CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE D'ENSEIGNEMENT DE GRENOBLE

EXAMEN PROBATOIRE

en INFORMATIQUE

présenté par

Gilles BADOIL

Vérification de circuits

Problèmes et solutions,

exemple de TestBuilder

Soutenu le 26 mars 2002, à Grenoble, devant le jury :

Mme Véronique DONZEAU-GOUGE	(CNAM Paris)
M. Christian CARREZ	(CNAM Paris)
M. Jean-Pierre GIRAUDIN	(CNAM Grenoble)
M. André PLISSON	(CNAM Grenoble)
M. Radu MATEESCU	(INRIA Rhône-Alpes)
M. Solofo RAMANGALAHY	(BULL & INRIA Rhône-Alpes)

Remerciements

Je tiens à remercier tout particulièrement,

– M. Hubert GARAVEL pour m'avoir présenté l'équipe VASY et la boîte à outils *CADP*.
– M. Radu MATEESCU pour la relecture de ce document.
– M. Solofo RAMANGALAHY pour m'avoir proposé ce sujet, pour ses précieux conseils et sa disponibilité.

Ce document a été réalisé sur un système Debian GNU/LINUX grâce aux logiciels EMACS (en mode AUCTEX) et LATEX.

Table des matières

1 Introduction **1**
1.1 Processus de conception d'un circuit complexe . . . 1
1.1.1 Utilisation des langages de description de matériel (HDL) . . . 1
1.1.2 Outils de CAO et outils de synthèse . . . 1
1.2 Qu'est ce que la vérification ? . . . 2
1.2.1 Une démarche qualité à trois niveaux . . . 2
1.2.2 La vérification fonctionelle . . . 2
1.2.3 Les tests structurels . . . 3
1.2.4 Le niveau applicatif . . . 3
1.3 Importance et coût de la vérification . . . 3
1.4 Les difficultés de la vérification . . . 4

2 Vérification : solutions **6**
2.1 Analyse statique du code . . . 6
2.2 Les outils de vérification formelle . . . 7
2.2.1 Le contrôle d'équivalence . . . 7
2.2.2 Le contrôle de modèle . . . 7
2.3 Les outils de simulation . . . 9
2.3.1 Les simulateurs . . . 9
2.3.2 Les visualiseurs . . . 10
2.3.3 Les co-simulateurs . . . 10
2.3.4 Les modèles de tierce partie . . . 10
2.3.5 Les modeleurs physiques et l'émulation . . . 11
2.3.6 Les outils de couverture de code . . . 11
2.4 L'écriture des bancs de tests . . . 12
2.4.1 Différents types de tests . . . 12
2.4.2 Les HDL et leurs limites . . . 12
2.4.3 Interaction avec la simulation depuis des langages traditionnels . . . 13
2.4.4 Modules pour HDL et HDL étendus . . . 13
2.4.5 Langages ou librairies spécialisés dans la vérification . . . 14
2.5 Une poussée de C++ dans les outils de CAO électronique . . . 17
2.6 Conclusion . . . 17

3 TestBuilder **18**
3.1 Présentation de la librairie . . . 18
3.2 La notion de transaction . . . 18
3.3 Avantages des transactions . . . 19
3.4 Mise en œuvre de techniques de tests avec les transactions . . . 19
3.5 Étude de la librairie . . . 20
3.5.1 Le choix de C++ . . . 20
3.5.2 Implémentation de concepts liés au matériel . . . 21
3.5.3 Implémentation des transactions . . . 22
3.5.4 Autres facilités . . . 24
3.6 Conclusion sur TestBuilder . . . 27

4 Conclusion **28**

Glossaire **29**

1 Introduction

L'AMÉLIORATION constante de la technologie des circuits intégrés conduit les industriels de l'électronique à développer des circuits de plus en plus complexes et performants. La densité d'intégration des circuits VLSI (Very Large Scale Integration) permet ainsi aujourd'hui de développer des « puces » généralistes (comme les processeurs), mais aussi des circuits spécialisés (fondamentaux dans les télécommunications, le traitement de l'image, etc.) composés de plusieurs millions de transistors.

Afin d'obtenir, des circuits intégrés qui fonctionnent parfaitement, malgré une telle complexité, il est nécessaire de détecter les erreurs le plus tôt possible dans la chaîne de conception, bien avant l'étape très coûteuse de fabrication. Le problème est donc de pouvoir *vérifier* au mieux un circuit avant même son existence physique réelle.

L'étude de ce problème est l'objet de ce document. Pour cela, nous aborderons rapidement le processus classique de conception d'un circuit complexe afin de présenter le contexte. Nous essaierons alors de définir plus précisément *ce qu'est vraiment* la vérification avant de dresser un panorama des techniques et outils existants.

Nous présenterons ensuite une solution particulière : *TestBuilder* de la société CADENCE®.

La conclusion sera l'occasion de récapituler l'état de l'art de la vérification mais aussi d'envisager les perspectives d'évolution à court terme.

1.1 Processus de conception d'un circuit complexe

Présentons brièvement le contexte. Un bref glossaire est par ailleurs disponible en page 29. Nous utiliserons le plus possible de termes français mais nous donnerons systématiquement l'équivalent en anglais.

1.1.1 Utilisation des langages de description de matériel (HDL)

Après l'écriture d'un cahier des charges et des spécifications, la conception d'un circuit débute par une description, au niveau algorithmique puis RTL[1] de la fonctionnalité du système en utilisant un langage de description de matériel (HDL, Hardware Description Language). Les deux langages les plus courants sont *Verilog®* et *VHDL*. Ils sont aujourd'hui standardisés [Hou00].

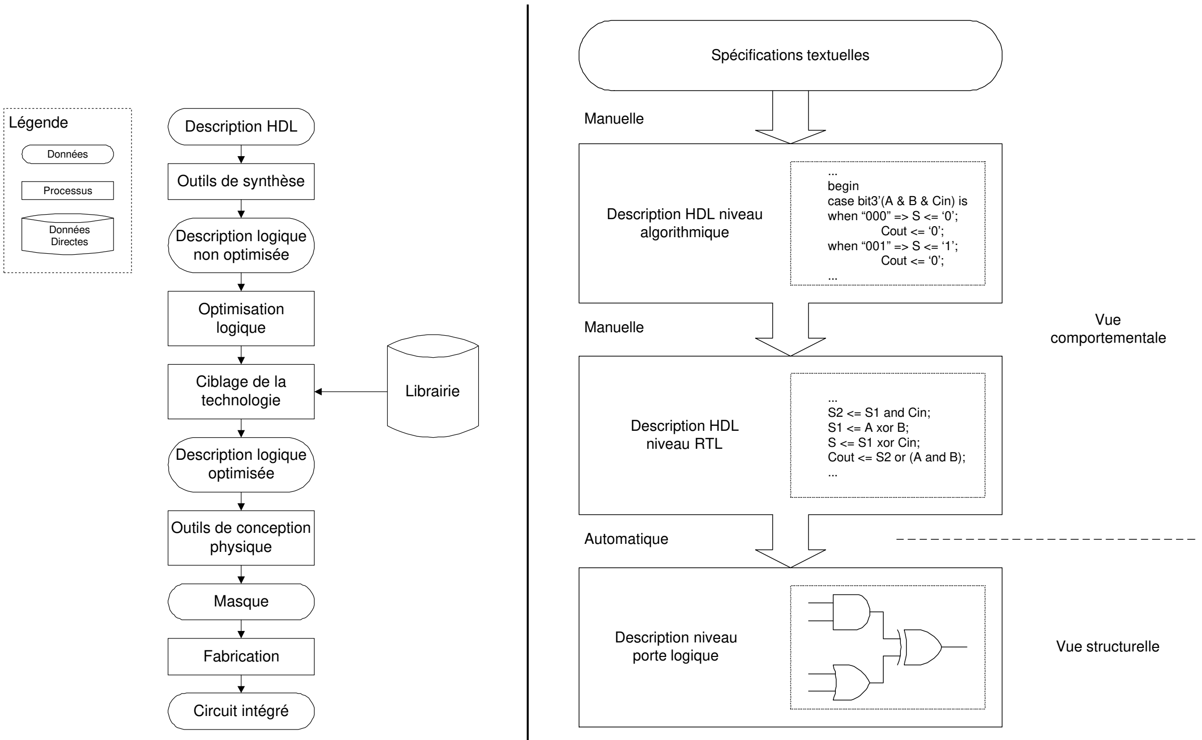
- ☞ **VHDL** : acronyme de VHSIC Hardware Description Language, il a été développé dans le cadre du projet VHSIC (Very high speed integrated circuits) commandité par le département de la défense américaine. C'est un standard IEEE depuis 1987 largement utilisé en Europe.
- ☞ **Verilog** a été développé comme un langage propriétaire, puis rendu public à la fin des années 80. Il est plus concis que VHDL, mais aussi plus limité. Verilog reste le standard au États-Unis [Pel].

1.1.2 Outils de CAO et outils de synthèse

Ensuite, les outils de *synthèse* sont utilisés pour transformer la description HDL de niveau *RTL* en une description au niveau logique (le schéma logique obtenu est appelé une « netlist »), puis au niveau transistors et ils produisent finalement le masque (niveau layout) qui est employé pour fabriquer le circuit. La figure 1.1a en page 2 illustre le processus de conception typique pour un circuit complexe [Pao01].
La densité d'intégration actuelle permettant de créer des circuits de plusieurs millions de transistors, il est bien évident que l'utilisation des outils de CAO (Conception Assistée par Ordinateur) est devenue indispensable.

Ces outils ont suivi l'évolution technologique en gagnant un niveau d'abstraction par décennie [Rou]. Ainsi, ont été commercialisés successivement des outils de dessins de masques (niveau *layout*), puis des outils de

[1]niveau transfert de registres (RTL pour register transfer level) qui donne une description *compilable* par un outil de synthèse logique. Il s'agit en fait du niveau de la logique synchrone ; le temps est divisé en intervalles appelés étapes de contrôle (control steps). A chaque étape de contrôle, on spécifie les conditions qui doivent être testées, tous les transferts entre registres qui doivent être exécutés et l'étape suivante [Rou].



a. Chaîne de conception　　　b. Vues comportementales et structurelles

FIG. 1.1 – Processus de conception

synthèse logique (niveau porte logique) et, enfin, actuellement, des outils de synthèse travaillant au niveau RTL[1].

Le concepteur travaille aujourd'hui essentiellement au niveau *comportemental* (Fig. 1.1b). Il intervient encore parfois au niveau *structurel*, essentiellement pour des raisons d'optimisation.

1.2 Qu'est ce que la vérification ?

1.2.1 Une démarche qualité à trois niveaux

On peut intuitivement déterminer trois phases, pendant lesquelles la qualité d'un circuit peur être appréhendée : la phase de *conception*, la phase de *fabrication* et la phase d'*utilisation*.
H. TROY définit respectivement les niveaux *fonctionnel*, *structurel* et *applicatif* [Cal93].

1.2.2 La vérification fonctionelle

La *vérification* est un terme que l'on réserve à la phase de **conception** en haut de la chaîne exprimée par le schéma 1.1.
Il s'agit de garantir, avant sa fabrication, que le composant répond bien aux spécifications désirées.

On distingue classiquement deux approches :
– les méthodes *formelles* permettant de vérifier qu'une description a le comportement attendu pour *toutes* les entrées possibles.
– les méthodes de *simulation* sélective vérifiant le comportement d'une description avec un ensemble de vecteurs de test, sous-ensemble de toutes les entrées possibles.

[1]On notera l'existence récente d'outils de synthèse comportementale (HLS en anglais pour High Level Synthesis). Des outils HLS commerciaux (*Behavioral Compiler* de SYNOPSYS® , *Visual Architect* de CADENCE® , *Monet* de MENTOR GRAPHICS) commencent d'ailleurs à apparaître et permettent la traduction d'une description de niveau algorithmique en une description de niveau RTL ou de niveau porte logique. Cependant, à l'instar des outils de synthèse logique, qui ont mis plus de 7 ans à s'imposer dans le monde industriel, les outils de synthèse comportementale ne semblent pas aujourd'hui exploités pleinement. Il faut dire qu'une connaissance approfondie des méthodes d'optimisation qu'ils intègrent et du style de descriptions de niveau algorithmique pouvant être pris en compte est encore nécessaire [Pao01].

La vérification formelle est souvent délicate à mettre en œuvre. Nous verrons cependant que deux méthodes particulières sont aujourd'hui de plus en plus utilisées : le contrôle d'équivalence (Equivalence Checking) et le contrôle de modèle (Model Checking).

La vérification basée sur la simulation reste l'approche la plus répandue. Le principe est d'exciter un modèle (souvent informatique) du futur circuit par une série de stimuli constituant le *banc de tests* (testbench). Les tests à ce niveau visent généralement à activer au maximum les possibilités du circuit afin de vérifier ses fonctionnalités (on parle parfois de vérification *fonctionnelle*).
La difficulté est alors d'obtenir un bon taux de *couverture* (c.-à-d. étudier un nombre suffisant de cas pertinents) en un temps raisonnable.
Nous verrons qu'il existe plusieurs moyens d'écrire des *bancs de tests* efficaces.

L'ensemble de ces techniques sera décrit dans le prochain chapitre.

1.2.3 Les tests structurels

Le contrôle des circuits en **fabrication** appartient à une phase que l'on appelle généralement la *phase de test* (testing) [Ber00]. Cette étape ne fait donc pas à proprement parler partie de la *vérification* mais il faut savoir qu'il convient de prévoir les tests de fabrication dès la conception du circuit.

Il n'est pas rare, même après avoir abouti à une validation fonctionnelle, d'obtenir des circuits qui ne sont pas corrects. Il y a de nombreuses causes possibles à ses défauts allant des impuretés aux défauts de lithographie. Cela se traduit par des structures défectueuses : un transistor à état fixe par exemple, ce qui conduit à un niveau logique bloqué appelé *collage*.
Les tests en fabrication consistent alors à activer, via des *vecteurs de tests* les constituants de base du circuit pour vérifier que chacun se comporte correctement et que les connexions sont correctes.

Il est possible d'obtenir de manière automatique les vecteurs de tests aptes à modifier tous les nœuds du circuit (méthode ATPG pour Automatic Test Pattern Generation) ; les vecteurs se déduisent en effet des équations logiques du circuit [Cal93]. Mais pour un circuit complexe (et en particulier séquentiel), le nombre de combinaisons possibles peut être astronomique.
L'objectif est donc de prévoir, dès la conception, un circuit qui soit relativement facile et rapide à tester en phase de fabrication. On utilise la notion de *testabilité* pour définir la facilité de génération des vecteurs de tests. La testabilité fait elle même appel aux notions de *contrôlabilité* et d'*observabilité* des nœuds du circuit.

- ☞ La **contrôlabilité** d'un nœud interne du circuit est la mesure de la facilité avec laquelle il est possible de fixer les états 0 et 1 pour ce nœud à partir des entrées.
- ☞ L'**observabilité** d'un nœud est la mesure de la facilité avec laquelle il est possible de connaître l'état de ce nœud à partir de la seule observation de la sortie.

