCTIONAL DESIGN BLOCS	_____
3	IP	_____
4	SYSTEM	_____
5	I DON'T KNOW	_____

Q20 **At which verification level do you use each language?**
(Please circle your answer(s).)

1	VHDL	BLOCK	INTEGRATION	IP	SYSTEM	NONE
2	VERILOG	BLOCK	INTEGRATION	IP	SYSTEM	NONE
3	E	BLOCK	INTEGRATION	IP	SYSTEM	NONE
4	OPEN VERA®	BLOCK	INTEGRATION	IP	SYSTEM	NONE
5	C/C++	BLOCK	INTEGRATION	IP	SYSTEM	NONE
6	TCL SCRIPTS	BLOCK	INTEGRATION	IP	SYSTEM	NONE
7	PERL SCRIPTS	BLOCK	INTEGRATION	IP	SYSTEM	NONE
8	TESTBUILDER™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
9	SUPERLOG™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
10	OTHER (PLEASE SPECIFY) _____	BLOCK	INTEGRATION	IP	SYSTEM	NONE

Q21 **At which level do you use each verification tool?**

(Please circle your answer(s).)

1	SPECMAN ELITE™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
2	VERA™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
3	VERIFICATION COCKPIT™ BY CADENCE	BLOCK	INTEGRATION	IP	SYSTEM	NONE
4	BESTBENCH™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
5	TESTBENCHER PRO™	BLOCK	INTEGRATION	IP	SYSTEM	NONE
6	OTHER (PLEASE SPECIFY) _____	BLOCK	INTEGRATION	IP	SYSTEM	NONE

Q22 **At which verification level do you use each stimuli generation method?**

(Please circle your answer(s).)

1	DETERMINISTIC TESTS	BLOCK	INTEGRATION	IP	SYSTEM	NONE
2	RANDOM STIMULI	BLOCK	INTEGRATION	IP	SYSTEM	NONE
3	PSEUDO RANDOM DIRECTED TESTS	BLOCK	INTEGRATION	IP	SYSTEM	NONE
4	OTHER (PLEASE SPECIFY) _____	BLOCK	INTEGRATION	IP	SYSTEM	NONE

Q23 **Do you use embedded checkers in your testbenches to verify assertions during simulation?**

1 YES

2 NO

3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, at which verification level do you use embedded checkers?
(Please circle your answer(s).)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN BLOCKS
3 IP
4 SYSTEM
5 I DON'T KNOW

If you use embedded checkers, with which language(s) do you code the assertions?

1 VHDL
2 VERILOG
3 E
4 OPEN VERA
5 OTHER (PLEASE SPECIFY)_____

Q24 **In which way reuse is part of your verification methodology?**
(Circle the number(s) for your response)

1 WE HAVE AN INTERNAL VERIFICATION MODULES REUSE POLICY
2 WHEN NEEDED, WE IDENTIFY PARTS OF CODE TO REUSE
3 WE ARE USING PRE-DEFINED FRAMEWORKS FROM WHICH ARE BASED ALL OUR TESTBENCHES
4 WE DO NOT REUSE ANY PART
5 OTHER (PLEASE SPECIFY) _____

Q25 **Do you have a design for verification methodology?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHODOLOGY

If yes, please give an example.

_____

Q26 **Do you use golden vectors or golden models during simulation?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, at which verification level(s) do you use golden models/vectors?
(Circle the number(s) for your response)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN BLOCS
3 IP
4 SYSTEM
5 I DON'T KNOW

Q27 **At which verification level do you use scoreboards during simulation?**
(Circle the number(s) for your response)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN BLOCS
3 IP
4 SYSTEM
5 I DON'T KNOW

Q28 **Which HDL design language is being used in your company?**
(Circle the number(s) for your response)

1 VHDL
2 VERILOG
3 SYSTEMC
4 SUPERLOG
5 OTHER (PLEASE SPECIFY) _____

Q29 **Is code coverage part of your verification methodology?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, what type of code coverage do you use?
(Circle the number(s) for your response)

1 STATEMENT COVERAGETOGGLE COVERAGE
2 FSM ARC COVERAGE
3 VISITED STATE COVERAGE
4 TRIGGERING COVERAGE
5 BRANCH COVERAGE
6 EXPRESSION COVERAGE
7 PATH COVERAGE
8 SIGNAL COVERAGE
OTHER (PLEASE SPECIFY) _____

Which tool(s) do you use for code coverage?

_____

Q30 **Is functional coverage part of your verification methodology during simulation?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, at which verification level do you use functional coverage?
(Circle the number(s) for your response)

1 FUNCTIONAL DESIGN UNITS
2 INTEGRATION OF FUNCTIONAL DESIGN BLOCS
3 IP
4 SYSTEM
5 I DON'T KNOW

Q31 **Do you perform co-simulations?**

1 YES
2 NO
3 I AM NOT FAMILIAR WITH THIS METHOD

If yes, which co-simulation tool(s) do you use?
(Circle the number(s) for your response)

1 SEAMLESS BY MENTOR GRAPHICS
2 ONE OF THE COCENTRIC FAMILY OF TOOLS BY SYNOPSYS
3 EAGLEI BY SYNOPSYS
4 OTHER (PLEASE SPECIFY)? _____

Q32 **According to your methodology, what defines the end of the functional verification process?**
(Circle the number(s) for your response)

1 ALL TESTS HAVE SUCCEDED
2 CODE COVERAGE OBJCETIVES (ALL TYPES INCLUDED) HAVE ALL BEEN MET
3 FUNCTIONAL COVERAGE OBJECTIVES (ALL TYPES INCLUDED) HAVE ALL BEEN MET
4 TIME TO MARKET RESTRICTIONS
5 FINANCIAL RESTRICTIONS
6 OTHER (PLEASE SPECIFY) _____

Q33 **In your opinion, what can be reused in a testbench?**
(Circle the number(s) for your response)

1 TESTCASES
2 BUS FUNCTIONAL MODELS
3 COVERAGE ELEMENTS
4 ASSERTIONS
5 STIMULI GENERATION MODULES
6 OTHER (PLEASE SPECIFY) _____

Q34 **Do you use object-oriented programming to build your testbenches?**

1 YES
2 NO

Q35 **Do you know the aspect-oriented programming concept?**

1 YES
2 NO

If yes, do you think that aspect-oriented programming is useful to enhance the reuse potential of a testbench?

1 YES
2 NO
3 NO OPINION/NOT SURE

Q36 **When you think that a testbench module could eventually be reused without being a 100% sure that it will effectively be reused, which approach is more appropriate?**

1 A DESIGN FOR REUSE APPROACH SHOULD BE APPLIED
2 RE-DESIGN THE VERIFICATION MODULE THE NEXT TIME IT IS REQUIRED
3 OTHER (PLEASE SPECIFY) : _____

Q37 **In your opinion, what is missing in the verification process?**

_____

Q38 **Are there known limitations of the verification process in your organization that you can/are willing to share with us? Please explain briefly.**

_____

_____

We would like to know more about you…

Q39 **What is your title in your company?**

_____

Q40 **To which activity domain is your company related?**

1 TELECOMMUNICATIONS
2 MULTIMEDIA
3 MICROPROCESSORS
4 OTHER (PLEASE SPECIFY) _____

Q41 **How many years of experience in electronics do you possess?**

1 0 – 5 YEARS
2 6 – 10 YEARS
3 11 – 15 YEARS
4 15 – 20 YEARS
5 21 YEARS AND MORE

Q42 **What is your last completed degree?**

1 BACHELOR'S DEGREE
2 MASTER'S DEGREE
3 PHD
4 OTHER (PLEASE SPECIFY) _____

Q43 **In what field did you study?**

1 ELECTRICAL ENGINEERING
2 SOFTWARE ENGINEERING
3 OTHER (PLEASE SPECIFY) _____

Q44 **Do you have any special comments concerning this survey?**

_____

_____

_____

_____

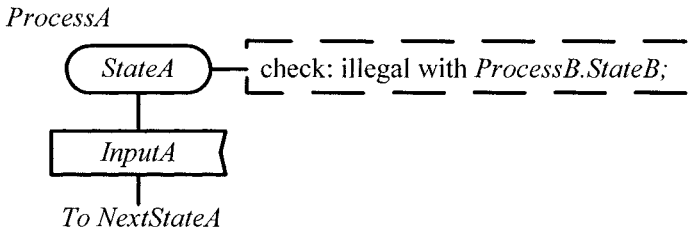
Q45 **If you are interested in receiving the results of this survey, please write down your email address below.**

_____

ANNEXE B

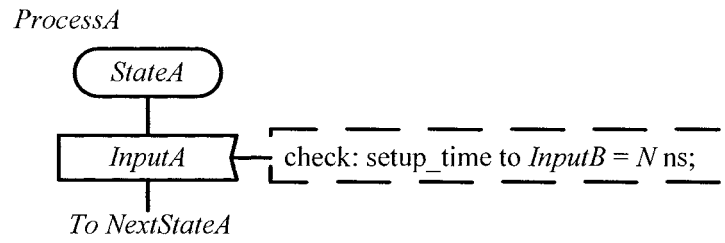
Patrons d'implantation d'assertions en langage *e* à partir d'assertions SDL

Assertion d'exclusion d'états



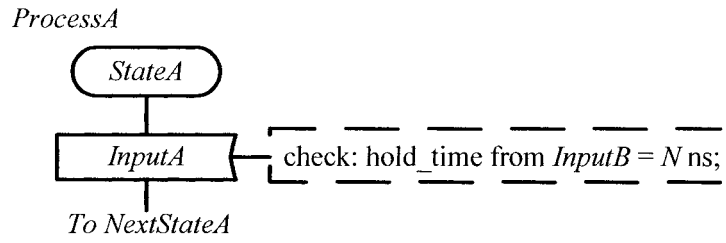

```
event InputA is [event signal mapping];

state_machine() @sys.any is
{
   all of
   {
      state machine ProcessA_State
      {
         StateA => NextStateA
         {
            check_machine();
            wait @InputA;
         };
         …
      };

      state machine ProcessB_State
      {
         StateB => …
         {
            …
         };
      };
   };
};


check_machine() is
{
   check that (not (ProcessA_State == StateA and
                    ProcessB_State == StateB));
};
```

Assertion de temps de préparation



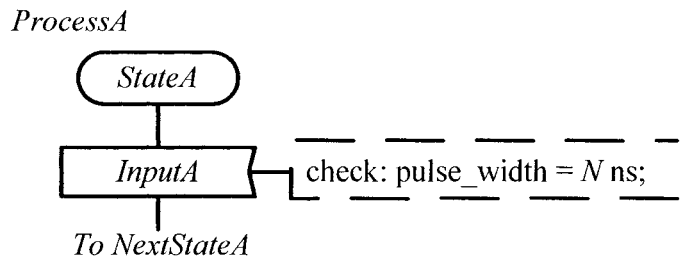

```
event InputA is [event signal mapping];
event InputA_setup is only {@InputA; delay(N)};
event InputB is [event signal mapping];

expect InputA_Setup is @InputA => {[..]*not @InputB;
                                   @InputA_setup}
   else dut_error("InputA setup time to InputB violation!");
```

Assertion de temps de maintien




```
event InputA is [event signal mapping];
event InputA_hold is only {@InputB; delay(N)};
event InputB is [event signal mapping]

expect InputA_Hold is @InputB => {[..]*not @InputA; @InputA_hold}
   else dut_error("InputA hold time from InputB violation!");
```

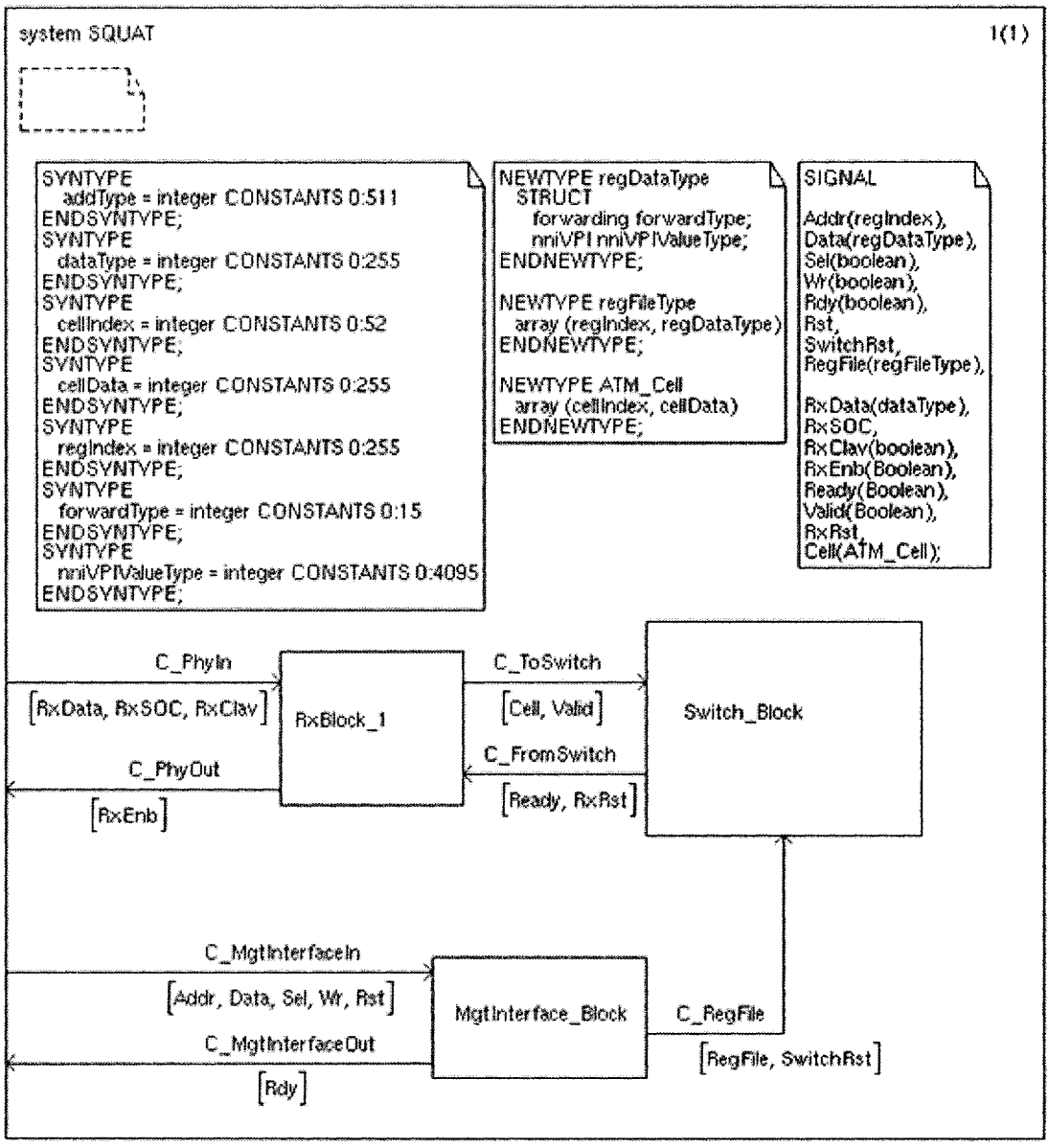
Assertion de temps de validation

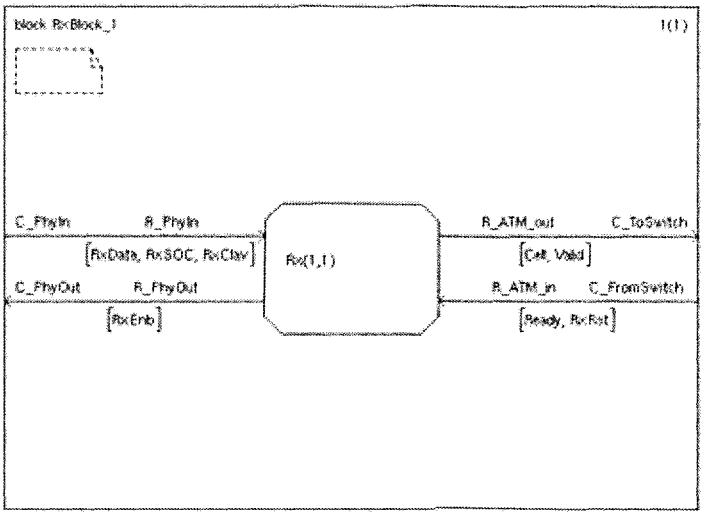


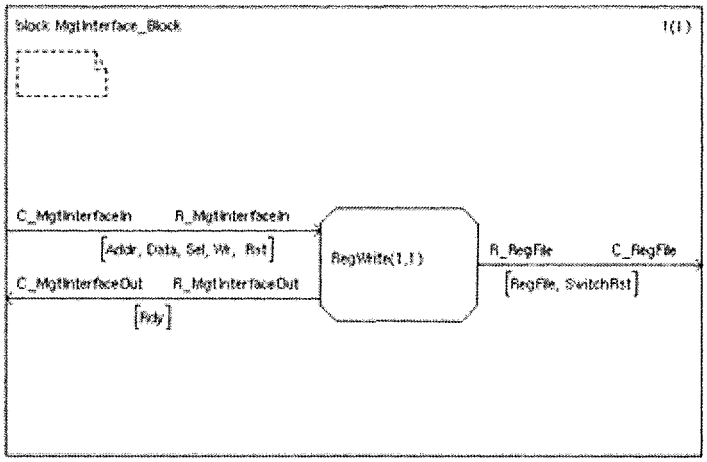

```
event InputA is [event signal mapping];
event end_InputA is [event signal mapping];
event InputA_width is only{@InputA; delay(N)};

expect InputA_Pulse is @InputA => {[..]*not @end_InputA;
                                   @InputA_width}
   else dut_error("InputA pulse width violation");
```