Afin d'améliorer la testabilité, il est possible d'ajouter des *points de tests* qui correspondent à une logique supplémentaire pour les nœuds difficiles à observer et/ou à contrôler.
Mais on peut également utiliser des *techniques systématiques* durant la conception. L'idée essentielle de ces techniques vise à connecter tous les bistables internes selon un registre à décalage (scan chain), ce qui permet la contrôlabilité et l'observabilité de *tous* les nœuds séquentiels internes. Ces techniques sont aussi appropriées pour la génération automatique de solutions testables.

1.2.4 Le niveau applicatif

Le niveau **applicatif**, lui, consiste simplement à insérer le composant créé dans le système global et à vérifier s'il fonctionne correctement. Ce type de test est bien sûr le plus simple à générer mais il suppose que le circuit soit fabriqué et offre l'assurance la plus faible que le composant *complet* est fonctionnellement correct. Ce niveau est fréquemment utilisé par les clients. Pour le constructeur du circuit, les tests à ce niveau devraient simplement permettre de *confirmer* le bon fonctionnement du circuit. Des erreurs détectées à ce stade remettent en cause toute la chaîne de conception du circuit et illustrent des lacunes importantes dans les phases de vérification précédentes.

1.3 Importance et coût de la vérification

Afin d'obtenir un circuit intégré qui fonctionne conformément aux spécifications, il est nécessaire que la description HDL soit correcte et qu'aucune erreur ne soit introduite pendant les transformations aux niveaux inférieurs. Bien sûr, il est souhaitable de détecter les erreurs de conception au plus tôt. Il est beaucoup moins onéreux de détecter un problème lors de la conception de la description HDL qu'au niveau porte logique. De

même, il est beaucoup moins onéreux de déceler une erreur au niveau porte logique qu'après la fabrication du circuit. La détection de problèmes en phase de fabrication ou d'utilisation coûte extrêmement cher puisqu'elle nécessite la fabrication de nouveaux masques et occasionne des retards importants.
Il est donc nécessaire de pouvoir *vérifier* le circuit dès les toutes premières phases de conception.

On estime aujourd'hui que la *vérification* coûte jusqu'à 70% de l'effort de conception. Mieux, le nombre d'ingénieurs de vérification est parfois près du double de celui des concepteurs. Lorsqu'un projet est terminé, le code destiné aux tests occupe près de 80 % du volume total de code. « Le problème réel n'est pas tellement de créer un circuit intégré comportant 12 millions de portes et fonctionnant à 600Mhz, mais plutôt de savoir comment le *vérifier*. » [Ber00].

Il s'agit cependant d'une phase indispensable : il faut savoir que le coût de la réalisation d'un nouveau jeu de circuits prototype est d'environ 100 000 $ et la durée de plusieurs mois (dus à la nécessité de refabriquer le jeu de masques et de re-parcourir l'ensemble de la chaîne de production) [Anc97].

Ajoutons à cela que les circuits complexes ont de plus en plus de *responsabilité* dans les systèmes.
Ils sont aujourd'hui omniprésents, y compris dans des domaines où les conséquences d'une erreur peuvent être désastreuses (secteur médical, armement, transports, etc.).

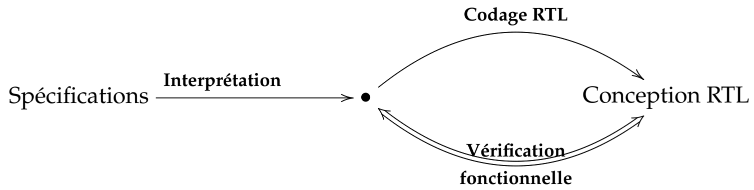
Les conséquences financières, enfin, peuvent être catastrophiques pour une entreprise. On peut bien sûr citer le *fameux* « bug de calcul[1] » du Pentium® en 1994 qui a été très médiatisé [Mar94]. INTEL® a du rappeler les Pentium vendus au public. La perte a été estimée à plus de 400 millions de dollars.

Mais il ne s'agit pas d'un cas isolé[2], de nombreux exemples d'erreurs de conception sont disponibles sur les sites des constructeurs. Plus de soixante *bugs* sont ainsi répertoriés par exemple pour le Pentium III et on en compte au moins 6 dans l'Athlon d'AMD [Geo01].
Pour un circuit donné, on compte parfois plusieurs versions (*révisions* ou stepping).

1.4 Les difficultés de la vérification

Le facteur humain

Les documents de spécification sont le plus souvent écrits en langage naturel dans le but de *communiquer* entre individus[3]. Tout document est donc soumis à interprétation.
Le facteur *humain* est déterminant dans l'interprétation des spécifications. Si une même personne vérifie, par exemple, le code RTL qu'elle a elle-même conçu, il est possible qu'elle fasse les mêmes erreurs d'interprétation lors de la vérification que lors de la programmation. Elle va comparer *sa* conception avec *son interprétation* des spécifications et non avec les spécifications de départ.
La figure ci-dessous, proposée dans [Ber00] (reconvergence model) met en évidence cette difficulté.



Afin de réduire les risques d'erreurs de ce type, il est possible de constituer des équipes qui contrôlent mutuellement leur travail. Il s'agit bien sûr d'une solution coûteuse. Notons que l'on procède à l'identique pour des sections de code critiques en développement logiciel.

Un problème complexe

Les densités d'intégration aujourd'hui possibles rendent la conception très complexe. Mais simuler le comportement d'un circuit même modeste est déjà un problème *fondamentalement* complexe.
Pour illustrer cela, reprenons l'anecdote proposée dans [Seg] : prenons un simple boîtier RAM de 256 bits ;

> « Supposons que l'on souhaite simuler de façon exhaustive toutes les transitions d'états possibles de ce circuit, combien de temps cela prendra-t-il ? Une RAM de 256 bits a approximativement 10^{80} combinaisons d'états et d'entrées possibles. Si nous faisons l'hypothèse (assez improbable) que nous pouvons utiliser toute

[1] L'algorithme de division fournit des résultats incorrects dans certaines conditions.

[2] Pour la petite histoire, ce rapport est essentiellement rédigé sur un portable équipé d'un Pentium 120 atteint du "F0 0F" bug : une séquence d'octets particulière suffit à bloquer le processeur alors qu'il devrait plutôt produire une exception de type *instruction invalide*.

[3] Le langage naturel est d'ailleurs souvent la seule façon de communiquer entre personnes de profils différents (informatique, électronique, physique, etc.) comme c'est le cas en micro-électronique.

la matière de l'univers pour fabriquer des ordinateurs ($10^{17}kg$), que chaque ordinateur est de la taille d'un électron ($10^{-30}kg$), que chaque ordinateur simule 10^{12} cas par seconde et que nous avons commencé la simulation au moment du Big Bang (il y a environ 10^{10} ans), nous aurions aujourd'hui atteint 0,05% de notre tâche.» [Seg]

Il est donc inutile de compter uniquement sur la *force brute* d'un ordinateur, même ultra puissant.

De plus, la complexité croissante des circuits n'est pas seulement dûe à la densité d'intégration. Des *algorithmes complexes*, à forte logique séquentielle, sont de plus en plus courants dans les puces dédiées à la compression et au traitement d'images par exemple.

Signalons la généralisation de principes d'*optimisation* comme les pipelines dans les processeurs par exemple, qui compliquent l'ordonnancement et rendent difficile le contrôle de l'exécution des séquences d'instructions.

Par ailleurs, une forte demande de *miniaturisation* (par exemple en téléphonie) pousse les industriels à créer des systèmes complets sur une seule puce (on parle alors de SoC pour System on a Chip). Ces circuits sont composés de plusieurs *blocs* devant *communiquer*, ce qui suppose des protocoles de communication qui doivent eux aussi être soumis à vérification. Cela suppose aussi des fonctionnements en parallèle qui deviennent très vite difficiles à appréhender.

Il faut enfin parfois contrôler des circuits qui *embarquent* du logiciel, ce qui pose le problème de la vérification *simultanée* du logiciel et du matériel, sachant que ce dernier n'existe pas encore physiquement !

Quand s'arrêter ?

Il est facile de montrer qu'une conception implante mal une fonctionnalité en mettant en évidence un cas de dysfonctionnement mais l'inverse n'est pas vrai. Il n'est pas possible de prouver par test qu'il n'y aura *jamais* de dysfonctionnement. La question *quand* doit-on arrêter les tests est donc délicate : ce n'est pas parce qu'aucune erreur n'a été détectée depuis plusieurs jours qu'il n'y a plus de risques. On peut, au contraire consacrer beaucoup d'énergie à chercher des erreurs dans une conception parfaitement saine. Nous verrons qu'il existe malgré tout des métriques permettant d'évaluer la couverture des tests effectués.

Les contraintes économiques

En même temps, les contraintes du marché poussent à réduire au maximum les temps de développement (time to market) afin de s'adapter aux cycles de vie de plus en plus courts des produits et de manière à être toujours les premiers sur le marché.

Nous avons mis en évidence dans cette partie, la nécessité de détecter au plus tôt les erreurs de conception lors de l'élaboration d'un circuit : c'est ce que l'on appelle couramment la *vérification*. Garantir la fiabilité d'un circuit le plus tôt possible lors de sa mise au point représente un enjeu économique important. C'est aussi devenu, avec la complexité grandissante des circuits, l'étape la plus problématique dans le processus de conception.
Nous allons maintenant décrire l'ensemble des techniques utilisées couramment dans le cadre de la vérification.

2 Vérification : solutions

NOUS ALLONS maintenant dresser un panorama des techniques et types d'outils les plus courants dans le domaine de la vérification de circuits.

Nous aborderons d'abord brièvement les outils d'analyse statique de code.
Puis nous verrons les solutions formelles, notamment à travers deux méthodes qui sont aujourd'hui sorties du domaine de la recherche pour intégrer les produits commerciaux : le contrôle d'équivalence (Equivalence Checking) et le contrôle de modèle (Model Checking).
Nous entrerons enfin dans le monde de la *simulation*, qui reste la solution la plus utilisée. Nous évoquerons les questions de *couverture* de vérification et nous verrons quelles sont les différentes solutions permettant d'ameliorer la programmation des bancs de tests.

Notre étude se limite aux solutions éprouvées et disponibles commercialement[1] , les solutions présentées dans le domaine actif de la recherche ne sont mentionnées que très rarement.

2.1 Analyse statique du code

Vérifier une conception, c'est déjà vérifier le code source de la description écrite en HDL afin d'éliminer le maximum d'erreurs.
Un moyen simple d'éliminer quelques erreurs est de faire relire le code produit par un concepteur par un de ses collègues. C'est aussi un bon moyen d'évaluer la « *maintenabilité* » d'un code.
Mais il existe aussi des outils qui permettent d'*automatiser* certains contrôles.

Étude préventive du code avec des outils de type Lint

Les outils de type LINT (Linting Tools) détectent un certain nombre d'erreurs classiques. On donne ce nom à cette catégorie d'outils par analogie au programme Unix LINT qui analyse un programme C et rapporte un certain nombre de problèmes potentiels et d'erreurs courantes[2]. Ces outils présentent l'avantage d'être *autonomes* : on n'a pas à écrire, au contraire de la simulation, de bancs de tests pour les utiliser. Mais en contrepartie, ils ne peuvent identifier que les problèmes déductibles d'une analyse statique du code. Ils ne peuvent en rien vérifier des problèmes d'algorithmique ou de flux de données.

- ☞ **Application à du code Verilog :** Verilog est un langage non typé : il est tout à fait possible d'assigner n'importe quelle valeur à un registre ou à un argument de sous-programme avec des conséquences qui peuvent évidemment être désastreuses. Les outils de type Lint permettent de détecter ce genre d'erreur très rapidement. Pour détecter de telles erreurs en simulation, au contraire, il faut prévoir de très nombreux cas sans même être sûr d'y parvenir.
- ☞ **Application à du code VHDL :** parce que VHDL est fortement typé, il nécessite moins l'utilisation d'outils de ce type. Malgré tout, cela peut être utile pour détecter d'autres types de problèmes ou simplement pour garantir une bonne conformité du code vis-à-vis de préconisations d'usage (coding guidelines).

On pourrait comparer ce genre d'outils à des correcteurs orthographiques : ils identifient les mots inexistants, éventuellement quelques fautes d'accord, mais ils ne peuvent reconnaître un terme inapproprié ou un problème de sémantique.

Pour donner quelques exemples de logiciels commerciaux, nous pouvons citer *SureLint*[3] de Verisity®, *HDLint*[4] qui travaille aussi bien avec Verilog que VHDL, ou encore *nLint* [5] de la société NOVAS[6].

[1] Nous citerons des exemples de produits pour illustrer le propos. La plupart des logiciels d'EDA (Electronic Design Automation) sont développés pour Unix. On note une percée récente de Linux. Seuls quelques outils fonctionnent sous Windows® NT. L'origine américaine de nombreuses compagnies d'EDA explique sans doute la plus forte disponibilité d'outils pour Verilog que pour VHDL. Le prix des licences, pour les logiciels importants, s'exprime assez souvent en dizaines de milliers de dollars.
[2] La qualité des compilateurs C actuels a rendu son utilisation caduque.
[3] http://verisity.com/products/surelint.html
[4] http://www.veritools-web.com/hdl.htm
[5] http://www.novas.com/products/prod-nLint.html
[6] http://www.novas.com

2.2 Les outils de vérification formelle

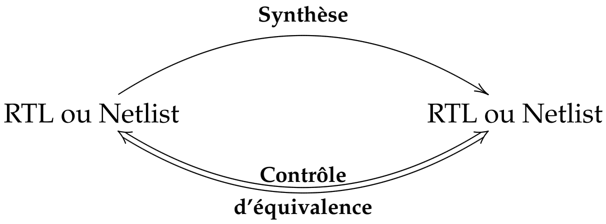
La vérification formelle consiste à *démontrer* qu'un circuit va bien se comporter comme il est souhaité. Au contraire de la simulation, cette démonstration se veut exhaustive par nature (c.-à-d. valable *pour toutes les entrées possibles*) et met en œuvre des techniques à forte « teneur mathématique ».

La démonstration automatique (Theorem Proving) a été l'une des premières approches formelles de vérification. Elle suppose que les spécifications et l'implémentation soient toutes deux décrites dans un formalisme permettant le raisonnement mathématique. La démonstration automatique est en principe capable de répondre à toutes les questions de vérification qui se posent en pratique, mais sa mise en œuvre est lourde et compliquée et les outils actuels sont seulement capables d'assister un ingénieur à qui revient la charge de construire la preuve [Sch99].

Si la démonstration automatique a été utilisée avec succès pour démontrer par exemple les algorithmes de calculs en virgule flottante des processeurs AMD, il s'agit d'une méthode qui n'est pas généralisée car trop complexe [Dil]. Par contre, deux méthodes particulières sont aujourd'hui assez largement utilisées : le contrôle d'équivalence et le contrôle de modèle.
Un aperçu de nombreux cas d'étude utilisant des méthodes formelles est disponible sur [CW].

2.2.1 Le contrôle d'équivalence

Le contrôle d'équivalence (ou Equivalence Checking) consiste à prouver *mathématiquement* l'équivalence de deux descriptions[1].