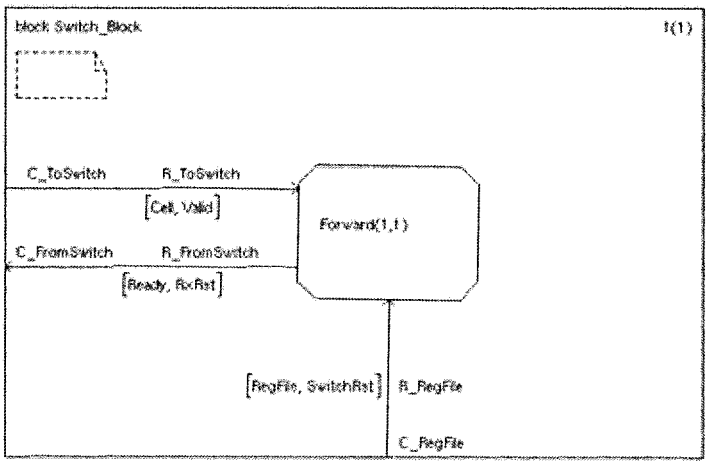
ANNEXE C

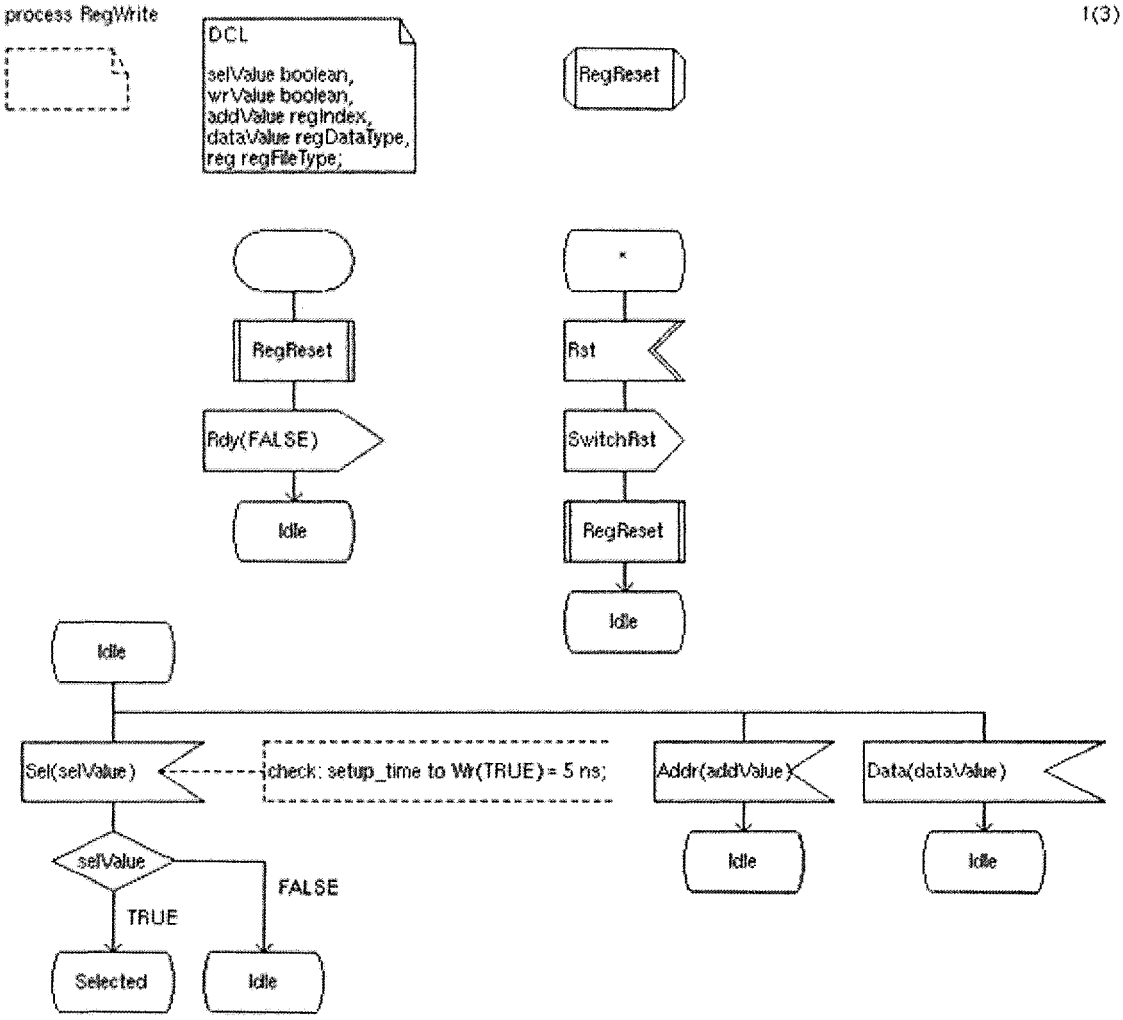
Spécification SDL du commutateur ATM

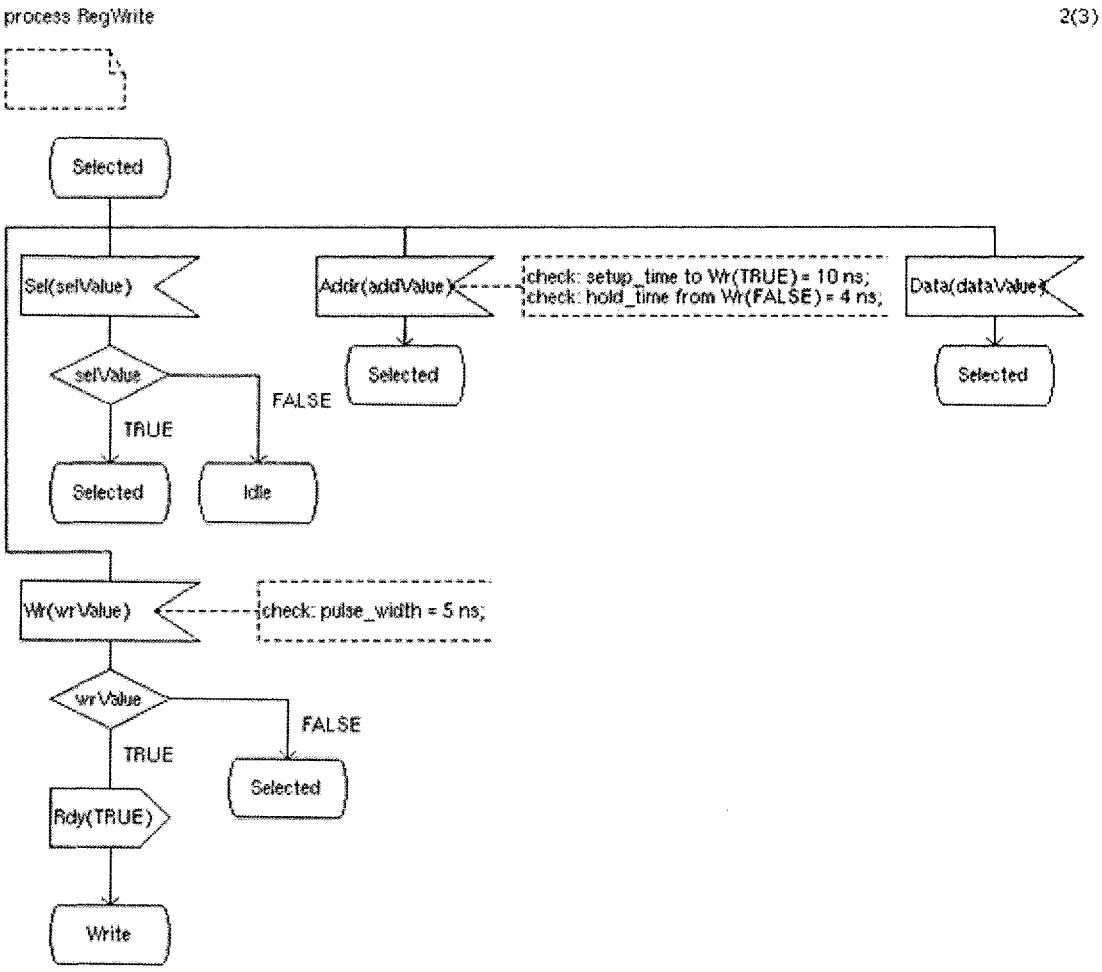



block RxBlock_1
1(1)
C_PhyIn
R_PhyIn
[RxData, RxSOC, RxClav]
Rx(1,1)
R_ATM_out
C_ToSwitch
[Cell, Valid]
C_PhyOut
R_PhyOut
[RxEnb]
R_ATM_in
C_FromSwitch
[Ready, RxRst]

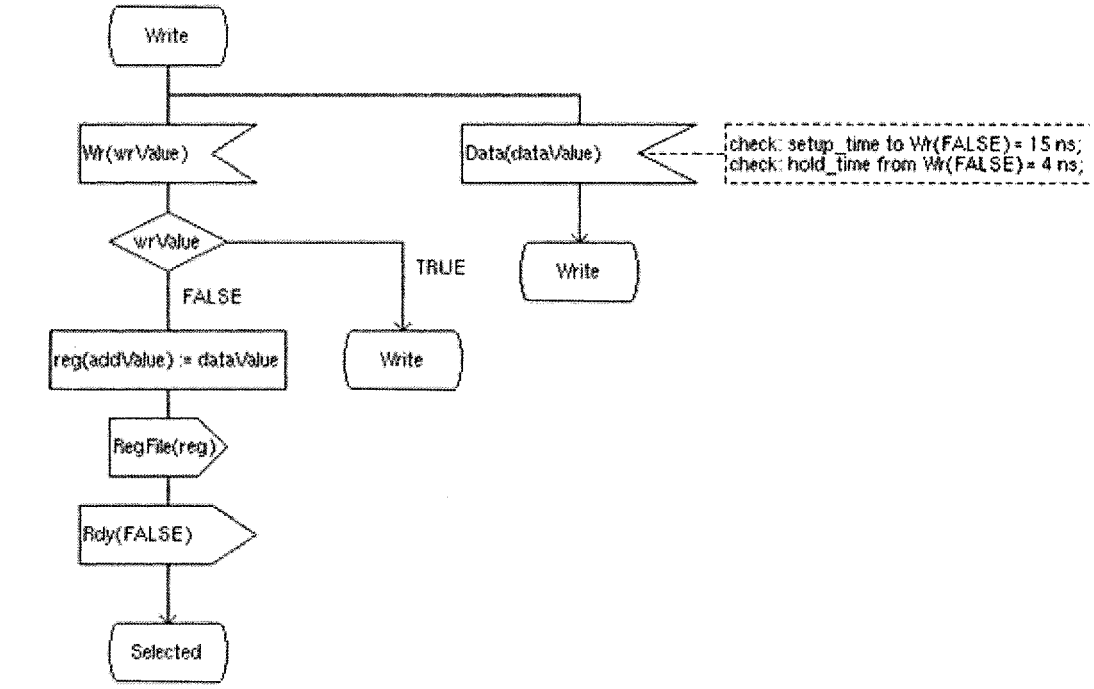

block MgtInterface_Block
1(1)
C_MgtInterfaceIn
R_MgtInterfaceIn
[Addr, Data, Sel, Wr, Rst]
RegWrite(1,1)
R_RegFile
C_RegFile
[RegFile, SwitchRst]
C_MgtInterfaceOut
R_MgtInterfaceOut
[Rdy]

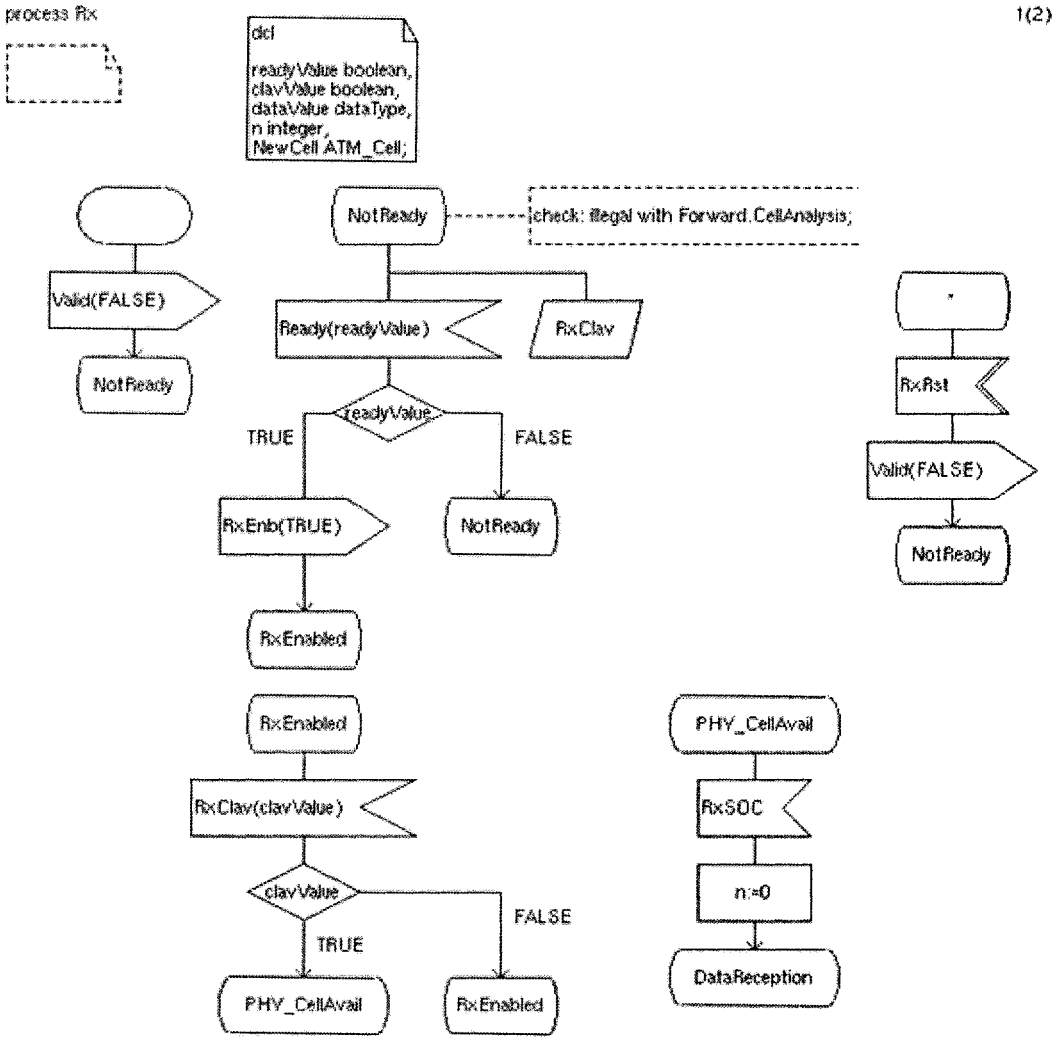

block Switch_Block
1(1)
C_ToSwitch
R_ToSwitch
[Cell, Valid]
Forward(1,1)
C_FromSwitch
R_FromSwitch
[Ready, RxRst]
[RegFile, SwitchRst]
R_RegFile
C_RegFile


process RegWrite
1(3)
DCL
selValue boolean,
wrValue boolean,
addrValue regIndex,
dataValue regDataType,
reg regFileType;
RegReset
RegReset
Rdy(FALSE)
Idle
*
Rst
SwitchRst
RegReset
Idle
Idle
Sel(selValue)
check: setup_time to Wr(TRUE) = 5 ns;
Addr(addrValue)
Data(dataValue)
Idle
Idle
selValue
TRUE
FALSE
Selected
Idle


process RegWrite
2(3)
Selected
Sel(selValue)
Addr(addValue)
check: setup_time to Wr(TRUE) = 10 ns;
check: hold_time from Wr(FALSE) = 4 ns;
Data(dataValue)
selValue
FALSE
TRUE
Selected
Selected
Selected
Idle
Wr(wrValue)
check: pulse_width = 5 ns;
wrValue
FALSE
TRUE
Selected
Rdy(TRUE)
Write

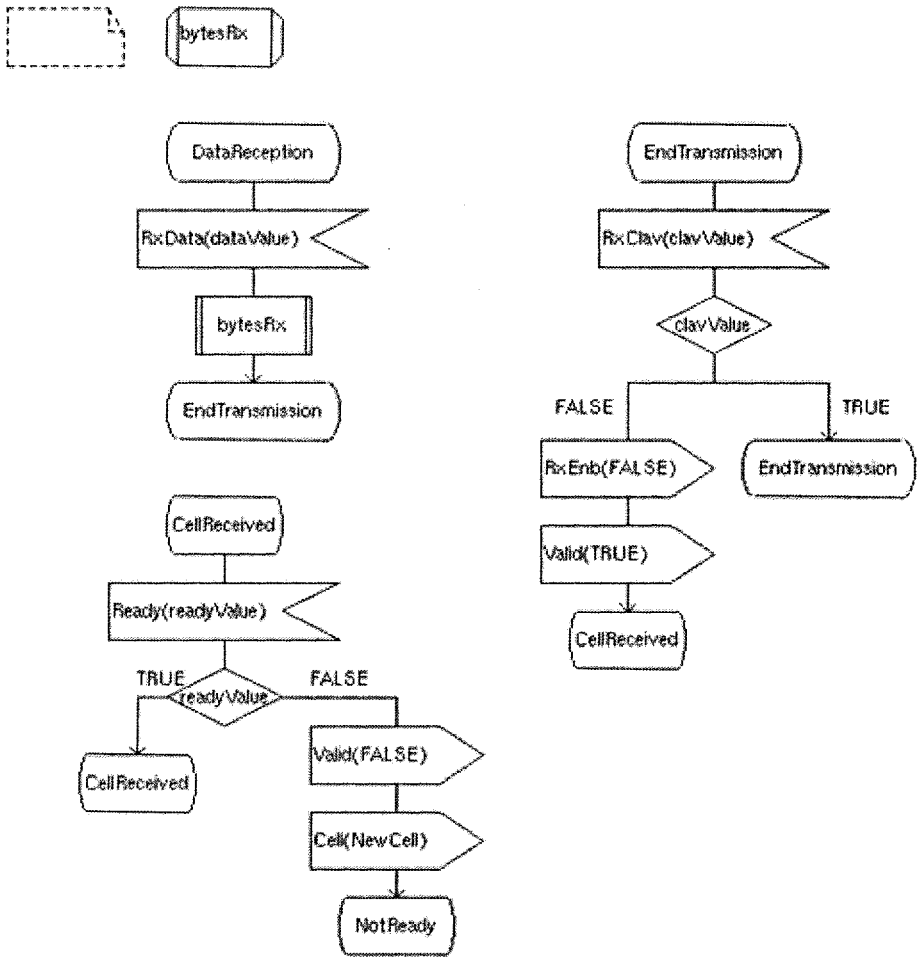
process RegWrite 3(3)

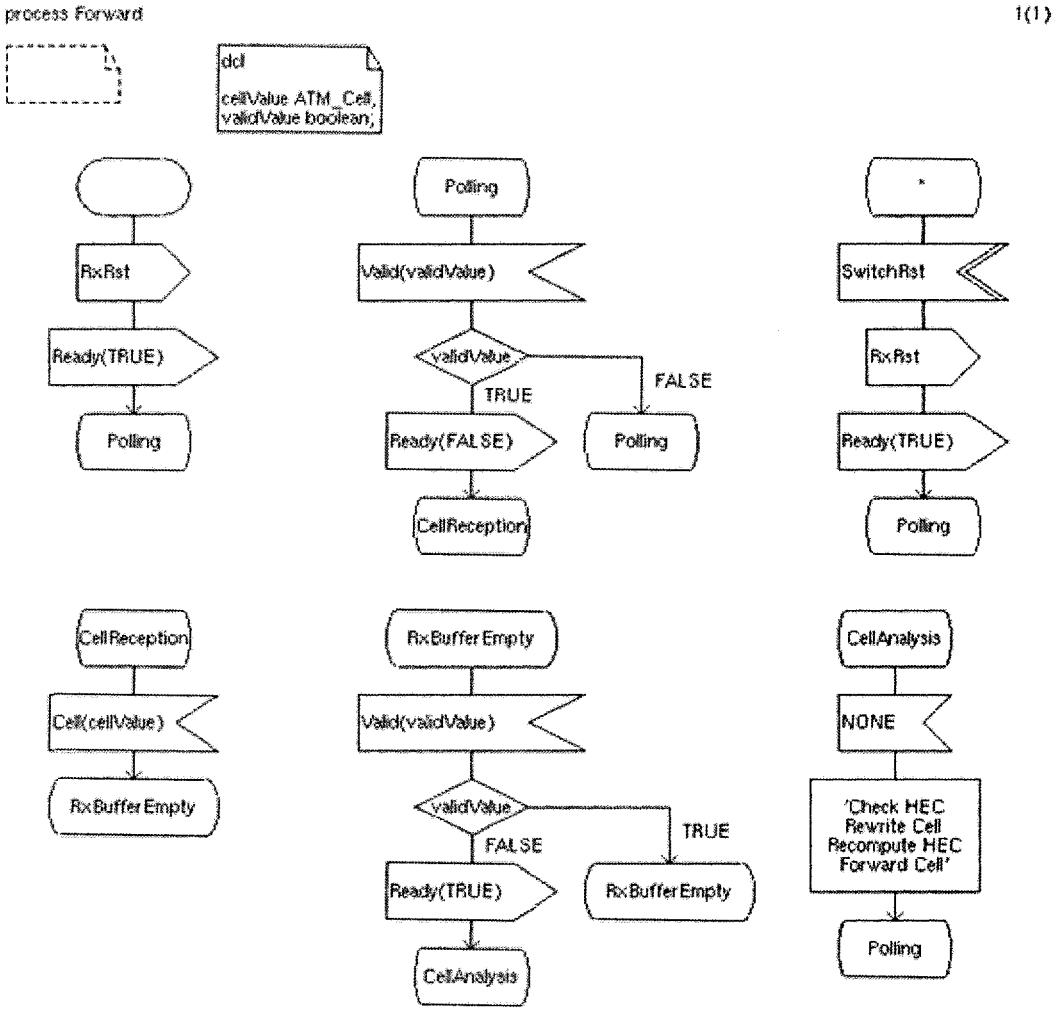



process Rx
1(2)
dcl
readyValue boolean,
clavValue boolean,
dataValue dataType,
n integer,
NewCell ATM_Cell;
NotReady
check: illegal with Forward.CellAnalysis;
Valid(FALSE)
NotReady
Ready(readyValue)
RxClav
readyValue
TRUE
FALSE
RxEnb(TRUE)
NotReady
RxEnabled
RxEnabled
RxClav(clavValue)
clavValue
TRUE
FALSE
PHY_CellAvail
RxEnabled
*
RxRst
Valid(FALSE)
NotReady
PHY_CellAvail
RxSOC
n:=0
DataReception

process Rx

2(2)




process Forward
1(1)
dcl
cellValue ATM_Cell,
validValue boolean;
RxRst
Ready(TRUE)
Polling
Polling
Valid(validValue)
validValue
TRUE
FALSE
Ready(FALSE)
Polling
CellReception
*
SwitchRst
RxRst
Ready(TRUE)
Polling
CellReception
Cell(cellValue)
RxBufferEmpty
RxBufferEmpty
Valid(validValue)
validValue
FALSE
TRUE
Ready(TRUE)
RxBufferEmpty
CellAnalysis
CellAnalysis
NONE
'Check HEC
Rewrite Cell
Recompute HEC
Forward Cell'
Polling