On utilise le plus souvent cette méthode pour comparer deux *netlists* et s'assurer que l'introduction d'une modification manuelle, une optimisation ou un traitement facilitant par exemple la détection des fautes en production (voir la section 1.2.3) n'ont pas altéré la description de départ.

Une autre utilisation de plus en plus courante est la vérification que la description logique correspond bien au code RTL original. Les outils de synthèse sont en effet des logiciels complexes pas toujours exempts de *bugs*. On ne peut leur accorder une confiance absolue.

Bien que très pratique, le contrôle d'équivalence a des limites : il permet de *montrer* l'équivalence de deux représentations mais il ne permet pas d'isoler les *bugs* fonctionnels. Deux descriptions peuvent être équivalentes mais incorrectes. De plus, les outils ne permettent encore de travailler que sur des portions de circuits.

Il n'y a pas encore énormément de logiciels commerciaux mais on peut citer les quatre logiciels suivants : *Conformal LEC* (VERPLEX SYSTEMS INC.[2]), *Design Verifyer* (AVANT ![3]), *Formality* (SYNOPSIS® [4]) et *FormalPro* (MENTOR GRAPHICS® [5]).

On trouvera un comparatif récent (novembre 2001) et détaillé de l'ensemble des logiciels commerciaux de contrôle d'équivalence sur [Phi01].

2.2.2 Le contrôle de modèle

Le contrôle de modèle (Model Checking) est une méthode assez récente qui s'applique en fait à une large classe de systèmes : tous ceux qui sont modélisables par un automate fini (ou une variante de cette représentation générale). Elle se veut exhaustive et en grande partie automatique. Au contraire de la démonstration automatique, le travail de l'ingénieur se limite à la construction d'un modèle formel du système et à la formalisation des propriétés à vérifier.

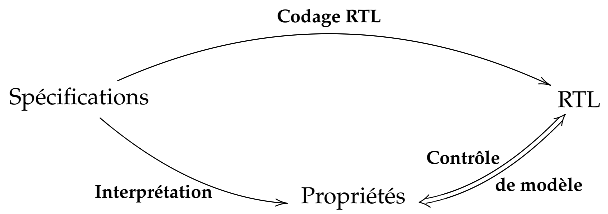
Fondamentalement, cette méthode repose sur trois éléments.

[1] Dans le schéma, il y a bien *convergence* des points extrêmes de la synthèse et la vérification.
[2] http://www.verplex.com
[3] http://www.avanticorp.com
[4] http://synopsys.com
[5] http://www.mentor.com

– Le modèle formel du système : il s'agit d'une représentation sous forme d'automate mais certains logiciels actuels permettent de travailler directement sur une représentation de niveau RTL.
– Des propriétés à vérifier : elles s'expriment le plus souvent dans une logique *temporelle*, qui outre les connecteurs booléens classiques propose des *opérateurs temporels* et des *quantificateurs de chemins*. Il est ainsi possible d'exprimer des formules du genre : *Il est possible d'avoir la propriété ϕ dans un état futur, on aura forcément ϕ dans un état futur, ...* [Sch99]
– Le model-checker qui implémente l'algorithme qui permettra de vérifier si le modèle satisfait ou non une propriété.

On peut par exemple, grâce au contrôle de modèle, vérifier qu'un automate ne contient pas d'états inaccessibles. Un algorithme plus puissant pourra déterminer si des conditions de blocage (deadlocks) sont susceptibles d'apparaître.
Pour prendre des exemples plus concrets, on pourra vérifier, pour un processeur, que seule l'instruction `trap` peut passer d'un mode *utilisateur* à un mode *superviseur* ou encore qu'une interruption sera traitée au maximum après t unités de temps après la demande d'interruption [Seg].

La plus grosse difficulté est de déterminer quelles propriétés choisir pour la démonstration. Comme le montre le schéma ci-dessous [Ber00], il n'y a pas de point de *convergence* entre les origines du codage RTL et de la vérification ; une mauvaise rédaction des propriétés rend la vérification inéxacte.



De plus, la technologie actuelle ne permet pas de travailler sur des propriétés de haut niveau pour montrer qu'une fonctionnalité complexe est correctement implémentée. Il y a donc un important travail pour formuler les bonnes propriétes à vérifier.

L'intérêt des model-checkers est de *prouver* des propriétés. Mais un de leurs points forts est d'être capable de fournir un *contre-exemple* lorsqu'une propriété n'est pas vérifiée. Ce contre-exemple est directement utilisable par les ingénieurs pour analyser le problème. Le contrôle de modèle peut donc être utilisé dans le but de cerner des erreurs. Là encore, c'est à l'ingénieur de vérification de définir les propriétés qui permettront la mise en évidence de telles erreurs par la production de contre-exemples.

D'un point de vue pratique, le principal obstacle que rencontrent les algorithmes de model-checking est le problème dit de l'« explosion du nombre d'états » (state-explosion problem).
Plutôt que de représenter de façon *explicite* l'ensemble des états et des transitions du modèle, on a par conséquent cherché à représenter ce dernier de façon *symbolique*. On parle alors de contrôle de modèle *symbolique*. Généralement, on utilise des *diagrammes de décisions binaires* (ou BDD pour Binary Decision Diagrams) pour représenter les ensembles d'états. Il s'agit d'une représentation canonique introduite par S.B. AKERS, puis R.E. BRYANT et J-P. BILLON [Anc97]. L'intérêt d'une telle représentation est que la taille des graphes peut être relativement réduite, même pour des fonctions complexes.

Exemples d'outils :

☞ ***SMV***[1] : *SMV* a été développé par K.L. MC MILLAN sous la direction de E.M. CLARKE à l'Université de Carnegie-Mellon. *SMV* est considéré comme le premier outil qui a pu, grâce aux BDD, vérifier de façon exhaustive que des systèmes de taille très élevée satisfaisaient leurs spécifications. Parmi les études de cas réalisés avec *SMV*, on peut noter divers protocoles de cohérence de cache (Gigamax, Futurebus+), des circuits logiques ou des automates industriels.

☞ ***Kronos***[2] : Développé au laboratoire VERIMAG de Grenoble, il permet l'analyse d'*automates temporisés*. Les *automates temporisés* ont été proposés par ALUR et DILL pour représenter des informations quantitatives sur l'écoulement du temps. *Kronos* a permis de vérifier, entre autres, de nombreux protocoles de communication temporisés (CSMA/CD, Audio-Control de Philips, ...).

☞ ***CADP***[3] : Il s'agit d'une boîte à outils developpée par l'équipe VASY de l'INRIA en collaboration avec le laboratoire VERIMAG. Elle est spécialisée dans l'ingénierie des protocoles et des systèmes distribués.

[1] http://www.cs.cmu.edu/~modelcheck/smv.html
[2] http://www-verimag.imag.fr/TEMPORISE/kronos/
[3] http://www.inrialpes.fr/vasy/cadp

La modélisation se fait via le langage LOTOS[1]. Parmi les toutes dernières vérifications, on note par exemple la détection d'un problème dans le protocole FireWire[2].

Les outils commerciaux[3] sont relativement récents mais on compte aujourd'hui une offre de près d'une dizaine de produits[1] parmi lesquels : *Verifier*, *BlackTie*, *Design Verity-Check*, *FormalCheck™*, *Formal Model Checker*, *imPROVE-DL*, *Solidify* ou *Verix*.
Là encore, on pourra se référer à [Phi01] pour un comparatif récent des outils commerciaux.
Les sites [Eet] et les listes de dicussions abritées sur [Ber] et [Dee] constituent aussi de bonnes sources d'informations et d'échanges d'expériences.

2.3 Les outils de simulation

2.3.1 Les simulateurs

Les *simulateurs* constituent encore l'outil de vérification le plus utilisé.
Ils permettent aux concepteurs d'*interagir* avec un circuit avant que ce dernier ne soit construit. On utilise l'acronyme DUV (design under verification) ou DUT (design under test) pour désigner le circuit, tel qu'il existe dans l'univers artificiel créé par le simulateur.
Un des avantages des HDL est qu'ils permettent d'écrire des descriptions de circuits qu'il est directement possible de simuler.
Bien sûr, un simulateur ne fait qu'*approcher* la réalité et certaines caractéristiques physiques sont simplifiées[2].

Au contraire des outils d'analyse de code vus précédemment (section 2.1), les simulateurs ne sont pas des outils statiques. Ils demandent au contraire à l'utilisateur d'intéragir en reproduisant l'environnement auquel le circuit est destiné. C'est finalement la description de cet environnement que l'on appelle *banc de tests* ou testbench. L'ingénieur de test va ainsi décrire des stimuli en entrée du DUV et observer les sorties. C'est lui qui déterminera alors si le circuit paraît correct : ce n'est en aucun cas le simulateur qui *valide* une conception. Nous consacrerons la section 2.4 à l'écriture des bancs de tests.

Le problème majeur de la simulation est celui de la vitesse. Les simulateurs sont censés reproduire un monde où des millions de transistors peuvent changer d'état un milliard de fois par seconde alors qu'ils sont exécutés sur des machines au mieux capables de traiter quelques centaines de millions d'instructions par seconde [Ber00].
Une façon de contourner ce problème est de ne simuler *que* ce qui est *utile*. Ainsi, si un courant continu est appliqué en entrée d'une porte logique dans le monde réel, la seule chose qui nous intéresse en fait est un *changement* au niveau des entrées.
La simulation étant alors conditionnée par les changements d'états des entrées, on parle de simulateur événementiel (event driven simulator) pour insister sur le fait qu'on s'intéresse aux *événements*.

Mais de nombreux événements intermédiaires, dans les circuits synchrones, ne sont pas pertinents du point de vue de l'étude fonctionnelle : un élément du circuit peut nécessiter plusieurs cycles pour son fonctionnement interne mais ces événements n'ont aucun intérêt au niveau du système à simuler.
On utilise alors des simulateurs basés sur le cycle d'une horloge (cycle based simulators) pour lesquels toute logique combinatoire est traitée sous forme d'équations qui sont recalculées lorsqu'un signal d'horloge (clock) est déclenché. On gagne évidemment beaucoup en vitesse de simulation mais on perd la notion de temps puisqu'on néglige la propagation des événements intermédiaires. Ce type de simulation suppose que l'horloge soit le seul événement significatif dans le changement d'état du circuit. Cela ne pose pas de problème pour les circuits synchrones.

Certains simulateurs abritent les deux modèles : un premier simulateur s'occupe des parties synchrones alors que l'autre est de type événementiel. Les deux tournent en parallèle et leur coopération permet de simuler l'ensemble du circuit. Il est déjà question ici d'une forme de co-simulation (voir la section 2.3.3).

Malgré tous ces efforts, il ne faut pas perdre de vue que la simulation reste un processus fondamentalement lent. Un modèle de processeur complexe ne peut guère être simulé à plus de quelques Hertz. Il faut déployer

[1] Language Of Temporal Ordering Specification – Langage standard OSI de description formelle de protocoles et de systèmes distribués.

[2] bus série très haut débit.

[3] Quelques sociétés importantes ont développé leurs propres logiciels en interne, il peut s'agir d'un avantage concurrentiel et ces logiciels ne sont donc pas vendus à l'extérieur.

[1] Certains ont d'ailleurs une origine universitaire. *FormalCheck™* est par exemple issue de Cadence-SMV qui est lui-même une évolution de SMV. Il est vendu 95 000$.

[2] Pour un simulateur numérique par exemple, les seules valeurs possibles d'un signal sont : '0', '1', 'inconnu' et 'haute impédance'. Le monde physique est plutôt « analogique » et un signal peut prendre des valeurs continues.

des moyens colossaux (grappes de serveurs et beaucoup de temps de simulation) pour obtenir l'équivalent de quelques secondes de fonctionnement d'un processeur à 1Ghz [Ben].

On trouve de très nombreux simulateurs sur le marché. On peut citer, par exemple pour les simulateurs Verilog : *Verilog-XL*[7] qui est un standard de fait , *VCS*™ [8] qui a la réputation d'être parmi les plus rapides (il procède par compilation), *ModelSim*[9] qui est peut-être le plus répandu grâce à son prix et son interface graphique pratique. Mentionnons enfin *Icarus Verilog*[10] et *VBS*[11] qui sont des produits libres (licence GPL[12]).

2.3.2 Les visualiseurs

Les visualiseurs (*Waveform Viewers*) affichent les résultats des simulations sous une forme graphique familière. Ils permettent de visualiser les transitions de nombreux signaux au cours du temps ainsi que les relations entre les transitions. Il est généralement possible de « zoomer » sur certaines séquences, de mesurer des différences de temps ainsi que d'afficher des séries de signaux sous une forme plus facile à appréhender (chaînes de caractères, hexadécimal ou autres. . .).

Ces outils sont indispensables pour inspecter le résultat d'une simulation et analyser un problème. Ce sont en quelque sorte les *débogueurs* de la simulation. On peut les utiliser *pendant* la simulation mais aussi, et surtout, *à posteriori* où ils utilisent les informations qui ont été sauvegardées au cours de la simulation[13].
Il est alors possible de *naviguer* rapidement dans une simulation qui a pu prendre des heures.
Les visualiseurs les plus performants permettent de comparer deux ensembles de courbes, l'un étant considéré comme une référence. Les différences sont mises en évidence, ce qui facilite la recherche car il ne faut pas oublier que le résultat d'une simulation peut être très complexe (nombreux signaux évoluant dans le temps).

Quelques exemples d'outils : *SignalScan*[14] de CADENCE, *GigaWave Viewer*[15] de SYNOPSYS mais encore *nWave*[16] de NOVAS ou le logiciel libre *gtkwave*[17].

2.3.3 Les co-simulateurs

On parle de *co-simulation* lorsque deux (ou plus) simulateurs coopèrent pour simuler un circuit. Nous avons vu que des simulateurs complexes pouvaient associer les modèles *event-based* et *cycle-based*. Mais d'autres formes de co-simulation existent. On peut ainsi *mélanger* simulation analogique et digitale, VHDL et Verilog, etc.
Une nouvelle forme de co-simulation permet de travailler sur des circuits avec logiciel embarqué. Dans [NVHPJ], on trouve par exemple la description d'un co-simulateur C-VHDL et notamment de la partie la plus sensible, à savoir la *synchronisation* des deux parties (simulateur VHDL et simulateur logiciel). Pendant une co-simulation, tous les simulateurs doivent en effet progresser selon un temps unique (ce qui suppose un alignement sur le plus lent).
Comme exemple de produit commercial, nous citerons *Seamless*[18] de MENTOR GRAPHICS pour le développement de circuit avec logiciel embarqué.

2.3.4 Les modèles de tierce partie

Pour obtenir un environnement de simulation complet, il peut être nécessaire de disposer des descriptions des circuits avec lequel le DUV va devoir dialoguer (par exemple des mémoires). Si ces circuits sont fournis par un constructeur externe, il est tout à fait possible qu'il puisse également en fournir des *modèles* utilisables dans les simulateurs. Mais il existe aussi des compagnies qui fournissent des modèles standards de mémoires et de processeurs notamment, fabriqués par d'autres firmes. Pour des raisons de licences et de propriété intellectuelle, les modèles sont le plus souvent livrés sous forme *compilée* plutôt que sous forme de code source Verilog ou VHDL.

[7] http://www.pcb.cadence.com/pcb/datasheets/cae/verilog-vl-family.asp
[8] http://synopsys.com/products/simulation/vcs_ds.html
[9] http://www.model.com/products/
[10] http://www.icarus.com/eda/verilog
[11] http://www.flex.com/~jching
[12] La GPL est la licence publique GNU. Voir http://gnu.org pour tout savoir sur le projet GNU.
[13] généralement sous forme de fichier au format VCD (Value-change-Dump)
[14] http://www.cadence.com/datashetts/affirma_sig_wav_view.html
[15] http://www.syncad.com/gwv_main
[16] http://www.novas.com/products/index.html
[17] http://www.cs.man.ac.uk/amulet/tools/gtkwave/
[18] http://mentor.com/seamless/products.html

2.3.5 Les modeleurs physiques et l'émulation

Parfois, un circuit peut être tellement complexe qu'aucun fournisseur, du moins pendant un temps, n'est en mesure d'en proposer un modèle. Ainsi, à la sortie d'un microprocesseur, il se peut qu'aucun modèle ne soit disponible. Ce fut par exemple le cas avec le Pentium III [Ber00]. Or pour valider une carte mère par exemple avant sa fabrication, il faut bien trouver un moyen de simuler le processeur.

Les modeleurs physiques (hardware modelers) offrent une solution à de telles situations. Ils s'agit en fait de dispositifs physiques capables d'accueillir le circuit *réel* qui doit rentrer dans le cadre de la simulation. Ils communiquent avec le simulateur via une interface particulière, ce qui leur permet d'alimenter le circuit avec les données du simulateur, et en retour, de retourner les sorties du circuit vers le simulateur.

En bref, un modeleur physique permet de *brancher* un circuit physique réel dans un simulateur.
L'utilisation de tels dispositifs n'est jamais une tâche triviale. Il faut notamment tenir compte des problèmes de décalage de temps (timing) entre la simulation et le monde matériel.
Mais ils sont également parfois utilisés pour des raisons de performance. Simuler de façon traditionnelle un circuit très complexe peut s'avérer en effet assez laborieux, comme on l'a vu en section 2.3.1.

Précisément parce qu'intégrer du matériel dans une simulation permet de gagner du temps de simulation, il peut être intéressant de réaliser *physiquement* un morceau du circuit pour soulager en partie le simulateur. C'est ce qu'on appelle l'*émulation*. On travaille généralement avec des circuits programmables de type FPGA.

2.3.6 Les outils de couverture de code

La *couverture de code* (code coverage) est une méthode qui est aussi utilisée dans le développement logiciel, et qui vise à répondre à la question : y'a t-il une portion de code qui n'a pas été exécutée ? Tout code doit être considéré comme susceptible de contenir des erreurs à priori.
Le code source représentant la description HDL est d'abord *instrumenté*. Cette phase consiste à ajouter des points de vérification à des endroits stratégiques du code source qui permettront de savoir si la simulation est *passée* ou non sur une structure de code particulière.
Le code instrumenté est ensuite simulé de façon traditionnelle, en utilisant les bancs de tests classiques. L'ensemble des *traces* de la simulation est enregistré dans une base de données à partir de laquelle on pourra établir un certain nombre de métriques de couverture et connaître le détail du code qui a été simulé.
Les rapports les plus classiques sont les couvertures d'instructions, de chemins et d'expressions.

- **Couverture d'instructions** (Statement Coverage) : ce type de rapport mesure le nombre de lignes qui ont été effectivement exécutées. Généralement, une interface graphique permet de naviguer dans le code et d'identifier rapidement les blocs qui n'ont pas été simulés. Il est nécessaire de comprendre quelles sont les conditions utiles pour que ces lignes soient exécutées et le cas échéant, ajouter un test pour prévoir ce cas de figure.
- **Couverture de chemins** (Path Coverage) : la couverture de chemins explore toutes les voies possibles que peut prendre un programme. Ainsi, pour un code qui contient deux instructions `if` par exemple, on compte déjà quatre chemins possibles (vrai-vrai, vrai-faux, faux-vrai, faux-faux).
- **Couverture d'expressions** (Expression Coverage) : la couverture d'expressions va encore plus loin et analyse le détail des expressions pour couvrir l'ensemble des possibilités.
  Si on simule le code `if (a==0 || a==1)` avec la valeur 0 pour `a`, on n'obtiendra par exemple qu'une couverture de 50%.

Notons que les outils les plus puissants sont capables d'extraire d'une description RTL la définition d'un automate (FSM pour Finite State Machine) et de tester la couverture en fonction des états et transitions explorés pendant la simulation. Là encore, l'interface graphique permettra de mettre en évidence la couverture de façon pratique.

Mais que signifie finalement une couverture de 100% ? Cela montre que la vérification est passée sur l'ensemble du code source mais ne donne finalement aucune indication sur la validité des tests. Les outils de couverture de code peuvent être utilisés avec profit pour identifier les cas particuliers difficiles à détecter (corner cases) auquels on n'aurait pas forcément pensé lors de l'écriture des tests. Ils donnent un bon *indice de confiance* sur l'étendue des tests réalisés et les métriques qu'ils fournissent sont exploitées pour mesurer la *progression* de la phase de déboguage. Mais il ne faut en aucun cas considérer qu'une couverture de 100% garantit une conception forcément exempte d'erreurs.

Il existe de nombreux outils de couverture. La plupart du temps, les éditeurs vendent d'ailleurs des *suites* complètes et plus ou moins intégrées de synthèse et/ou de vérification. On peut par exemple citer *Covermeter*[1] qui fait partie de la suite de vérification de Synopsys intégrant, entre autre, le simulateur VCS et le

[1] http://www.synopsys.com/products/simulation/covermeter_ds.html

langage VERA (voir section 2.4.5). Citons aussi *Verification Navigator*®[1] qui se veut pratique à utiliser ou encore *SureCov*[2] et son interface graphique *SureSight*.

2.4 L'écriture des bancs de tests

Nous avons vu qu'une simulation n'avait de sens que si on arrivait à recréer, grâce aux *bancs de tests*, le contexte du circuit (à travers des stimuli) afin d'observer son comportement. Nous étudions dans cette section les méthodes et outils permettant de programmer efficacement des bancs de tests. Il nous faut tout d'abord introduire quelques notions générales importantes.

2.4.1 Différents types de tests

Niveaux de visibilité

Lors de la simulation et de la préparation des bancs de tests, on peut tenir compte ou non de l'implémentation. On distingue ainsi classiquement deux approches :

- ☞ **Boîte noire** : dans l'approche *boîte noire*, la vérification se fait sans aucune connaissance de l'implémentation. Toutes les vérifications se font à travers les interfaces qui sont les seuls éléments connus. Une vérification de type *boîte noire* constitue un test de *conformité*. On vérifie qu'un circuit répond bien aux spécifications, indépendamment de la façon dont il est implémenté. Cette indépendance vis-à-vis de l'implémentation constitue un avantage mais cette méthode souffre d'un manque de visibilité.

  Notons que c'est la seule possible lorsque la vérification fonctionnelle doit être conçue *en même temps* que l'implémentation du circuit lui-même.
- ☞ **Boîte blanche** : à l'inverse, la méthode *boîte blanche* offre une pleine visibilité et un contrôle total de la structure interne et de l'implémentation du circuit à vérifier. Cela est particulièrement pratique pour tester *rapidement* un type de combinaisons d'entrées où isoler une fonction particulière. On peut tester des cas que l'on sait délicats (puisque l'on connait l'implémentation).

Les tests de type *boîte blanche* sont plus efficaces puisqu'il permettent au développeur de gagner des cycles de simulation en tenant compte de l'implémentation du DUV. Mais les tests sont parfois moins *réalistes* puisqu'ils ne se contentent pas d'utiliser les interfaces. De plus, la dépendance vis-à-vis-de l'implémentation oblige à une réécriture des tests en cas de modification du DUV.

Tests deterministes et tests aléatoires

Les tests peuvent être écrits *manuellement* suivant des scénarios précis et complètement déterminés mais il est souvent plus pratique de lancer des stimuli *aléatoires*. Bien sûr, cela suppose que l'on puisse automatiquement *comparer* le résultat de la simulation avec le résultat attendu.
Pour que cette méthode soit efficace, il faut disposer, d'autre part, de moyens de contrôler les tirages aléatoires en fonction des *contraintes* du circuit [Ore00].

Les tests déterministes sont utilisés pour contrôler des configurations bien précises alors que les tests purement aléatoires permettent de tester toutes sortes de situations, ce qui peut être utile pour découvrir des cas *limites*. Les tests aléatoires avec contraintes présentent l'avantage de permettre une forte *automatisation* des tests (en évitant au programmeur de décliner l'ensemble des combinaisons à tester) tout en respectant un contexte réaliste.
Nous aurons l'occasion de voir que les outils spécialisés dans la vérification permettent de travailler efficacement sur ce type de tests.

Un banc de tests complet utilise généralement l'ensemble de ces approches.

2.4.2 Les HDL et leurs limites

Les bancs de tests peuvent et sont en partie écrits directement dans le langage de description du circuit. C'est même assez pratique de n'avoir à utiliser qu'un seul langage. On utilise alors les primitives d'entrées/sorties du langage. Par exemple *$display* (qui provoque un affichage à l'instant où il est exécuté) ou *$monitor* (qui déclenche un processus automatique qui va rafraîchir l'affichage chaque fois qu'un des signaux spécifiés change d'état) en Verilog. Les HDL intègrent la mécanique nécessaire à l'utilisation de fichiers. En VHDL, on dispose même d'un système d'*assertions* :

[1] http://www.transeda.com/products/vn.html
[2] http:www.verisity.com/products/surecov.html

```
assert condition
   report message
      severity level
```

Toutefois, ces langages ont été conçus avant tout pour décrire du matériel et peuvent paraître assez vite limités pour l'écriture de bancs de tests complexes. Verilog, par exemple, a été conçu avec comme objectif la possibilité de décrire des structures matérielles de bas niveau. Il ne supporte pas les structures de données complexes, ni l'orientation objet maintenant courante dans les langages de programmation.
Il existe bien des primitives de type `$random`, mais les limitations des langages ne facilitent pas la programmation des bancs de tests aléatoires avec contraintes. Les tests sont d'un niveau d'abstraction assez faible, ce qui nuit à la réutilisation des programmes.
Le manque de concision des HDL (en ce qui concerne l'écriture des tests) conduit à des bancs de tests représentant jusqu'à 80% du code d'un projet [Ber00].
De plus, si la simulation lance en parallèle de nombreux processus pour reproduire la réalité physique du circuit, il est difficile de contrôler ces processus par programmation.
En bref, bien que les HDL puissent être utilisés pour l'écriture des bancs de tests, il n'ont pas été conçus pour cela et ne sont donc pas particulièrement adaptés à cette tâche.
Devant ce problème, plusieurs pistes on été explorées.
- D'abord, des mécanismes (en fait des API ou Interfaces de programmation) ont été developpés afin de pouvoir interagir avec la simulation depuis des langages traditionnels.
- D'autres initiatives visent à étendre les HDL, soit à travers des modules standards, écrits en HDL, soit en créant un sur-ensemble des HDL originaux.
- Des langages spécialisés dans la vérification (HVL pour Hardware Verification Languages) ont par ailleurs vu le jour.
- Des *librairies* ont enfin été définies pour rendre des langages traditionnels aptes à travailler sur des questions de description électronique et de bancs de tests matériels.

2.4.3 Interaction avec la simulation depuis des langages traditionnels

Il est possible de mélanger les langages HDL avec des langages de programmation traditionnels grâce aux API des HDL. Le PLI (Programming Language Interface) est un ensemble de routines d'interface C qui permettent d'accéder à une simulation Verilog[1]. Cela permet de lire ou écrire des données d'un module Verilog, de récupérer des informations sur l'environnement de simulation, mais aussi de lancer des fonctions externes (écrites en C) depuis la simulation [Cla]. Un mécanisme similaire existe pour VHDL à travers VHPI.

Grâce à ces mécanismes, on peut écrire des bancs de tests en C par exemple, mais aussi en Perl ou en TCL. En contrepartie, il est nécessaire pour le programmeur de connaître les fonctionnalités proposées par les API. De plus, il faut *lier* le code externe et le code HDL suivant des procédures plus ou moins contraignantes.

2.4.4 Modules pour HDL et HDL étendus

Deux initiatives méritent aujourd'hui d'etre signalées :

- ☞ ***OVL***[2] (Open Verification Library) est une librairie open-source supportée par VERPLEX SYSTEMS INC. Plus précisément, il s'agit d'une bibliothèque de modules Verilog permettant de spécifier des propriétés du modèle devant être vérifié. Cette librairie permet de travailler avec des assertions définies d'une façon qui se veut unique quelque soit l'outil utilisé. Une version VHDL est en cours. Plusieurs compagnies (VERPLEX, AXIS SYSTEMS, FORTE, etc.) ont annoncé des produits utilisant *OVL*.
- ☞ ***Superlog***[3] de CO-DESIGN AUTOMATION INC[4] peut être vu comme un sur-ensemble de Verilog. Superlog n'utilise pas les mécanismes PLI pour dialoguer avec un simulateur traditionnel. Il nécessite au contraire un simulateur particulier (produit par CO-DESIGN AUTOMATION) qui comprend Verilog mais aussi les extensions du langage. Pour CO-DESIGN AUTOMATION, c'est un gage de rapidité.

  Superlog ajoute à Verilog de nombreuses caractéristiques : les types définis par l'utilisateur, les types énumérés, les structures de données complexes, les pointeurs, la récursivité, les notions de pile et de tas, ou encore les chaînes de caractères. Il est possible de gérer des processus dynamiquement, de travailler avec des tableaux dynamiques ou encore de représenter facilement des automates. La syntaxe est un mélange de Verilog et de C . Il n'oblige pas à un changement de méthodologie trop important.

[1] Avec la version 2.0 de PLI, un nouveau type de routines (rassemblées sous le nom VPI (pour Verilog Procedural Interface) a été ajouté afin de faciliter encore une interaction dynamique avec la simulation.
[2] http://verificationlib.org
[3] http://www.superlog.org/index.htm
[4] http://www.co-design.com

N'oublions pas, enfin, que les langages HDL évoluent eux aussi. La norme Verilog 2001 vient par exemple d'être définie et ajoute quelques aspects intéressants (fonctions récursives, amélioration des fonctions aléatoires ou des entrées-sorties, etc.) notament pour l'écriture des bancs de tests.

2.4.5 Langages ou librairies spécialisés dans la vérification

HVL (Hardware Verification Languages) : des langages spécialisés

Il s'agit ici de langages spécialisés (domain specific languages) qui nécessitent un apprentissage particulier complètement lié à l'outil et à la méthodologie préconisée par l'industriel qui le commercialise. Le langage fait partie généralement d'une suite logicielle complète, qui contient d'autres outils de vérification (outils de couverture, visualiseurs, ...). On parle de testbench authoring tools.