ANNEXE D

Code *e* généré par l'outil pour l'exemple du commutateur ATM

```
-------------------------------------------------------------------------
-- Title         : Assertion Checkers Module
-- Project       : SQUAT
-------------------------------------------------------------------------
-- File          : tb/base/SQUAT_Checkers.e
-- Author        : ACERT Application
-- Created       :
-- Last modified :
-------------------------------------------------------------------------
-- Description   : Definition of assertions to be verified dynamically
--                 during the simulation.
-------------------------------------------------------------------------
module: tb/base/SQUAT_Checkers.e;
aspect: base;
<'
-------------------------------------------------------------------------
--************************************************************************
--*********************CODE SECTION ADDED BY ACERT*********************
--************************************************************************
-------------------------------------------------------------------------
unit TimingChecker {

   id: uint;
   is_enabled: bool;

   event Addr;
   event Addr_holdTimeFrom_WrFALSE;
   event Addr_setupTimeTo_WrTRUE;
   event Data;
   event Data_holdTimeFrom_WrFALSE;
   event Data_setupTimeTo_WrFALSE;
   event Sel;
   event Sel_setupTimeTo_WrTRUE;
   event Wr;
   event WrFALSE;
   event WrTRUE;
   event Wr_pulseWidth;
   event end_Wr;

   expect @Sel => {[..]*not @WrTRUE; @Sel_setupTimeTo_WrTRUE};
   expect @Addr => {[..]*not @WrTRUE; @Addr_setupTimeTo_WrTRUE};
   expect @WrFALSE => {[..]*not @Addr; @Addr_holdTimeFrom_WrFALSE};
   expect @Wr => {[..]*not @end_Wr; @Wr_pulseWidth};
   expect @Data => {[..]*not @WrFALSE; @Data_setupTimeTo_WrFALSE};
   expect @WrFALSE => {[..]*not @Data; @Data_holdTimeFrom_WrFALSE};
};
'>
```

```
<'
--------------------------------------------------------------------------
--************************************************************************
--*********************CODE SECTION ADDED BY ACERT*************************
--************************************************************************
--------------------------------------------------------------------------

type RegWrite_StateType: [Idle, Selected, Write];
type Rx_StateType: [NotReady, RxEnabled, PHY_CellAvail,DataReception,
EndTransmission, CellReceived];
type Forward_StateType: [Polling, CellReception, RxBufferEmpty,
CellAnalysis];

unit StateChecker {

   id: uint;
   is_enabled: bool;

   !RegWrite_State: RegWrite_StateType;
   !Rx_State: Rx_StateType;
   !Forward_State: Forward_StateType;

   run() is {
      start state_machines();
   };

   event Addr;
   event Cell;
   event Data;
   event Data_FALSE;
   event Data_TRUE;
   event Ready_FALSE;
   event Ready_TRUE;
   event Rst;
   event RxClav_FALSE;
   event RxClav_TRUE;
   event RxData;
   event RxRst;
   event RxSOC;
   event Sel_FALSE;
   event Sel_TRUE;
   event SwitchRst;
   event Valid_FALSE;
   event Valid_TRUE;
   event Wr_FALSE;
   event Wr_TRUE;
```

```
state_machines() @sys.any is {
   all of
   {
      state machine RegWrite_State
      {
         Idle => Idle
         {
            check_state_machines();
            wait @Addr;
         };

         Idle => Idle
         {
            check_state_machines();
            wait @Data;
         };

         Idle => Selected
         {
            check_state_machines();
            wait @Sel_TRUE;
         };

         Idle => Idle
         {
            check_state_machines();
            wait @Sel_FALSE;
         };

         * => Idle
         {
            wait @Rst;
         };

         Selected => Selected
         {
            check_state_machines();
            wait @Addr;
         };

         Selected => Selected
         {
            check_state_machines();
            wait @Data;
         };

         Selected => Selected
         {
            check_state_machines();
            wait @Sel_TRUE;
         };
```

```
    Selected => Idle
    {
        check_state_machines();
        wait @Sel_FALSE;
    };

    Selected => Selected
    {
        check_state_machines();
        wait @Wr_FALSE;
    };

    Selected => Write
    {
        check_state_machines();
        wait @Wr_TRUE;
    };

    Write => Write
    {
        check_state_machines();
        wait @Data;
    };

    Write => Selected
    {
        check_state_machines();
        wait @Wr_FALSE;
    };

    Write => Write
    {
        check_state_machines();
        wait @Wr_TRUE;
    };

};


state machine Rx_State
{
    NotReady => RxEnabled
    {
        check_state_machines();
        wait @Ready_TRUE;
    };

    NotReady => NotReady
    {
        check_state_machines();
        wait @Ready_FALSE;
    };
```

```
    * => NotReady
    {
       wait @RxRst;
    };

    RxEnabled => PHY_CellAvail
    {
       check_state_machines();
       wait @RxClav_TRUE;
    };

    RxEnabled => RxEnabled
    {
       check_state_machines();
       wait @RxClav_FALSE;
    };

    PHY_CellAvail => DataReception
    {
       check_state_machines();
       wait @RxSOC;
    };

    DataReception => EndTransmission
    {
       check_state_machines();
       wait @RxData;
    };

    EndTransmission => CellReceived
    {
       check_state_machines();
       wait @RxClav_FALSE;
    };

    EndTransmission => EndTransmission
    {
       check_state_machines();
       wait @RxClav_TRUE;
    };

    CellReceived => NotReady
    {
       check_state_machines();
       wait @Ready_FALSE;
    };

    CellReceived => CellReceived
    {
       check_state_machines();
       wait @Ready_TRUE;
    };

};
```

```
            state machine Forward_State
            {
               Polling => CellReception
               {
                  check_state_machines();
                  wait @Valid_TRUE;
               };

               Polling => Polling
               {
                  check_state_machines();
                  wait @Valid_FALSE;
               };

               * => Polling
               {
                  wait @SwitchRst;
               };

               CellReception => RxBufferEmpty
               {
                  check_state_machines();
                  wait @Cell;
               };

               RxBufferEmpty => CellAnalysis
               {
                  check_state_machines();
                  wait @Valid_FALSE;
               };

               RxBufferEmpty => RxBufferEmpty
               {
                  check_state_machines();
                  wait @Valid_TRUE;
               };

               CellAnalysis => Polling
               {
                  check_state_machines();
               };

            };
         };
      };

      check_state_machines() is
      {
         check that (not (Rx_State == NotReady and
                          Forward_State == CellAnalysis));
      };
   };
   '>
```

```
<'
--------------------------------------------------------------------------------
--*****************************************************************************
--*********************CODE SECTION ADDED BY ACERT*****************************
--*****************************************************************************
--------------------------------------------------------------------------------
extend TimingChecker {
   parent: SQUAT_vring;
};
'>

<'
--------------------------------------------------------------------------------
--*****************************************************************************
--*********************CODE SECTION ADDED BY ACERT*****************************
--*****************************************************************************
--------------------------------------------------------------------------------
extend SQUAT_vring {
   TimingAssChecker: TimingChecker is instance;

   keep TimingAssChecker.parent == me;
   keep TimingAssChecker.is_enabled == TRUE;
};
'>

<'
--------------------------------------------------------------------------------
--*****************************************************************************
--*********************CODE SECTION ADDED BY ACERT*****************************
--*****************************************************************************
--------------------------------------------------------------------------------
extend StateChecker {
   parent: SQUAT_vring;
};
'>

<'
--------------------------------------------------------------------------------
--*****************************************************************************
--*********************CODE SECTION ADDED BY ACERT*****************************
--*****************************************************************************
--------------------------------------------------------------------------------
extend SQUAT_vring {
   stateAssChecker: StateChecker is instance;

   keep stateAssChecker.parent == me;
   keep stateAssChecker.is_enabled == TRUE;
};
'>
```

```
<'
--------------------------------------------------------------------------
--************************************************************************
--*********************CODE SECTION ADDED BY ACERT************************
--************************************************************************
--------------------------------------------------------------------------
extend TimingChecker {
   when is_enabled TimingChecker {
       Addr : string;
       Data : string;
       Sel : string;
       Wr : string;
       WrFALSE : string;
       WrTRUE : string;

       //event Addr is only change ('(Addr)');
       //event Addr_holdTimeFrom_WrFALSE is only {@Addr; delay(4)};
       //event Addr_setupTimeTo_WrTRUE is only {@Addr; delay(10)};
       //event Data is only change ('(Data)');
       //event Data_holdTimeFrom_WrFALSE is only {@Data; delay(4)};
       //event Data_setupTimeTo_WrFALSE is only {@Data; delay(15)};
       //event Sel is only change ('(Sel)');
       //event Sel_setupTimeTo_WrTRUE is only {@Sel; delay(5)};
       //event Wr is only rise|fall|change ('(Wr)');
       //event WrFALSE is only rise|fall|change ('(WrFALSE)');
       //event WrTRUE is only rise|fall|change ('(WrTRUE)');
       //event Wr_pulseWidth is only {@Wr; delay(5)};
       //event end_Wr is only rise|fall|change ('(Wr)');
   };
};
'>

<'
--------------------------------------------------------------------------
--************************************************************************
--*********************CODE SECTION ADDED BY ACERT************************
--************************************************************************
--------------------------------------------------------------------------
extend StateChecker {
   when is_enabled StateChecker {
      Addr_s: string;
      Cell_s: string;
      Data_s: string;
      Data_FALSE_s: string;
      Data_TRUE_s: string;
      Ready_FALSE_s: string;
      Ready_TRUE_s: string;
      Rst_s: satring;
      RxClav_FALSE_s: string;
      RxClav_TRUE_s: string;
      RxData_s: string;
      RxRst_s: string;
      RxSOC_s: string;
      Sel_FALSE_s: string;
```

```
      Sel_TRUE_s: string;
      SwitchRst_s: string;
      Valid_FALSE_s: string;
      Valid_TRUE_s: string;
      Wr_FALSE_s: string;
      Wr_TRUE_s: string;

      //event Addr is only rise|fall|change ('(Addr_s)')@sim;
      //event Cell is only rise|fall|change ('(Cell_s)')@sim;
      //event Data is only rise|fall|change ('(Data_s)')@sim;
      //event Data_FALSE is only rise|fall|change
      //('(Data_FALSE_s)')@sim;
      //event Data_TRUE is only rise|fall|change ('(Data_TRUE_s)')@sim;
      //event Ready_FALSE is only rise|fall|change
      //('(Ready_FALSE_s)')@sim;
      //event Ready_TRUE is only rise|fall|change
      //('(Ready_TRUE_s)')@sim;
      //event Rst is only rise|fall|change ('(Rst_s)')@sim;
      //event RxClav_FALSE is only rise|fall|change
      //('(RxClav_FALSE_s)')@sim;
      //event RxClav_TRUE is only rise|fall|change
      //('(RxClav_TRUE_s)')@sim;
      //event RxData is only rise|fall|change ('(RxData_s)')@sim;
      //event RxRst is only rise|fall|change ('(RxRst_s)')@sim;
      //event RxSOC is only rise|fall|change ('(RxSOC_s)')@sim;
      //event Sel_FALSE is only rise|fall|change ('(Sel_FALSE_s)')@sim;
      //event Sel_TRUE is only rise|fall|change ('(Sel_TRUE_s)')@sim;
      //event SwitchRst is only rise|fall|change ('(SwitchRst_s)')@sim;
      //event Valid_FALSE is only rise|fall|change
      //('(Valid_FALSE_s)')@sim;
      //event Valid_TRUE is only rise|fall|change
      //('(Valid_TRUE_s)')@sim;
      //event Wr_FALSE is only rise|fall|change ('(Wr_FALSE_s)')@sim;
      //event Wr_TRUE is only rise|fall|change ('(Wr_TRUE_s)')@sim;
   };
};
'>
<'
-------------------------------------------------------------------------
--*****************************************************************
--********************CODE SECTION ADDED BY ACERT*********************
--*****************************************************************
-------------------------------------------------------------------------
extend TimingChecker {
   when is_enabled TimingChecker {
      //keep soft Addr == "Enter HDL Signal Name";
      //keep soft Data == "Enter HDL Signal Name";
      //keep soft Sel == "Enter HDL Signal Name";
      //keep soft Wr == "Enter HDL Signal Name";
      //keep soft WrFALSE == "Enter HDL Signal Name";
      //keep soft WrTRUE == "Enter HDL Signal Name";
   };
};
'>
```

```
<'
--------------------------------------------------------------------------
--************************************************************************
--********************CODE SECTION ADDED BY ACERT*************************
--************************************************************************
--------------------------------------------------------------------------

extend StateChecker {
   when is_enabled StateChecker {

      //keep soft Addr_s == "Enter HDL Signal Name";
      //keep soft Cell_s == "Enter HDL Signal Name";
      //keep soft Data_s == "Enter HDL Signal Name";
      //keep soft Data_FALSE_s == "Enter HDL Signal Name";
      //keep soft Data_TRUE_s == "Enter HDL Signal Name";
      //keep soft Ready_FALSE_s == "Enter HDL Signal Name";
      //keep soft Ready_TRUE_s == "Enter HDL Signal Name";
      //keep soft Rst_s == "Enter HDL Signal Name";
      //keep soft RxClav_FALSE_s == "Enter HDL Signal Name";
      //keep soft RxClav_TRUE_s == "Enter HDL Signal Name";
      //keep soft RxData_s == "Enter HDL Signal Name";
      //keep soft RxRst_s == "Enter HDL Signal Name";
      //keep soft RxSOC_s == "Enter HDL Signal Name";
      //keep soft Sel_FALSE_s == "Enter HDL Signal Name";
      //keep soft Sel_TRUE_s == "Enter HDL Signal Name";
      //keep soft SwitchRst_s == "Enter HDL Signal Name";
      //keep soft Valid_FALSE_s == "Enter HDL Signal Name";
      //keep soft Valid_TRUE_s == "Enter HDL Signal Name";
      //keep soft Wr_FALSE_s == "Enter HDL Signal Name";
      //keep soft Wr_TRUE_s == "Enter HDL Signal Name";
   };
};

'>
```