Ces langages ont une syntaxe qui se veut claire et relativement simple. Ils dialoguent avec les simulateurs via les mécanismes d'API mais de façon complètement transparente pour l'utilisateur, qui n'a pas à ce soucier de cet aspect. Ils permettent une approche de *haut niveau*, ce qui permet d'augmenter la couverture *fonctionnelle*. Si les outils de couverture de code permettent de connaître les portions de code qui ont été simulées, la couverture *fonctionnelle* permet de répondre à des questions comme « Avons nous essayé telle valeur particulière dans ce bloc-ci, lorsque ce dernier était dans tel état ? » ou encore « Avons-nous testé telle séquence alors qu'une interruption était déclenchée ? ». Il s'agit donc de permettre à l'ingénieur de vérification d'avoir une vue globale de son système et de lui permettre d'exprimer des tests et requêtes de haut-niveau.
D'après les chiffres que l'on peut trouver sur le site Deepchip[1], trois solutions se partagent le marché. Les trois apportent des fonctionnalités a peu près similaires. On va retrouver les caracteristiques que l'on a déjà cité pour *Superlog* (structures de données complexes, gestion et synchronisation des processus, ...) mais ces langages facilitent de plus la réutilisation des tests en introduisant une notion de spécialisation (on parlerait d'*héritage* dans les langages objets).
Essayons de distinguer les points forts de chacun des produits en quelques lignes[2].

☞ **VERISITY *e/Specman Elite*™** [3]

Specman Elite est un environnement complet de vérification et se présente comme ayant deux objectifs :
- la réduction du temps et des ressources nécessaires à la vérification
- une amélioration de la qualité des tests

Pour cela, Verisity propose de mettre en œuvre trois technologies :
- la génération de tests avec contraintes
- la vérification de propriétés et de propriétés temporelles
- l'analyse de la couverture fonctionnelle

et a définit un nouveau langage appelé *e*. Nous le présenterons brièvement dans la section suivante.

Verisity a des accords avec des fournisseurs d'IP (Intellectual Properties) qui permettent à ces derniers de fournir de véritables composants de vérification (les eVCs[4] pour e Verification Components). Ainsi, lors de la conception d'un circuit SoC (System On a Chip), il est possible d'acheter une partie du circuit, mais aussi le composant de test qui permettra d'intégrer l'IP à la vérification globale du système.

☞ **SYNOPSYS *Vera Testbench automation*** [5]

Vera se définit lui aussi comme un langage spécialisé dans la vérification fonctionnelle. Sa syntaxe reprend à la fois celle des HDL et de C/C++. L'objectif clairement affiché est, là encore, d'accroitre la *productivité* des ingénieurs de vérification. *Vera* s'interface avec VHDL, Verilog ou C/C++ ainsi que d'autres langages. Il est ainsi possible d'utiliser les modèles et bancs de tests déjà développés. Il est même possible, de connecter, via des sockets n'importe quel langage (C++, Perl, Java, ...) pour utiliser l'API de Vera. *Vera* obéit aux principes de la programmation orientée objet. Il est donc possible de créer des composants de tests réutilisables.

Vera supporte les tests automatiques (self-checking tests) et il est possible de définir des contraintes utilisées pour générer automatiquement les stimuli. Avec les tests aléatoires, la question est *combien de temps* faut-il exécuter un test particulier ? Pour résoudre ce problème, Synopsys insiste sur la génération de tests *réactifs* qui est le point fort de leur logiciel : *Vera Testbench Automation* a la capacité de changer les contraintes et les distributions aléatoires *en cours* de simulation. Il est capable de travailler avec *VCS Coverage Metrics* pour évaluer *dynamiquement* la couverture et agir en conséquence. En définissant des objectifs de couverture, il est possible de déterminer quand un test est complet et le stopper automatiquement pour passer au suivant. Ainsi, le test se gouverne lui-même jusqu'à être complet.

[1] http://www.deepchip.com/items/dac01-14.html

[2] Notons qu'il est assez difficile de trouver (gratuitement) de la documentation sur ces produits commerciaux. Les informations qui figurent ici sont donc issues des annonces et publicités des fabriquants mais aussi des dialogues échangés sur les listes [Dee] et [Ber].

[3] http://verisity.com/products/specman.html

[4] http://verisity.com/products/evc.html

[5] http://synopsys.com/products/vera/vera.html

☞ **FORTE**[1] ***Rave***

Quickbench[2] propose lui aussi d'élever le niveau d'abstraction. Moins répendu que les deux précedents, il met en avant la notion de « Bus Functional model ». La terminologie est différente mais il s'agit en fait des *transactions* que nous verrons en section 3.2. *Rave* (Reuse Architecture for Verification) est un langage spécial (basé sur Perl) qui peut être utilisé pour diriger les stimuli, comparer les résultats ou interagir avec ces transactions. QuickBench gère la génération aléatoire mais il n'intègre pas véritablement d'outils de couverture fonctionnelle.

Signalons enfin une initiative récente :
☞ ***Jeda***[3] : *Jeda* est un langage de vérification créé par Atsushi KASUYA, ingénieur de vérification à JUNIPER NETWORKS. Ce langage à la particularité d'être *libre* (au sens d'un logiciel libre) puisque son auteur l'a placé sous licence GPL. Il s'agit d'un projet personnel, non d'un développement de JUNIPER NETWORKS[4]. L'objectif est de rassembler dans un même langage les meilleures caractéristiques de C, Java, C++, Perl ou Verilog. *Jeda* offre des mécanismes pour la programmation concurrente, orientée objet et dispose d'un ramasse-miettes (garbage-collector).

Brève présentation de *e*

Pour avoir une idée un peu plus précise des particularités des HVL, examinons brièvement *e*. Au contraire de *Vera* ou *Rave*, *e* ne revendique aucune filiation, dans la syntaxe, avec le C ou Perl. Il s'agit d'un langage vraiment original.

Le langage *e*[5] présente des caractéristiques classiques d'un langage de programmation : construction de boucles (`while`, `for`, `for each`, ...), instructions conditionnelles (`if`, `case`, ...). Il peut naturellement travailler avec des types numériques, mais au contraire des HDL, il sait aussi travailler sur des chaînes de caractères. De très nombreuses fonctions permettent de travailler facilement avec des listes (`list.clear()`, `list.add(value)`, `list.pop()`, `list.reverse()`, `list.size()`, etc.).
Les tableaux associatifs (`list.key_exists(value)`, etc.) et les types énumérés (`kind`) sont également présents.

Le langage *e* permet de définir des structures (`struct`) qui rappellent les classes des langages objets, et qu'il est possible de réutiliser et spécialiser (`extend`, `is also`, `is first`, `is only`, `when`).

Bien sûr, des mécanismes sont prévus pour interfacer le programme *e* avec une description écrite en HDL. Des fonctions (`pack`, `unpack`) permettent par exemple de faire correspondre aux signaux de bas niveau (en HDL) les données structurées (`struct`) utilisées par *e*.

Mais, c'est grâce à des spécificités propres à la vérification que le langage prend toute sa dimension. Nous allons les évoquer brièvement à travers les techniques de tests qu'elles permettent de mettre en œuvre.

✓ **Tests avec contraintes** : *e* définit des mots clés (`gen`, `keep`, `keep soft`, `select`) pour générer facilement des tirages aléatoires avec contraintes. Il est même possible de définir des contraintes dépendant du contexte (on-the-fly c.-à-d. pendant la simulation en cours d'exécution). Cela est particulièrement pratique pour définir des cas particuliers (corner cases) pour certaines phases de tests.
Exemple : Lors d'un test de processeur, on veut privilégier les opérations arithmétiques et dans une moindre mesure les opérations logiques. Ici, on veut privilégier, de plus les chances d'obtenir un JMPC lorsque le signal `carry` de la description HDL est positionné.

```
keep soft opcode == select {
40 : [ADD, ADDI, SUB, SUBI];  // poids 40
20 : [AND, ANDI, XOR, XORI];  // poids 20
10 : [JMP, CALL, RET, NOP];   // poids 10

'top.carry' * 90: JMPC ;      // poids 90 si le signal carry
                              // est positionné (0 sinon)
};
```

✓ **Définition de points de couverture** : il est possible de définir des *points de couverture* qui pourront être ensuite analysés (de façon graphique) dans l'environnement *Specman*. Le principe, pour définir une couverture (`cover`) est de donner un événement (`event`) auquel on associe la sauvegarde des différents éléments (`item`). Il est possible, à ce stade d'organiser et de mettre en forme (`illegal`, `range`,

[1] *QuickBench Sequencer* et *Rave* ont été développés par CHRONOLOGY qui a fusionné récemment avec CYNAPPS pour former FORTE (http://www.ForteDS.com)
[2] http://www.ForteDS.com/products/qb_overview.html
[3] http://jeda.org
[4] Un ingénieur propose un langage de vérification en source ouverte : http://www.edtnfrance.com/technologies/OEG20011121S002
[5] Les termes en police à pas fixe (type `machine à écrire`) correspondent ici aux mots clés du langage.

`transition`, ...) les éléments pour explorer ensuite facilement les résultats. C'est ainsi que l'on peut garder des traces de haut niveau de l'activité de la simulation et des tests effectués.

Specman permet de travailler sur des couvertures mais aussi sur des intersections de plusieurs couvertures (`cross`).

✓ **Vérification de propriétés temporelles** (temporal checks) : la commande `expect` permet d'effectuer la vérification d'une propriété temporelle. Le paramètre attendu est une *expression temporelle* ("TE" pour Temporal Expression) que l'on peut exprimer et utiliser grâce aux facilités du langage (`event`, `emit`, `on`, `delay`, ...)

Il est possible d'utiliser une syntaxe proche des expressions régulières pour marquer la répétition d'expressions temporelles. Un opérateur particulier (=>) permet d'indiquer l'implication de deux expressions temporelles[1].

Il est possible de créer un nouveau fil d'exécution (thread) simplement en détachant (`detach`) une expression temporelle.

Exemples :

```
// Définition d'expression :
// @clk correspond au signal d'horloge de la description en HDL
// Evénement A : "TE2 survient entre n1 et n2 cycles d'horloge après TE1"
       event A is {TE1; [n1-1..n2-1]; TE2} @clk;

// Vérification d'expression :
// TE3 doit arrivé n cycles après que TE1 ou TE2 soit survenue
       expect (TE1 or TE2) => { [n-1]; TE3 }@clk;
```

✓ **Vérification de propriétés** (data checks) : La vérification de propriétés consiste à contrôler, à certains moments, l'état de certains aspects du DUV par rapport à un modèle de référence (golden model). Ce dernier peut être écrit en Verilog, en VHDL, en C ou, bien sûr en *e*. Cela suppose deux choses :

– Le besoin de *synchroniser* le modèle de réference avec le DUV. Cela passe par des lignes de programmes supplémentaires mais ne nécessite pas de fonctionnalités supplémentaires au niveau du langage.
– Des points de contrôles réguliers. Le langage *e* met à disposition pour cela la commande `check` qui effectue le contrôle et réagit en cas de problème.

Exemple :

```
//on définit un événement pour définir les moments de vérification
  event testevent is @sig;  // a chaque fois qu'on a le signal sig
// La vérification :
  on testevent() is {
  check that sys.module.var==sys.module.refmodel.var else
    dut_error("Var ne correspond pas à ce que prévoit le modèle !") ;
  };
```

Specman offre par ailleurs un environnement de déboguage complet pour le langage *e*.

L'absence de standard

Il y a plusieurs langages de vérification, mais pas de véritable standard. Or, ce qui a fait le succès de Verilog et de VHDL, c'est qu'ils sont devenus des standards supportés par de nombreux produits. L'arrivée des bibliothèques de classes C++, souvent en open-source[2] (voir section 2.5) met encore plus ce problème en lumière. Synopsys a réagit en créant l'initiative[3] Open-Vera[4] et Verisity vise une standardisation d'une partie du langage *e*[5].

Il est également question de continuer à faire évoluer les HDL eux-mêmes. Lors d'une conférence récente à San Jose[6], les propositions d'amélioration pour Verilog étaient généralement basées sur *Superlog* et *OVL*. Mais Superlog reste un langage propriétaire[7] même s'il semblerait que CO-DESIGN AUTOMATION vise aussi une standardisation[8].

[1] TE1 => TE2 signifie « Si l'expression TE1 survient, alors TE2 *doit* survenir »

[2] Un programme *open-source* est livré avec ses sources, ce qui permet éventuellement de comprendre son fonctionnement, de le modifier ou de le compléter. La distribution de versions modifiées est possible ou non selon la licence du logiciel.

[3] Synopsys ouvre *Vera* comme langage de vérification standard : http://www.eetimes.com/story/OEG20010402S0048

[4] http://open-vera.com

[5] Verisity prépare la standardisation du langage *e* : http://www.eetimes.com/story/OEG20000501S0053

[6] La standardisation approche pour la prochaine génération de Verilog : http://www.eetimes.com/story/OEG20011114S0029

[7] à l'exception de la partie "Extended Synthesizable Subset" (ESS)

[8] Co-Design prépare la standardisation de *Superlog* :http://www.eetimes.com/story/OEG20010611S0102

Les librairies de classes pour la vérification

Plutôt que de développer de nouveaux langages spécialisés, deux sociétés ont choisi d'enrichir les possibilités de langages *standards* en fournissant une librairie permettant de travailler sur des bancs de tests matériels efficaces. Là encore, il est question de travailler à un plus haut niveau d'abstraction sans avoir à gérer le détail des communications avec le simulateur.

- ☞ ***Pivot***[1] : commercialisé par GREENLIGHT, il s'agit d'un ensemble de classes facilitant l'écriture de bancs de tests en Perl 5. *Pivot* est commercialisé à partir de 15 000 $[2].
- ☞ ***TestBuilder*** : Développé par la société CADENCE, *TestBuilder* est une librairie de classes C++. *TestBuilder* est téléchargeable gratuitement sur Internet. Nous consacrons le chapitre 3 à l'étude de cette librairie.

2.5 Une poussée de C++ dans les outils de CAO électronique

Notons que le C++ commence également à être utilisé pour la *modélisation* de circuits grâce a des bibliothèques de classes appropriées. Il s'agit d'une percée récente mais qui semble assez forte, et ces initiatives dessinent peut-être une évolution profonde, notamment parce qu'elles sont plus aptes à modéliser des systèmes hybrides (contenant à la fois du matériel et du logiciel).
Il n'est cependant pas question, pour le moment de remplacer Verilog ou VHDL, qui ont l'avantage fondamental d'être *synthétisables* (du moins pour une partie du langage).
On peut citer trois librairies de modélisation C++, les trois sont téléchargeables sur Internet : ***SpecC***[3], ***SystemC***[4] et ***Cynlib***[5].

2.6 Conclusion

Nous avons présenté, dans cette partie, les différents types d'outils disponibles pour la vérification de circuits.

Les outils de type *Lint* automatisent en quelque sorte la relecture d'une description HDL pour détecter quelques erreurs classiques. Verilog n'assurant pas le contrôle de types, ces outils sont encore largement utilisés.

Le *contrôle d'équivalence* permet de s'assurer que deux descriptions sont équivalentes, ce qui garantit les étapes de synthèse et les optimisations qui peuvent y être associées.
Le *contrôle de modèle* permet, lui, de démontrer des propriétés. On l'utilise aussi bien pour valider des descriptions RTL que des protocoles. Mais on peut aussi exploiter son aptitude à fournir des contre-exemples pour dépister des erreurs de conception.