ANNEXE E

Aspect Partitioning for Hardware Verification Reuse

Sébastien Regimbal[1], Jean-François Lemire[2], Yvon Savaria[1], Guy Bois[2], El Mostapha Aboulhamid[3] and André Baron[4]

[1] *Electrical Engineering Department, École Polytechnique de Montréal, P.O. Box 6079, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3A7.*
[2] *Computer Engineering Department, École Polytechnique de Montréal, P.O. Box 6079, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3A7.*
[3] *Department of Computer Science and Operational Research, Université de Montréal, C.P. 6128, Succ. Centre-ville, Montréal, Québec, Canada, QC H3C 3J7.*
[4] *PMC-Sierra, 3333 Graham Blvd, Suite 500, Ville Mont-Royal, Québec, Canada, H3R 3L5.*

Key words: Functional Verification, Aspect-Oriented Programming, Object-Oriented Modeling.

Abstract: Technology advances strongly impact integrated circuits (IC) complexity. Current IC design methods have difficulties to handle this growth. In particular, hardware verification has become the main bottleneck of any major digital design effort. It is thus necessary to develop efficient methodologies for designing verification environments. To deal with this complexity, hardware verification languages (HVL), such as e, allow reducing the verification effort and contribute to raise the level of abstraction at which test benches are described. In addition, aspect-oriented programming (AOP) is an emerging technique, which promotes a better separation of concerns and improves reusability. We propose a partitioning method based on e, which uses AOP to facilitate verification program development, and we apply our method to a concrete example; verification of 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 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.

HDL languages such as VHDL or Verilog have improved hardware design by allowing high-level hardware descriptions. However, because hardware verification needs some sophisticated mechanisms that HDL languages do not possess, like an advanced temporal logic or complex data structures, HDL languages are not ideal for verification tasks. The use of Hardware Verification Languages (HVL), such as Verisity's e [5], improves the verification process by addressing the requirements of complex functional verification software development. 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 [2][3] are guidelines for the creation of object-oriented verification environments.

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. To our knowledge, the explicit use of an aspect-oriented approach in a concrete hardware verification environment has never been reported, even though [7] observed that a verification language such as Specman's e possesses constructs that can support AOP.

This paper shows how an aspect partitioning of verification environments promote reusability within and between projects. We propose a method that uses the aspect-oriented paradigm to enhance verification. The design under test (DUT) is a subset of a communication protocol converter SOC platform designed at the GRM (Groupe de Recherche en Micro-électronique) of École Polytechnique de Montréal. The paper discusses how a self-checking and configurable test bench was 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 a more formal partitioning methodology that facilitates reuse, through the use of aspects.

The remainder of the paper is organized as follows. In section 2, we review the methodology proposed for the design of our verification environment. In section 3, we briefly describe the DUT and we apply the methodology discussed in section 2 to verify this design. Section 4 discusses the lessons learned from applying our methodology to the verification example. Finally, in section 5, we present a conclusion and suggest possible areas of future investigation.

2. Proposed Methodology

2.1 Motivations

Ideally, placing a design in its real operational context is required to perform accurate verification. In general, such a context cannot be completely reproduced by simulation. Usually, it takes the form of a testbench able to:

- Emulate the environment of the design under test (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.

In this work, the proposed method facilitates reuse in a verification project. First, we verify a module, and by a bottom-up approach, we integrate the other parts of the design in a verification environment that is gradually constructed. This approach enables reusing some classes of our verification environment during the integration process.

2.2 Object-oriented analysis

The object-oriented decomposition methodology for the creation of our verification environment is based on three types of classes: the emulation classes, the verification classes and the utility classes. These class types are described in the Table 1.

Table 1. **Different Class Types**

Class Type	Definition
Emulation	An emulation class is a class that produces the same behaviour as the real hardware component that should be connected to the interface of the DUT. Moreover, emulation classes must be configurable, to allow the production of all test cases included in the verification plan.
Verification	Verification classes encompass the classes that enable the verification environment to be self-checking.
Utility	Utility classes comprise all other classes needed to complete the verification environment like data structures.

2.3 Aspect-Oriented Concepts

Aspect-oriented concepts are based on the principle of the separation of concerns [9]. An aspect defines a behaviour that affects several classes of a system. The object-oriented analysis organizes a system into a clear hierarchy of objects. Because some aspects are not confined in any object boundary, we say that they crosscut the system's modularity. An aspect-oriented methodology resolves that common modularity problem. For example, the inter-object communication elements of a class are usually encapsulated into each class of the objects hierarchy. However, if we group all inter-object communication elements into a single aspect that affect several classes, we can then pull

the communication aspects out of the original class. The reuse potential is then significantly improved, because this approach forces isolating design aspects, to avoid scattering them throughout the code. Consequently, with the aspect-oriented approach, we define objects by slices of behaviour and each slice defines an aspect of the system.

Once we subscribe to the idea of separating aspects, three questions [6] must be answered in order to pursue the development of a complete methodology:

- How to capture the aspects

  The e language solves this problem because it provides a construct, modules (e files), which can encapsulate aspects. In a module, we can create the base definition of a class and/or several extensions of other classes.

- What aspect composition mechanisms are to be used

  A composition mechanism is embedded in the e language with special extension features [7]. Specman Elite™ is then able to use the different defined aspects to generate the object-oriented framework of the verification environment. The available constructs of the e language needed to achieve this task are presented in [8].

- What issues need to be separated

  The answer to that question depends on the kind of applications. In our case, we want to develop a verification environment. Thus, in the next section, we propose an aspect partitioning solution to the problem of designing a verification environment.

2.3.1 Aspect Partitioning

An aspect-oriented analysis allows a concise separation of the verification environment by defining layers in each class of the verification environment. Decomposing the verification environment in aspects is the core of our method. It is important to determine the contribution of aspect partitioning to the verification design

effort. Our objective is to enhance the reusability of the verification environment. Thus, we must isolate the aspects of the environment that have a very low reuse potential. We have defined five aspect categories that encompass all the aspects of our verification environment. Each category defines an aspect of existing classes.

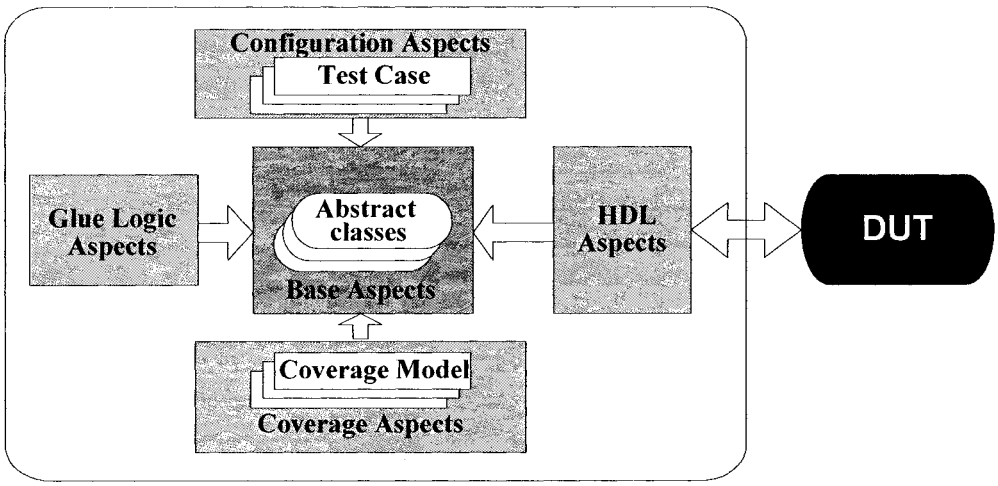


Figure 1. **Aspect categories**

As illustrated in Figure 1, the core of our aspect partitioning is the base aspects category. During the object-oriented analysis, we have determined that a certain set of classes was essential to define the core of our environment. The base aspects block includes the definition of all these classes. Each aspect from the other categories extends these base classes. The extensions are performed by adding features (aspects) to the base classes in other modules. These features have a very low reuse potential, and by defining them as new aspects, we separate them from the base aspects, which have a high reuse potential. Table 2 summarizes the purpose of each category.

Table 2. **Different Aspect Categories**

Aspect	**Definition**
Base	Defines the core of each class of the verification environment.
HDL	Define the connections and interactions of the classes with the DUT.
Glue Logic	Define the connections and interactions between classes.
Coverage	Define coverage models for the verification environment.

Aspect	Definition
Configuration	Define each test case necessary to achieve the verification plan of the DUT. Test cases are constraints added on top of the verification environment.

3. Application

3.1 The DUT

The DUT is a module of a SOC protocol conversion platform in development at the GRM of École Polytechnique de Montréal. Its purpose is to convert packets structured according to a network protocol stack into another one. 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 is the block that receives and memorizes the incoming packets (Figure 2). This module also creates packets related tags, identifies the incoming protocols and finally transmits the relevant information to a formatting engine.

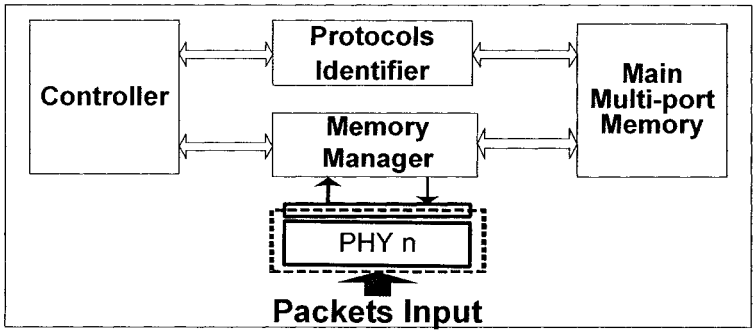


Figure 2. **DUT : Part of the Protocols Converter**

3.2 Partitioning Aspect Results

In this section, we explain how we applied the partitioning methodology developed in section 2 to verify the *Memory Manager* component. We also discuss how to extend the verification environment to cover the *Controller* component, by reusing certain aspects. The discussion is supported by concrete examples from our environment.