La *simulation* reste le principal moyen de vérification. De nombreux logiciels sont associés au simulateur pour faciliter l'exploitation des résultats ou encore permettre d'évaluer la couverture de la vérification.
La qualité d'une simulation dépend beaucoup des *bancs de tests* qui visent à recréer le contexte du circuit afin d'évaluer son comportement dans différentes circonstances. Avec l'augmentation de la complexité des circuits, les tests directifs se sont révélés insuffisants et il a fallu imaginer d'autres techniques. Les bancs de tests modernes ont, de ce fait, atteint un niveau de complexité comparable aux logiciels classiques. Or les HDL ne permettent pas autant de facilités que les langages de programmation traditionnels.

Nous avons présenté les différentes solutions proposées par les éditeurs, en introduisant les techniques spécifiques à la vérification (tests aléatoires avec contraintes, vérification de propriétés et de propriétes temporelles et notion de couverture fonctionnelle) à travers l'exemple du langage *e*.

Nous avons terminé en notant la progression du langage C++ dans les outils de CAO électronique. Cette tendance, bien que récente, est particulièrement notable en ce qui concerne la modélisation.

La librairie *TestBuilder* que nous allons maintenant étudier est elle-même écrite en C++ et prétend proposer les mêmes avantages que les langages de vérification, sans souffrir de leur côté propriétaire.

---

[1] La puissance de Perl intégrée à un environnement de développement de banc de tests : http://www.greenl.com/brochure.htm

[2] http://www.eedesign.com/story/OEG20010723S0029

[3] *SpecC* a été développé à l'université de Californie avec l'aide de TOSHIBA, HITACHI et d'autres compagnies. http://www.specc.org

[4] *SystemC* est né à travers l'initiative d'un consortium qui regroupe de grandes compagnies de conception d'outils de CAO électronique (dont SYNOPSYS et CADENCE). http://www.systemc.org

[5] Développé par CYNAPPS et diffusé aujoud'hui par Forte, *CynLib* fait partie de la suite QuickBench Verification Suite au même titre que *Rave*. http://www.forteds.com/products/cynlib.html

3 TestBuilder

3.1 Présentation de la librairie

TESTBUILDER[1] est une librairie de tests *open-source* écrite en C++ et distribué par CADENCE. La version actuelle est la 1.3.172[2]. Elle est supportée sur Solaris, HP-UX et Linux. La licence autorise la modification mais il y a des restrictions sur la *distribution* de versions modifiées. En clair, les modifications doivent être *approuvées* par CADENCE avant d'être définitivement intégrées à la librairie.

Nous allons détailler l'ensemble des facilités offertes par la librairie, mais il nous faut d'abord aborder la notion de *transaction* qui est l'argument phare de cette bibliothèque. Notons qu'il s'agit d'une notion relativement *classique* en vérification. Elle tend du moins à se généraliser car elle rationalise les étapes de vérification et elle facilite, comme on va le voir, la réutilisation des tests. Cette notion apparait parfois sous le nom de BFM (Bus functional model).
La particularité de *TestBuilder* est d'inciter très fortement le programmeur à utiliser cette voie par un jeu de classes appropriées.

3.2 La notion de transaction

La méthodologie TBV (Transaction-Based Verification) vise une élévation du niveau d'abstraction pour les tests, des signaux vers les *transactions* [Cad00]. Grâce aux transactions, on parlera par exemple d'opérations de *lecture/écriture* plutôt que des multiples signaux de bas niveau tels que *enable*, *read/write*, *address* et *data*.
L'objectif est de permettre aux concepteurs de ne plus être accaparés par les détails des signaux et les contraintes de gestion du temps (timing) mais de pouvoir au contraire se concentrer sur des tests au niveau du système global.
Le principe est de décomposer fondamentalement les tests en deux couches : les *tests* à proprement parlé (au niveau d'abstraction souhaité) et les *Transactors*[3] ou TVMs (Transaction Verification Models).

1. Le *Test* correspond à la structure de haut niveau. Un test manipule et coordonne des transactions au niveau du système sans se préoccuper des détails des signaux. Il s'agit en fait d'un programme qui génère des séquences organisées d'appels au TVMs.
2. Les *TVMs* ou *transactors* représentent la couche du bas. Ils permettent la correspondance entre le niveau des transactions utilisé par les tests et le niveau des signaux utilisés par les interfaces. Un TVM est formé d'une collection de *tâches*, chacune effectuant une transaction particulière.

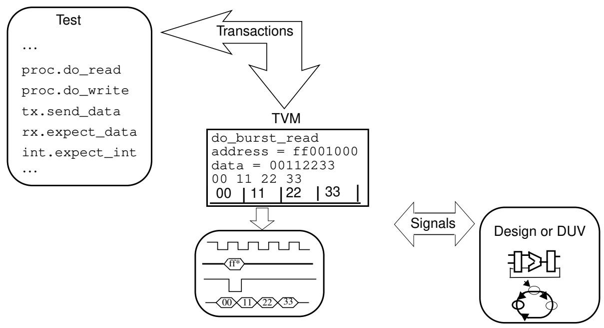


FIG. 3.1 – Méthodologie TBV (document CADENCE)

Nous verrons plus particulièrement le détail de l'implémentation en section 3.5.3 (page 22).
Notons déjà que les *transactors* s'exécutent en parallèle, et que des mécanismes de synchronisation sont prévus dans la librairie.

Une transaction peut être aussi simple qu'une lecture mémoire ou aussi élaborée que le transfert d'une structure de données complexe via un canal de communication. Ce sont les concepteurs des tests qui choisissent le niveau d'abstraction qui correspond le mieux à leur vision.

[1] http://testbuilder.net
[2] à la date du 7 mars 2002
[3] Aucune traduction de ce terme ne semble utilisée. Nous garderons donc exceptionnellement le terme d'origine.

3.3 Avantages des transactions

Voyons les avantages d'un tel modèle.

Augmentation du niveau d'abstraction et division des responsabilités

Une telle séparation en couches participe directement à une meilleure *organisation* des tests, qui peuvent être développés plus rapidement. Le détail d'un protocole est ici *encapsulé* dans un *transactor*.

Les ingénieurs ayant une bonne connaissance des protocoles et des signaux échangés au plus bas niveau développent les *transactors*, tandis que ceux qui doivent avoir une vision d'ensemble du système se concentrent sur les tests de haut niveau. Le principe emprunte beaucoup au domaine logiciel où l'on peut diviser la responsabilité des développements au sein d'une équipe entre développeurs d'objets de bas niveau (widgets graphiques, protocoles réseau, etc.) et développeurs d'applications. Les ateliers de génie logiciel où l'on travaille avec des *composants* ont permi de beaucoup gagner en temps de développement.

Des bancs de tests lisibles et concis

En encapsulant le protocole dans un module réutilisable, le banc de tests gagne en *lisibilité* et en *taille*. Il faut beaucoup moins de lignes de code pour programmer les tests. La *maintenance* est également plus simple.

Des possibilités accrues de réutilisation

Le modèle transactionnel favorise la *réutilisation*.
- Des tests similaires peuvent être utilisés sur des conceptions proches dans leur fonctionnalités mais différentes dans leurs implémentations (exemple : deux systèmes offrant des fonctionnalités identiques mais basés sur un protocole différent). Dans ce cas, seuls les *transactors* seront à réécrire ou à adapter.
- A l'inverse, les *transactors* peuvent être réutilisés au sein de différents projets. Il n'est pas nécessaire de (re)programmer les détails des protocoles et des échanges de signaux à chaque fois.

Nous verrons, de plus, que les TVMs et les tâches sont des classes C++. On peut donc utiliser la technologie objet pour *spécialiser* des éléments de tests par *héritage*.

Des bancs de tests plus complets ; amélioration de la couverture

Grâce aux modèles en couches, des scénarios de tests beaucoup plus complexes peuvent être imaginés et on a ainsi plus de chances de découvrir les erreurs difficiles à cerner.
En permettant de *penser* la vérification au niveau du système (plutôt qu'au niveau des signaux élémentaires), les tests deviennent plus naturels et plus intuitifs pour le concepteur, qui est plus à même d'avoir une vue globale des tests à effectuer pour obtenir une couverture satisfaisante.

Des outils utilisant les transactions

Bien sûr, il est plus pratique de travailler avec des outils de visualisation et de déboguage qui acceptent eux-mêmes le niveau transactionnel. CADENCE commercialise de tels outils. Au fur et à mesure de la simulation, les transactions sont enregistrées dans une base de données[1]. *SignalScanTX* permet de visualiser de façon graphique les transactions alors que *Transaction Explorer* (*TxE*) permet d'écrire des scripts de recherche dans le(s) base(s) de données créée(s) lors des simulations. Les scripts de recherche sont en fait écrits en TCL, dans un cadre fixé par *TxE* qui facilite la sélection du type de transactions recherché, leur mise en forme, etc... Une section *goal* permet par exemple d'indiquer si un but est atteint ou non en fonction des résultats de la recherche et des valeurs attendues. Ces informations sont directement affichables sous forme graphique dans *SignalScanTX*. Ainsi, l'écriture de scripts en *TxE* et l'affichage des résultats dans *SignalScanTX* permettent au concepteur d'améliorer sensiblement la *couverture fonctionnelle*. Il est possible d'acquérir l'ensemble de ces outils dans une suite de vérification appelée *Verification Cockpit*[2]. Outre *TestBuilder* et les outils que nous venons de citer, Cockpit intègre un outil de type Lint ainsi qu'un outil de couverture de code traditionnel.

3.4 Mise en œuvre de techniques de tests avec les transactions

Outre les avantages *directs* cités ci-dessus, le modèle transactionnel permet de mettre en œuvre de façon élégante des techniques de tests particulièrement puissantes :

[1] un simple fichier texte si on ne dispose pas des outils commerciaux de CADENCE.
[2] http://www.cadence.com/products/cockpit.html

Les tests de *causes à effet*

Le plus souvent, lors de l'écriture de tests, on a une idée de ce qu'on *devrait* obtenir en sortie lorsqu'on excite le modèle avec certains stimuli. On serait donc tenté d'introduire la *réponse attendue* quelque part dans les tests pour que la vérification se fasse de façon *automatique*. Malheureusement, il n'est souvent pas facile de tenir compte des contraintes temporelles qui font que le résultat n'est testable qu'après un temps de propagation et de réaction du circuit au stimuli. Devant ces difficultés de mise en œuvre, il est fréquent de renoncer à l'automatisation du test et on se contente alors d'enregistrer les résultats dans un fichier que l'on comparera *à posteriori*. On pourra bien sur utiliser des outils de comparaison automatique (de type diff) mais la procédure reste assez lourde.
Grâce à l'exécution *simultanée* des TVMs, il devient possible d'implémenter facilement les tests *de cause à effet* (cause & effect testing) sans se préoccuper des détails complexes de gestion du temps. On crée pour cela un TVM spécial (on parle d'esclave actif (active slave)) : tandis qu'un TVM crée le stimuli, l'esclave actif compare la sortie avec la réponse attendue et signale les erreurs éventuelles.
Ce modèle est particulièrement efficace lorsque les résultats de la transaction en jeu peuvent être *calculés* en utilisant un modèle de référence. Dans ce cas, on peut en effet générer les stimuli de façon *aléatoire*.

Tests aléatoires avec contraintes

L'utilisation de stimuli aléatoires permet d'une façon générale d'*automatiser* les bancs de tests. C'est une technique beaucoup plus économique en temps de développement que l'écriture de tests déterministes.
Mais d'autres paramètres que les stimuli peuvent être générés aléatoirement. Avec le modèle transactionnel et un bon mécanisme de synchronisation des tâches, il est possible d'intervenir sur :
– le TVM qui initialise la transaction
– la tâche spécifique (ou les séquences de tâches) appelés sur ce TVM
– les arguments utilisés dans l'invocation des tâches
– la synchronisation des tâches
Nous verrons (section 3.5.4) quelles sont les facilités implémentées dans *TestBuilder*.

3.5 Étude de la librairie

Venons en à une étude plus concrète de la librairie. Les informations ci-dessous résultent de la documentation ainsi que du code source de *TestBuilder* [TB]. Le logiciel *Doxygen*[1] a été utilisé pour explorer les fichiers sources et pour visualiser graphiquement la hiérarchie des classes.

3.5.1 Le choix de C++

L'utilisation du C++ apporte de nombreux avantages :

☞ C'est un langage *standard*, *connu* de nombreux programmeurs. On évite ainsi, au contraire des HVL propriétaires, un apprentissage particulier et une dépendance forte vis-à- vis d'un éditeur.
De plus, de nombreux outils existent déjà. C'est le cas des compilateurs bien sûr, mais aussi des éditeurs, débogueurs, etc.[2]

☞ Les possibilités d'extensions sont facilitées par rapport aux langages propriétaires. Ainsi, si le C++ standard n'offre pas de facilités pour le *multithread*, il a suffit d'ajouter une librairie (en fait une adaptation de Pthread et de QuickThread pour la version Linux) pour pallier ce problème. De même, le solveur de contraintes a été entièrement réécrit[3] et il utilise aujourd'hui le package Cudd de l'université du Colorado, qui permet de traiter des arbres de décisions binaires (BDD).

☞ Le C et le C++ sont déjà largement utilisés pour décrire des modèles de référence (golden models) de haut niveau d'abstraction. Par exemple, il est fréquent de coder en C un algorithme pour le tester avant même d'envisager de le cabler dans un circuit (algorithme de calcul, compression, etc.). *TestBuilder* étant lui même écrit en C++, on peut facilement réutiliser de tels modèles comportementaux au sein de bancs de tests. Nous avons vu l'intérêt de cette démarche pour la *vérification de propriétés* lors de notre présentation de *e*.
Nous avons noté également (section 2.5) une *émergence* du C++ dans la modélisation de systèmes complexes. *TestBuilder* est d'ores et déjà prêt à travailler avec SystemC.

[1] *Doxygen* est un logiciel libre de documentation. http://www.snack.nl/~dimitri/doxygen/
[2] *TestBuilder* se compile avec une version récente de gcc, le compilateur C/C++ du GNU.
[3] à l'occasion de la version 1.1 en juin 2001

☞ L'orientation objet de C++ facilite la réutilisation des composants utiles aux tests. Cela permet la création de bancs de tests ayant une grande couverture sans trop de redondance. On peut spécialiser des TVMs ou des tâches par *héritage*. Ainsi, si on a écrit un *transactor* pour gérer le bus PCI, l'implémentation du bus PCI-X devrait se faire par héritage et redéfinition de fonctions.

☞ Le C++ produit un code exécutable *efficace*. Il est important de ne pas ralentir le processus de simulation dont nous avons vu qu'il était fondamentalement lent.

☞ L'utilisation des Templates du C++ permet de travailler sur des types de données abstraits.

☞ La possibilité d'utiliser des pointeurs et références permet de travailler efficacement avec des blocs mémoire importants.

☞ La surcharge des opérateurs permet de travailler avec élégance sur de nouveaux objets (par exemple les signaux).

Le C++ a donc de nombreux atouts et il propose les fonctionnalités qui manquent aux HDL pour l'écriture de bancs de tests complexes. Mais le C++ reste un langage *généraliste*. Pour pouvoir l'utiliser avec efficacité en vérification de circuit, la librairie a dû introduire des concepts liés au materiel (Hardware Concepts).

C'est précisement cet aspect que nous allons maintenant aborder.
Nous verrons ensuite comment la notion de transaction a été implémentée et nous essaierons de couvrir de façon brève mais exhaustive l'ensemble des autres facilités offertes par la librairie.

3.5.2 Implémentation de concepts liés au matériel

TestBuilder définit des classes et des méthodes pour travailler avec les concepts fondamentaux d'une modélisation de matériel, notamment :
– les signaux et la manipulation des valeurs des signaux
– la gestion du temps (timing) et le suivi des signaux (monitoring).
– l'instanciation de diverses *structures* qui correspondent à des entités qui doivent être présentes tout au long de la simulation (surveillance de données, etc.).

Les signaux et leur manipulation

Les opérations sur les *signaux* sont définies dans deux classes : `tbvSignal2StateT` pour la logique deux états[1] et `tbvSignal4StateT` pour la logique quatre états.[2]
Exemple :

Il est par exemple possible de définir deux signaux de quatre et seize bits respectivement à l'aide des lignes :

```
tbvSignal2StateT myOpCode(3,0); 3
tbvSignal2StateT myOpA(15,0);
```

Et la surcharge des opérateurs permet d'écrire :

```
if (myOpCode == INCREMENT) ++myOpA;
```

Liaison avec le modèle HDL

En tant que librairie de test, *TestBuilder* se doit de supporter la communication entre les valeurs des signaux du programme C++ et la conception écrite en HDL. Pour cela, on utilise les classes `tbvSignalHDL2StateT` et `tbvSignalHDL4StateT` dérivées respectivement de `tbvSignal2StateT` et `tbvSignal4StateT`. Une instanciation d'une de ces classes peut être vue comme un *alias* à un signal de la description HDL.
Exemple :

Pour travailler sur le signal `regA` du module `top` d'une description Verilog, on pourra utiliser tout simplement :

```
tbvSignalHdl4StateT regA("top.regA");
regA = 3;
```

[1]Logique deux états : 0 ou 1 comme le type `BIT` de VHDL par exemple.
[2]Loqique quatre états (0, 1, x (indeterminé), z (haute-impédance)) comme en Verilog.
[3]Les bornes représentent, comme en Verilog, les bits extrèmes : poids le plus fort (MSB pour Most Significant Bit) et poids le plus faible (LSB pour Less Significant Bit).

Par défaut, la liaison entre *TestBuilder* et les signaux modélisés en HDL est bidirectionnelle (`READ_WRITE`). Mais il est possible de définir des alias `READ_ONLY` ou au contraire de prévoir des liaisons de type `WRITE_ONLY`.

La communication entre *TestBuilder* et le simulateur se fait via les mécanismes habituels (PLI, VPI, etc.) mais cela est transparent pour l'utilisateur. En fait *TestBuilder* utilise des *adaptateurs*[1]. Ainsi, il suffit de créer un nouvel adaptateur pour qu'un nouveau simulateur soit *compatible* avec *TestBuilder*. Cadence fournit des adaptateurs pour les simulateurs Verilog et VHDL les plus courants[2] mais une démonstration a été faite avec SystemC par exemple.

Du point de vue de l'utilisateur, il faut simplement ajouter quelques appels dans le code HDL. Prenons l'exemple de Verilog.
Il suffit de placer la ligne `$tbv_main` dans un module Verilog pour utiliser *TestBuilder*. En fait, lorsque cet appel est exécuté, la procédure `tbvMain` (qui constitue le point d'entrée du banc de tests) est lancée dans un fil d'exécution (*thread*) qui s'exécute *en parallèle* avec le simulateur.

Une autre forme d'appel est utilisée afin de permettre d'instancier des TVMs qui doivent être présents tout au long de la simulation. On utilise pour cela la procédure `$tbv_tvm_connect`.
Les paramètres à passer sont le nom du TVM utilisé[3], le nom que l'on donne à cette nouvelle instance et une correspondance entre les noms de variables C++ et les noms des signaux VHL.
Exemple :

> `initial $tbv_tvm_connect("thirdTvm","tvm3","a_in",a);` permet la création d'une instance du TVM `thirdTVM` nommé `tvm3`.

Gestion du temps et suivi de valeurs

La gestion du temps est *essentielle* en simulation. Une fonction globale `tbvWait` permet de suspendre le thread d'exécution tant que la simulation n'a pas avancé d'un nombre de pas.
Exemple :

> `tbvWait(5);` attend 5 unités de temps

Il est aussi possible d'attendre des changements de valeurs de signaux (monitoring) tout aussi simplement :
Exemple :

> `tbvWaitCycle(regA);` attend un changement de valeur sur `regA`.

On peut par ailleurs préciser l'attente d'un front montant ou descendant.

3.5.3 Implémentation des transactions

Transactors et tâches

Les concepts de TVM et de tâches apparaissent dans *TestBuilder* comme deux *classes abstraites* : `tbvTvmT` et `tbvTaskT`. Par exemple, un TVM de processeur peut être décrit comme une classe `myTvmT`, *dérivée* de `tbvTvmT` avec deux membres `doReadT` et `doWriteT` correspondant respectivement aux demandes de lecture et d'écriture. Les classes `doReadT` et `doWriteT` sont elles-mêmes dérivées de `tbvTaskT` et contiennent une référence à leur TVM parent.
Exemple :

```
class myTvmT : public tbvTvmT {
public:
   tbvSignalHdlT4State a;
   ...
   doReadT  doRead;
   doWriteT doWrite;
};
```

```
class doWriteT : public tbvTaskT {
protected:
   myTvmT& parent;
...
};
```

Le corps d'une tâche doit être spécifié dans une méthode `body` et l'argument utilisé lors de l'invocation est un pointeur sur une structure de donnée appelée smart data[4]. Une *smart-data* correspond à une donnée, ou à un ensemble de données (SmartRecord, SmartArray) pour lequel *TestBuilder* offre un certain nombre de facilités (voir 3.5.4 et figure 3.3 en page 24).
Ici par exemple, nous pourrons définir la classe writeDataT :

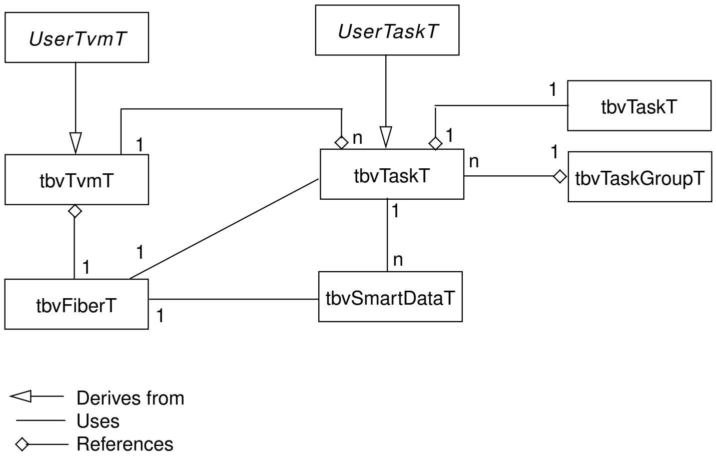
[1] *TestBuilder* définit une interface CHPI (Common HDL Procedural Interface) qui permet de masquer les particularités des interfaces de communication avec les simulateurs.
[2] Les simulateurs supportés sont à l'heure actuelle :Verilog-XL, NC-Sim, NC Verilog, NC VHDL et VCS.
[3] *TestBuilder* met en place des mécanismes d'enregistrement des TVM qui permettent de les appeler avec un nom en clair.
[4] Comme pour *transactor*, nous garderons exceptionnellement le terme d'origine.

```
class writeDataT : public tbvSmartRecordT {
public:
   tbvSmartIntT address;
   tbvSmartIntT data;
   ...
};
```

Avec ce mécanisme, on passe toujours un seul paramètre à la méthode `body`, quelle que soit la complexité des données en jeu.
La déclaration d'une méthode `body` est donc toujours de la forme :`virtual void body(tbvSmartDataT * d)`.

On doit également redéfinir, pour chaque classe de tâche, la méthode virtuelle[1] `tbvSmartDataT * createDefaultData` qui renvoie un pointeur sur un objet conforme à ce qu'attend la tâche.

La figure 3.2 montre les classes relatives aux *TVMs* et aux *tâches* mais aussi les classes environnantes que nous allons voir au fur et à mesure de notre étude.



– Un TVM fait généralement appel à plusieurs tâches pour implémenter les différentes transactions.
– Une tâche peut contenir une liaison avec une autre tâche ; il est ainsi possible d'établir des relations de précédence.
– Il est possible de *regrouper* les tâches. Cela peut permettre par exemple de créer des groupes de tâches afin de produire des appels aléatoires de tâches cohérents.
– Il est possible d'utiliser des smart-data pour représenter les données.
– On peut utiliser une fibre (fiber) en vue d'enregistrer des informations sur les transactions.

FIG. 3.2 – Transactors et tâches

Les tests

Un *test* se contente de communiquer avec les TVMs en appelant des *tâches*. Prenons l'exemple d'un test qui demande 100 écritures aléatoires ; on pourra écrire, grâce à la tâche `doWrite` que l'on vient de définir :

```
void test(myTvmT * tvmP)  {
    tbvSmartDataT * d = tvmP->doWrite.createDefaultData();
    for (int i=0; i<100; ++i)
      {d->randomize(); tvmP->doWrite.run(d); }
    delete d;
}
```

Au passage, signalons qu'il aurait tout à fait été possible d'invoquer la tâche dans un nouveau *thread* en utilisant la méthode `spawn` plutôt que `run`.
Ces deux méthodes appellent implicitement la méthode `body` mais alors que la première s'exécute dans le fil d'exécution courant, la deuxième en crée un nouveau.

Par défaut, l'exécution d'une tâche génère *automatiquement* une *trace* de la transaction dans une base de données en utilisant un objet de classe `tbvFibertT`. Il est ainsi stocké pour une transaction, des données temporelles (démarrage et fin) ainsi que les paramètres de l'appel d'une tâche[2]. Ces données pourront ensuite servir à des outils d'exploration des transactions comme ceux que l'on a évoqués en page 19.

[1]Les méthodes sont définies virtuelles pour pouvoir bénéficier du polymorphisme du C++.
[2]à condition de bien utiliser des types « Smart Data » ou dérivés.

3.5.4 Autres facilités

Les Smart data

Les *Smart Data* sont constituées d'un ensemble de classes (Fig. 3.3) et permettent d'utiliser des types de données personnalisés avec des facilités que n'offre pas le C++ en standard.
Parmi ces facilités, on peut citer :
– obtention d'informations relatives au type utilisé.
– lancement automatique de fonctions utilisateurs (call-backs) à l'initialisation et lors des changements de valeur[1].
– facilités pour les tirages aléatoires (voir plus bas)
La méthode `numFields`, par exemple, renvoie le nombre de champs dans un smart-record et on peut associer à l'assignation d'une donnée une méthode qui va enregistrer le changement dans une base de données...
Si l'on passe, comme conseillé, un smart data en paramètre lors de l'appel à une tâche, la transaction est enregistrée automatiquement avec les paramètres, ce qui facilite les exploitations à posteriori.

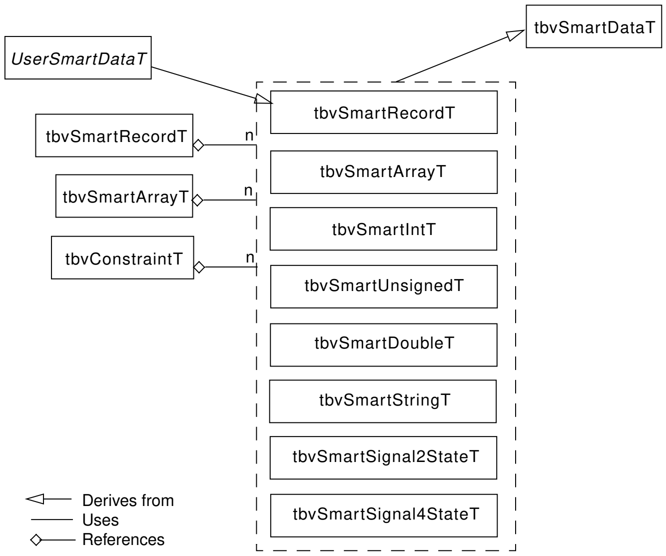


FIG. 3.3 – Hiérarchie des classes Smart Data

Structures de données

TestBuilder reprend les structures classiques de la libairie STL (Standard Template Library[2]) en ajoutant des caractéristiques (ou optimisations) qui peuvent être utiles pour l'écriture de tests. *TestBuilder* définit ainsi les modèles de classes suivants :

Modèles de classe	**Description**
tbvStringT	Classe de chaînes de caractères
tbvSparseArrayT	Tableau clairsemé [a] indéxé par des entiers
tbvWideSparseArrayT	Tableau clairsemé indéxé par des objets de type Signal
tbvAssociativeArrayT	Tableau associatif
tbvListT	Liste doublement chaînée
tbvPriorityListT	Liste doulement chaînée pour laquelle on peut choisir l'ordre[b]
tbvStackT	Pile simple
tbvQueueT	File d'attente FIFO (implémentée comme une liste STL)
tbvFastQueueT	File d'attente FIFO rapide (implémentée comme une file d'attente STL)
tbvPriorityQueueT	File d'attente pour laquelle on peut choisir l'ordre d'insertion
tbvSmartQueueT	Ensemble de files d'attente FIFO (sélectionnées par une fonction spécifique)
tbvBagT	Collection d'objets (doublons autorisés)
tbvSafeHandleT	Moyen d'accéder à un objet (handle) avec compteur d'utilisation[c]

[a] c.-à-d. économique en mémoire lorsque de nombreux éléments ont la même valeur
[b] `FIFO`, `LIFO`, `RANDOM`, ou différents modes personnalisables (`CUSTOM`)
[c] mécanisme permettant de ne détruire un objet partagé que lorsqu'il n'est plus utilisé

Afin de permettre de travailler sur n'importe quel type d'objet (principe de *généricité*), les classes *templates* du C++ ont été utilisées.

Enregistrement des transactions

Afin de permettre un enregistrement des transactions pour des analyses à posteriori, *TestBuilder* offre un mécanisme de sauvegarde via des canaux appelés fibres (fibers). L'enregistrement des transactions est pris en charge de façon automatique mais il est possible d'intervenir manuellement sur le mécanisme d'enregistrement en utilisant les méthodes définies dans la classe `tbvFiberT`.

[1] un peu comme le `Tie` du langage Perl 5.
[2] Voir http://www.sgi.com/tech/stl pour une documentation sur STL.