The main purpose of this environment is to emulate the DUT's functional interface to verify its behavior.

3.2.1 Memory Manager Verification

Figure 3 represents a functional view of the verification environment. It shows the main functional verification components used to stimulate the DUT and to verify its behavior.

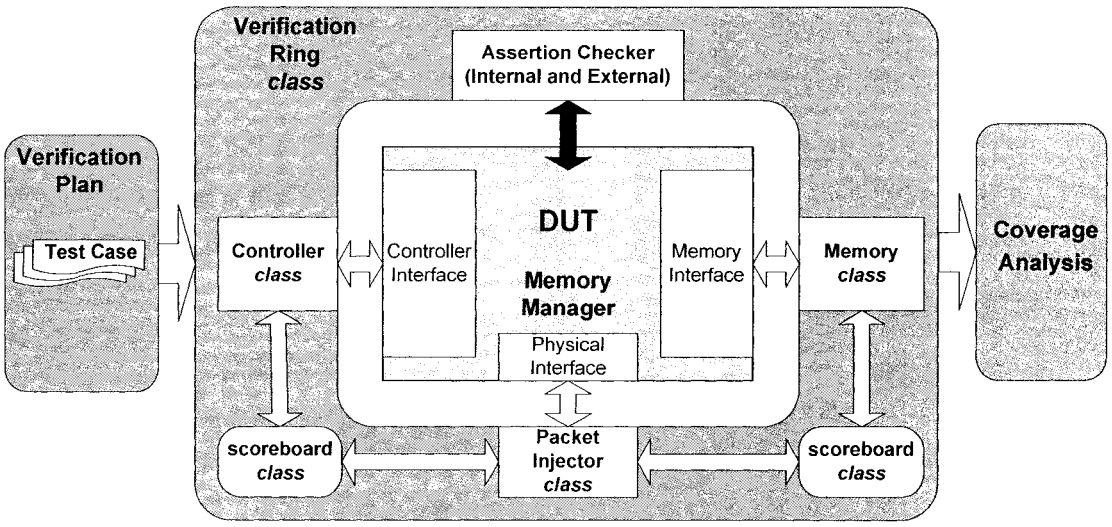


Figure 3. **Verification Environment Functional View**

In figure 3, we see that each functional verification component is associated with a class within the code structure. The *controller*, *memory* and *packet_injector* classes are part of the emulation type classes. The *assertion_checker* and *scoreboard* classes are part of the verification type classes. We created two *scoreboards* derived from a *scoreboard* super class, to ensure a self-checking process that verifies if the packets injected are transmitted adequately by the DUT, to the *controller* and the *memory*. The *assertion checker* class is a grey-box and black-box type assertion checker that uses the temporal logic features provided by the e language.

The main characteristic of our methodology resides in the encapsulation of aspects in modules (files) of our verification environment. Figure 5 shows the different aspects of our environment.

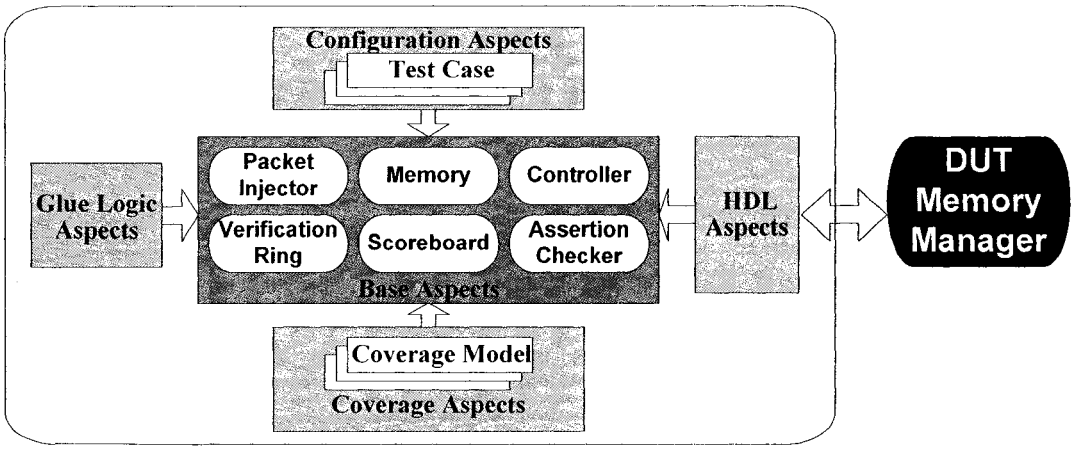


Figure 4. **Aspects framework**

At the highest-level of the environment, we have the *verification_ring* module aspect. It contains the highest-class definition of our class hierarchy. It is in this specific class that the environment initialization methods are defined. Each aspect in the base aspect group may contain class definitions that are extended in the modules connected by an arrow.

The purpose of the HDL aspect is to create a link between the environment modules and the DUT interface. Each aspect included in the base aspect group may have some of its classes extended to create a connection with the HDL code of the DUT. The glue logic aspect is responsible of connecting together all the other modules in the environment. Thus objects of the base aspect group classes are instantiated in this module. In the coverage aspect, coverage models are defined based on coverage events created in the extensions of the classes encapsulated in the base aspects. Finally, each test case of the configuration aspect category adds few constraints, on top of the existing environment, by extending some class of the environment. These test cases are then loaded on top of the current environment containing all the previously defined modules.

Figure 5 presents the implementation of the base aspect of the *packet_injector* class and a part of the glue logic aspect dedicated to our verification environment:

```
aspect: packet_injector (base)
<'
unit pkt_injector {
   pkts: list of pkt;
   ...
   keep soft pkts.size() == 6;
   inject()@sys.clk is{
      for each pkt (p) in pkts do {
         write_to_dut(p);
      };
   };
   write_to_dut(p: pkt)@sys.clk is empty;
};
'>
```

```
aspect: controller (base)
<'
unit controller {
   ...
   receive_pkt()@sys.clk is{
      var p  :pkt;
      wait @new_pkt;
      read_from_dut(p);
      ...
   };
   read_from_dut(p: *pkt)@sys.clk is empty;
};
'>
```

```
aspect: mm_glue_logic (Glue Logic)
<'
extend verification_ring {
   p: packet_injector  is instance;
   c: controller       is instance;
   s: scoreboard       is instance;
   keep p.father == me;
   keep c.father == me;
   keep s.father == me;
   ...
   init_test()@sys.clk is also {
      start p.inject();
   };
};
extend pkt_injector {
   father: verification_ring;
   inject()@sys.clk is also{
      father.s.addTransaction(p);
   };
};
extend controller {
   father: verification_ring;
   receave_pkt()@sys.clk is also{
      father.s.matchTransaction(p);
   };
};
'>
```

Figure 5. **Example of Aspect implementations**

The *packet_injector* class injects by its method a list of packets in the DUT. It is an abstract class because the method *write_to_dut* is undefined. We purposely avoid defining the *write_to_dut* method here, because this method does not belong to the base aspect of this class. It belongs to its HDL aspect.

The glue logic aspect, on the right of Figure 5, extends the base definition of the *verification_ring*, the *packet_injector* and the *controller* class. First, a *packet_injector*, a *controller* and a *scoreboard* are instantiated into the *verification_ring* class. The *packet_injector* sends packets into the DUT, the *controller* receives these packets from the DUT and the *scoreboard* verifies the progress of a packet through the DUT. This represents the connection part of the glue logic aspect. For the interaction part, we connect the *scoreboard* to the *packet_injector* and to the *controller* by extending the

inject and *receive_pkt* methods, with calls to the *scoreboard* methods *addTransaction* and *matchTransaction* respectively. We also start the packet injection with an extension of a method of the *verification_ring*.

3.2.2 Extension of the verification environment

To illustrate reusability within the same project, Figure 6 shows a functional view of the environment with the controller block added as a second design component to be verified. In this case, a version of the *controller* block replaces the class that emulates the controller block from the previous environment. Also, we see in Figure 6 that many classes can be reused to form the new environment. For instance, the base aspects, like *packet_injector* and *verification_ring*, have a high potential of reuse. By contrast, aspects like the glue logic aspect, discussed previously, are dedicated to a specific environment, since they establish the different connections between the environment classes. Then, these types of aspects are not easily reusable.

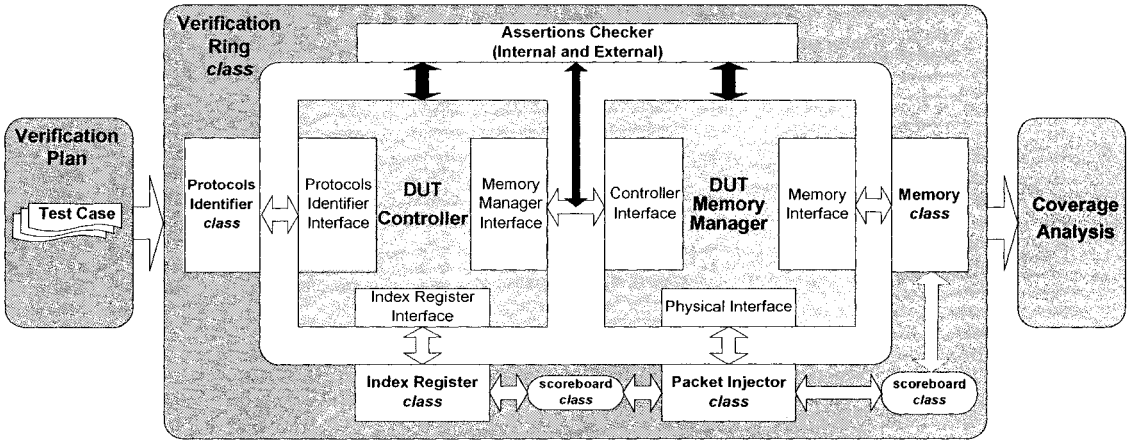


Figure 6. **Reusability Realization**

4. Discussion

In this section, we discuss the lessons learned from applying the method presented in the previous section.

- 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.

- 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 definition. After an experimentation of this technique, we found that the method proposed speed-up the implementation process by allowing an efficient parallel implementation.

- 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. Conclusion

In this work, an aspect partitioning applied to a hardware verification environment has been presented An aspect-oriented design methodology offers an efficient separation of concerns within an object-oriented environment. It has been shown that some aspects of our verification environment possess a high reuse potential. By contrast, aspects like the glue logic aspect are dedicated to a particular environment and are not reusable. A good direction to investigate would be to study how the reusable aspects could be

transformed into reusable frameworks, or executable patterns, and how the creation of non-reusable aspects could be automated in a functional verification platform.

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 Elite™ software by Verisity, and benefits from the technical guidance of PMC-Sierra. Finally, several Polytechnique's students, including Vincent Rocher, Gregory Donzel, and Mame 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] P. Whittemore and G. Dearth, Object-Oriented Approach to Verification, Proceeding of SNUG99 Boston, 1999.

[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 ECOOP97, pages 220-242, 1997.

[5] http://www.verisity.com

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

[7] Y. Hollander, M. Morley, and A. Noy, The e Language: A Fresh Separation of Concerns. Proceeding of TOOLS-38, pages 42-50, 2001.

[8] S. Regimbal, J.F. Lemire, Y. Savaria, G. Bois, E.M. Aboulhamid, A. Baron, Applying Aspect-Oriented Programming to Hardware Verification with e. accepted to HDLCON 2002.

[9] E.W. Dijkstra, A Discipline of Programming. Prentice Hall, 1976.

ANNEXE F

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] *Electrical Engineering Department, École Polytechnique de Montréal, P.O. Box 6079, Succ. Centre-Ville, Montréal, Québec, Canada, H3C 3A7. {regimbal, lemire, savaria, bois}@grm.polymtl.ca*

[2] *Department of Computer Science and Operational Research, Université de Montréal, C.P. 6128, Succ. Centre-ville, Montréal, Québec, Canada, QC H3C 3J7. aboulham@iro.umontreal.ca*

[3] *PMC-Sierra, 3333 Graham Blvd, Suite 500, Ville Mont-Royal, Québec, Canada, H3R 3L5. 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 Elite™ *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 caused by the complexity of test benches 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 of 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 artifact: 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.

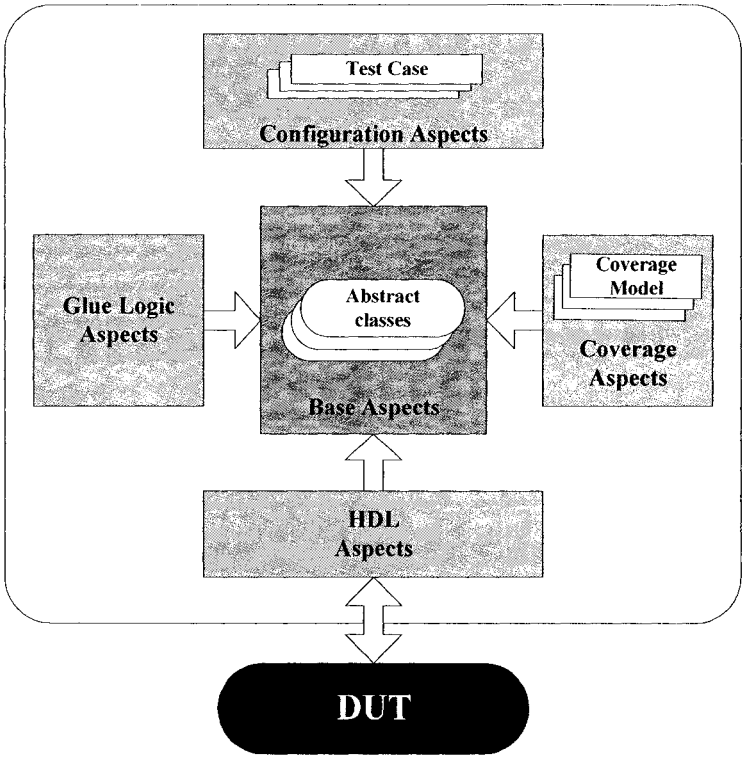


Figure 1 - Aspects of a verification environment

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.

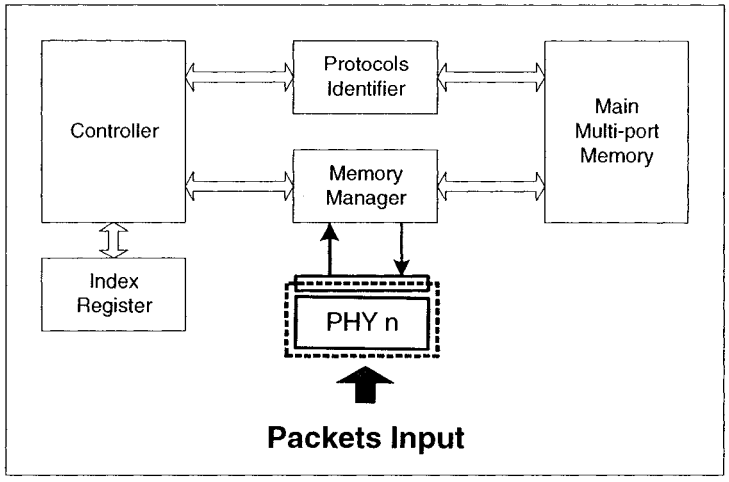


Figure 2 - Part of the protocol converter

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.

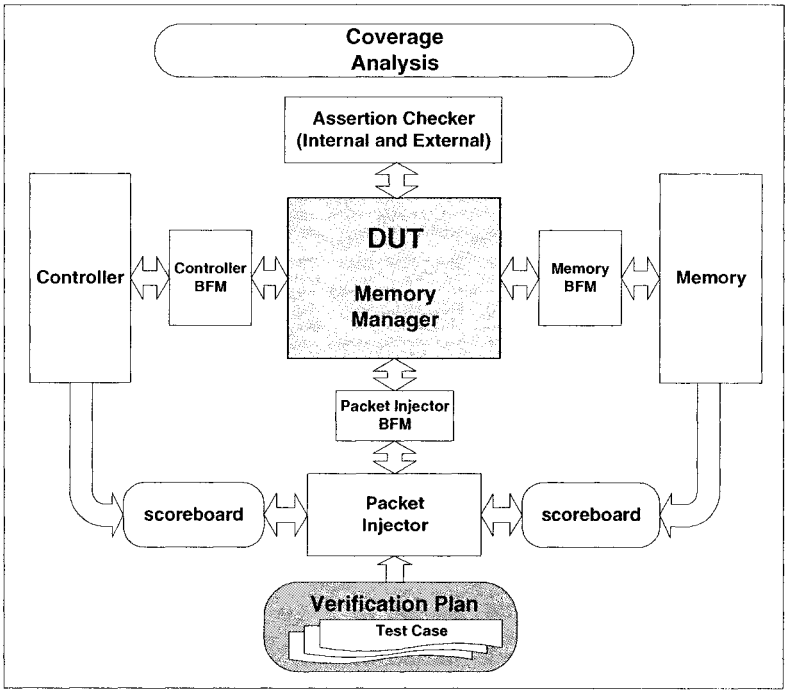


Figure 3 - Verification Environment (Functional View)

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.

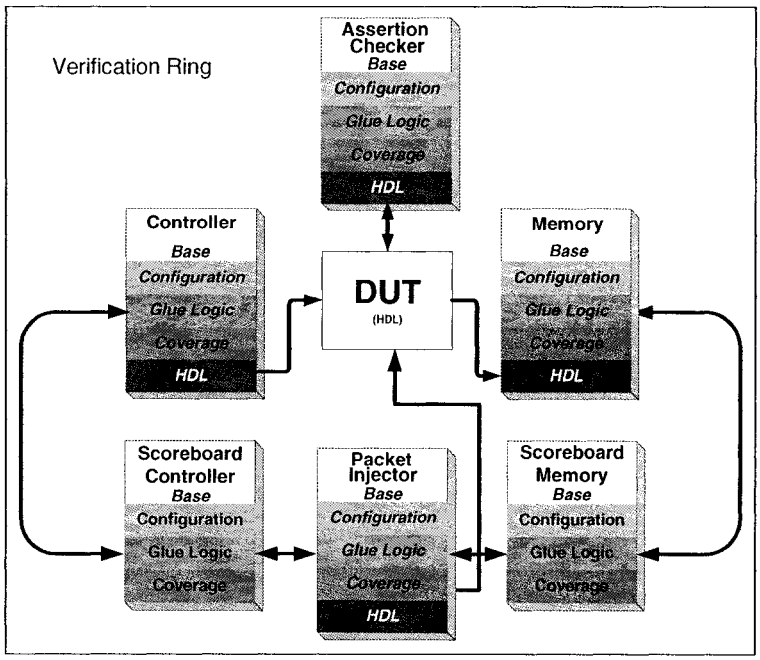


Figure 4 - Verification Environment (Aspect View)

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.

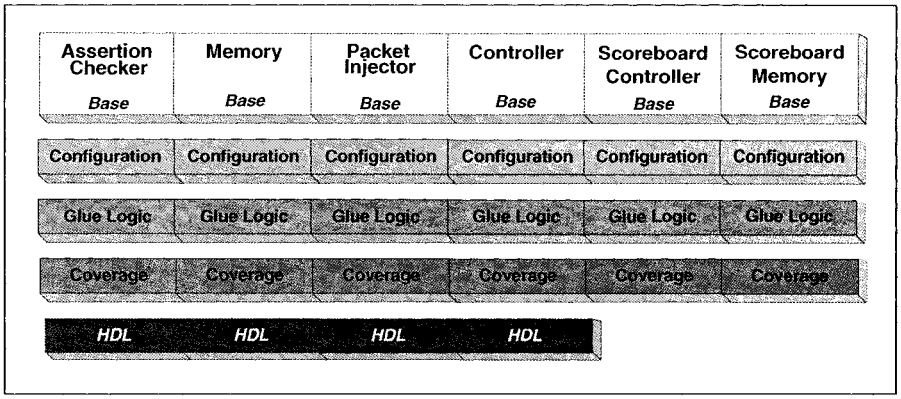


Figure 5 - Aspect Partitioning

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.

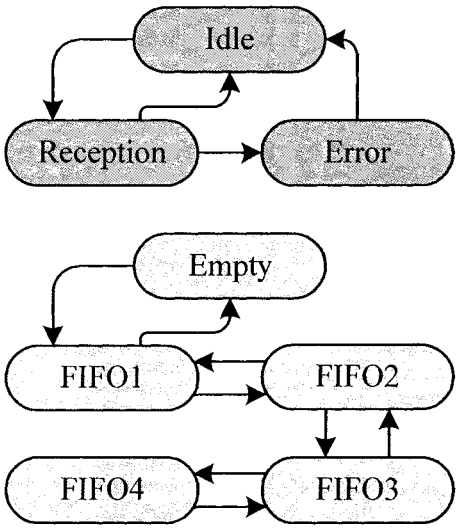


Figure 7 - State Machine

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.

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. 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. Even if at the time of completing this paper we did not investigate this issue in detail, we believe that the AOP methodology has a minor impact on performances.

7. 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 Elite™ software by Verisity, and benefits from the technical guidance of PMC-Sierra. Finally, several Polytechnique's students, including Vincent Rocher, Gregory Donzel, and Mame 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.