Tirages aléatoires et contraintes

La gestion basique de tirages aléatoires est prise en charge par la classe `tbvRandomT` qui permet d'obtenir un flux d'entiers non signés aléatoires. Mais *TestBuilder* offre de nombreuses autres possibilités. Il est ainsi possible, par exemple, d'utiliser `tbvSignalGeneratorT` pour obtenir un signal aléatoire. Les méthodes `keepOnly` et `keepOut` permettent par ailleurs de spécifier des plages de valeurs. On peut donc écrire, simplement :

```
tbvSignalGeneratorT randomSignal(32);   // générateur de signaux 32 bits
randomSignal.keepOnly("0xFF","0xFF00"); // selection d'une plage d'adresses
randomSignal.keepOut("0xEFF","0xFFF");  // exclusion de la ROM
for (int i=0; i<1000, ++i)
    tbvOut <<randomSignal.next() <<endl; //Génération de 1000 valeurs
```

Le moyen le plus pratique pour travailler sur des tests non déterministes, repose sur l'utilisation des *smart-data* et de la classe `tbvConstraintT` qui permet de définir des contraintes.
La méthode `randomize` des *smart-data* permet d'obtenir une valeur aléatoire en tenant compte des *contraintes*.
Par exemple, si `packetT` est un smart-record avec les champs `source` et `destination`, on pourra écrire :

```
packetT p; tbvConstraintT::add( p.source() != p.destination() );
p.randomize();
```

Notons que rien ne s'oppose à la redéfinition de la méthode `randomize` afin d'utiliser la pondération comme dans le langage *e*, mais cela conduit à un travail de programmation supplémentaire.

Il existe une façon élégante de programmer un tirage aléatoire des tâches à lancer : il est possible de *regrouper* les tâches. On dispose pour cela de la classe `tbvTaskGroupT` dont les deux principales méthodes sont :
– `addTask` pour ajouter une tâche
– `run` pour lancer une des tâches aléatoirement.

Signalons enfin que *TestBuilder* permet de choisir l'algorithme de tirage aléatoire (méthode `setSeedGenerationAlgorithm`) et rend possible l'obtention de séquences aléatoires *à l'identique* afin de pouvoir reproduire fidèlement une simulation. C'est particulièrement utile pour vérifier un circuit après une modification, dans le *même* contexte que celui qui a permi de découvrir un problème.

Vérification de propriétés et propriétés temporelles

TestBuilder étant écrit en C++, il est particulièrement adapté à la vérification de propriétés, puisqu'il s'agit généralement de comparer l'état du DUV avec un modèle le plus souvent écrit en C ou en C++. *TestBuilder* fournit par ailleurs un ensemble de classes permettant de tester des propriétés temporelles (*temporal checks*).
TestBuilder peut ainsi vérifier une condition particulière (la *propriété*) pendant une *fenêtre d'observation* précise et lancer une *action* particulière si la condition n'est pas vérifiée.
La classe `tbvPropertyT` permet de définir une propriété. On dispose de deux types de propriétés :
– `ALWAYS` : la condition doit être vraie à chaque fois qu'elle est évaluée pour une plage d'observation.
– `EVENTUALLY` : *TestBuilder* doit vérifier la condition au moins une fois au cours d'une fenêtre d'observation.
Si la propriété n'est pas vérifiée, *TestBuilder* affiche par défaut un message, enregistre une transaction particulière de type `Error` et peut lancer une routine que l'on aura définie.

D'autres classes sont utiles lors de la programmation de tests temporels :
– `tbvExpressionT` permet de créer des expressions booléennes qui seront évaluées plus tard (par exemple lorsqu'un événement survient).
– `tbvEventExprT` permet de définir et de *combiner* des événements.
– `tbvEventResultT` permet d'exprimer le résultat d'une attente sur un événement. Des opérateurs de comparaison ont été redéfinis.
– `tbvConditionT` pour évaluer *périodiquement* une condition.

TestBuilder fournit, avec ces classes, un nombre important de méthodes qu'il nous est impossible de détailler ici. Nous présenterons juste un exemple représentatif de test.
Exemple :
On pourra, grâce à de telles propriétés, vérifier qu'un signal accordant le bus (*grant*) intervient bien au cours des trois cycles d'horloge qui suivent la demande (*bus request*). On suppose qu'on dispose d'une description Verilog (à travers un module `simple` qui sollicite ces signaux)

```
include "TestBuilder.h"
void tbvMain()
tbvSignalHdl4StateT clk("simple.clk");       //alias sur l'horloge
tbvSignalHdl4StateT busreq("simple.busreq"); //alias sur le "bus request"
tbvSignalHdl4StateT busgra("simple.busgra"); //alias sur le "bus grant"
```

```
tbvEventExprT cycle(clk,tbvThreadT::POSEDGE);// expression "front montant horloge"
tbvPropertyT check("busgrantdelay");         // on définit la propriété
check.setEnable( busreq() );                 //  définit la condition pour
                                             //  le début de la fenêtre d'observation
check.setFulfill( busgra() );                //  la condition qui doit être vérifiée
check.setDisable( tbvExpressionT(FALSE) );   //  condition de fin d'observation
                                             //    (ici aucune)
check.setGlobalTrigger( cycle );             //  association à l'événement cycle
check.setWindowEnd( 3*cycle );               //  événement de fin de la fenêtre d'observation
check.setType(tbvPropertyT::EVENTUALLY);     //  la propriété doit être testée
                                             //  au moins une fois pendant
                                             //  la fenêtre d'observation
check.setActive(TRUE);                       // on active la verification
tbvWait(1000);                               // simulation pendant 1000 unités
tbvExit();                                   // on quitte la simulation
```

On notera que l'écriture est beaucoup moins concise qu'avec le langage *e* ou qu'avec une logique temporelle pour le contrôle de modèle.

Multithread et synchronisation

Nous avons vu qu'une tâche peut être lancée soit dans le thread courant (méthode `run`) soit dans un nouveau thread (méthode `spawn`). *TestBuilder* propose différentes facilités de synchronisation :

☞ **la synchronisation des *threads*** : *TestBuilder* implémente les notions classiques suivantes :
- L'exclusion mutuelle (classe `tbvMutexT`)
- Le sémaphore (classe `tbvSemT`)
- La barrière (classe `tbvBarrierT`)
- La boîte aux lettres (classe `TbvMailboxT`)

Exemple :

```
class myTvm : public tbvTvmT{
  public:
    ...
    tbvMutexT ifMutex;
    tbvSignalHdlT data;
    task1 t1;
  }
```

```
void task1::body(tbvSmartIntT* data) {
   ifMutex.lock(); // on "prend" le mutex
                   // si c'est possible;
                   // sinon, on attend .
   ... // traitement souhaité
   ifMutex.unlock{}; // on libère le mutex
 }
```

On dispose enfin des fonctions `WaitAllThreads()`,`WaitChildren()` et `WaitDescendant()`

☞ **la synchronisation avec la *simulation*** : Il doit être possible de réveiller tous les fils d'exécution en attente lorsqu'un événement attendu intervient dans la simulation. Une fonction `tbvGetWaiters()` permet d'avoir la liste de l'ensemble des threads qui attendent un événement particulier, qu'il s'agisse d'un signal ou d'un temps spécifique. *TestBuilder* définit par ailleurs différentes fonctions d'attente. Nous avons déjà vu des exemples en page 22.

Autres facilités

Pour être complet, il nous faut signaler l'existence de quelques facilités supplémentaires :

Flux de sortie : *TestBuilder* définit une classe `tbvOut` qui s'utilise comme le `cout` du C++. La seule différence avec `cout` est qu'en plus d'une écriture sur la sortie standard du simulateur, le message est aussi enregistré dans un fichier journal (par défaut `tbv.log`).

Fonction de sortie : La fonction `tbvExit()` permet de sortir de la simulation avec un code de statut.

Gestion des exceptions : *TestBuilder* propose par défaut une gestion des exceptions qui prévient l'utilisateur en cas de problèmes (par un message sur tbvOut), s'assure de l'integrité des données et retourne au code d'appel.

TestBuilder définit deux classes `tbvExceptionT` et `tbvExceptionEnvironmentT` pour personnaliser le traitement des exceptions. Il est ainsi possible de modifier les messages et/ou de lancer une procédure définie par l'utilisateur en cas d'erreur.

Un assistant : tbvWizard

TestBuilder est livré avec un utilitaire[1] appelé tbvWizard qui permet de créer la structure de base des fichiers C++. On peut l'utiliser de deux manières.

[1] écrit en Shell-script et en Perl

– On peut créer manuellement un fichier `header` C++ classique[1] qui contient la déclaration des TVMs, des tâches et des smart data utilisés auxquels on ajoute quelques macros particulières. tbvWizard utilisera ce fichier pour créer automatiquement le squelette des fichiers C++ avec les constructeurs, destructeurs, etc.
– Le fichier de macros utile à cette génération peut lui-même être généré automatiquement par tbvWizard depuis la description HDL contenant les appels à *TestBuilder* (voir `$tbv_tvm_connect` en 3.5.2).

L'assistant tbvWizard permet donc de créer automatiquement les squelettes des fichiers C++ nécessaires à l'implémentation des TVMs. Il permet d'éviter des erreurs et de gagner du temps : on peut directement se lancer dans l'écriture du code utile (méthode `body` des tâches par exemple).
Cela permet en partie de compenser la mécanique un petit peu lourde du C++ par rapport aux langages spécialisés (nécessité des constructeurs et destructeurs, gestion de la mémoire, etc.).

3.6 Conclusion sur TestBuilder

Grâce à la notion de *transaction*, *TestBuilder* permet d'élever le niveau d'abstraction des tests. Ces derniers gagnent en efficacité et en possibilité de réutilisation.

De nombreuses facilités permettent de travailler sur des *tests aléatoires avec contraintes* et des classes ont été écrites pour permettre la *vérification de propriétés temporelles*. On retrouve donc ici les arguments phares des environnements de vérification comme *e/Specman* (voir section 2.4.5).

TestBuilder permet d'enregistrer les transactions pour exploiter ensuite les données et évaluer la *couverture fonctionnelle*. Toutefois, à moins d'utiliser *Cockpit*, il y a un travail important de programmation pour exploiter ces données. La stratégie de CADENCE est donc probablement d'imposer *TestBuilder* pour vendre son logiciel *Cockpit*.

TestBuilder, comme nous l'avons vu, est probablement plus complexe à maîtriser que les langages spécialisés. Alors que *e* définit une syntaxe particulièrement concise et adaptée pour les expressions et vérifications temporelles, par exemple, il faut appeler un nombre important de méthodes avec *TestBuilder*.

En contrepartie, *TestBuilder* offre la liberté de développer plus facilement du « sur-mesure ». Un utilisateur averti peut en effet adapter l'outil à ses besoins et lui ajouter des fonctionnalités. Le côté standard du langage C++ et la libre disponibilité des sources de *TestBuilder* sont des arguments de poids.

« Notre vision de TestBuilder est d'en faire la base d'un standard ouvert pour la vérification » a déclaré Rahul RAZDAN[2] de Cadence. Mike O'REILLY[3] annonce, lui : « Grâce à la puissance du C++ et à la communauté Open-Source, les langages de vérification propriétaires vont devenir obsolètes » [4].

Nous avons vu que l'absence de standard dans les langages de vérification était un réel problème. L'industrie micro-électronique a déjà connu cette situation à propos des langages de description de matériel, avant que VHDL et Verilog[5] n'occupent la place qu'ils ont aujourd'hui. Il est clair qu'il y a aujourd'hui de nombreux outils et certains seront amenés à disparaître. Le poids des acteurs, la politique au sein des instituts de standardisation sont des facteurs qui interviendront pour savoir quelles solutions s'imposeront.

Il est donc encore trop tôt pour prédire la réussite d'une librairie comme *TestBuilder*. La bibliothèque est récente (la version 1.0 est sortie il y a moins de dix mois) et on ne dispose ni d'études scientifiques ni de cas d'études importants (mis à part ceux de la société CADENCE elle-même[6]).

Signalons tout de même une première initiative significative : la société SYNAPTICAD a décidé de supporter *TestBuilder* dans son logiciel *TestBencher Pro* [7] . SYNAPTICAD a pour l'occasion porté le code source de *TestBuilder* sous Windows, ce qui rend la solution multi-plateformes.

[1] le nom du fichier doit cependant se terminer par .tw et non par .h
[2] vice-président et directeur général de la division Systems & Functional Verification de *Cadence*.
[3] directeur marketing du groupe Advanced Verification Products de *Cadence*.
[4] www.actual.fr/Clients/Cadence/cadence-pr-test\%20builder\%201-230401.html
[5] Rappelons que Verilog etait un langage propriétaire avant d'être racheté par CADENCE qui l'a ensuite mis dans le domaine public en 1990. http://www.angelfire.com/ca/verilog/history.html
[6] C++ testbench verifies path to Utopia : http://www.eetimes.com/story/OEG200010615S0063
[7] Annonce du 15 Décembre 2001, http://www.euro-eda.com/SynaptiCADSupportsTestBuilder.htm. La version 8.0 de *TestBencher Pro* est présentée comme « un environnement de test complet à moins de 20 000$ ». *TestBencher* apporte à *TestBuilder* un environnement intégré complet : on contrôle *tout* depuis *TestBencher*, des fichiers de tests jusqu'au visualiseurs sans avoir à utiliser de lignes de commandes pour les compilations ou pour le lancement des simulations.

4 Conclusion

LA VÉRIFICATION constitue aujourd'hui, de l'avis de tous, le *goulot d'étranglement* le plus sérieux dans la conception des circuits. Or la complexité de ces derniers suit une courbe exponentielle tandis que les contraintes du marché ne permettent pas un temps de conception plus long. La situation est de plus en plus critique et le domaine est actuellement en pleine effervescence comme nous l'avons montré dans le panorama.

La *simulation* qui consiste à tester un circuit pour différentes valeurs de ces entrées permet de détecter un certain nombre d'erreurs. Mais tandis que le nombre de tests nécessaires augmente fortement avec la complexité des circuits, la simulation reste un processus fondamentalement lent.
Le point clé est par conséquent de maximiser la probabilité de simuler et détecter les erreurs pour un coût (en temps, en travail et en temps de simulation) qui reste raisonnable. L'écriture de bancs de tests efficaces est devenu un impératif et nécessite de plus en plus l'utilisation de techniques de programmation modernes.

Les langages de description de matériel (HDL) se révèlent assez vite limités dans ce cadre. Les éditeurs ont donc imaginé de nouvelles solutions et plusieurs langages spécialisés dans la vérification (HVL) se partagent aujourd'hui le marché.

Plutôt que de développer un langage spécialisé, la société CADENCE a préféré ajouter au C++ une librairie open-source contenant les classes nécessaires à l'écriture de bancs de tests matériels : *TestBuilder*, en intégrant notamment un modèle de *transaction* permet d'augmenter le niveau d'abstraction des tests. Les briques constituées par les TVMs permettent d'*encapsuler* signaux et protocoles. Cela rappelle les *composants* aujourd'hui couramment utilisés dans les ateliers de génie logiciel.
Nous avons noté les avantages d'un tel découpage, notamment en ce qui concerne la couverture et l'efficacité des bancs de tests. Nous avons vu comment un tel modèle permettait une division des responsabilités ou encore une bien meilleure réutilisation des tests.

L'étude de la librairie nous a permis d'explorer les techniques utiles à la programmation des bancs de tests. Nous avons vu, notamment, comment *TestBuilder* permettait de mettre en œuvre les *tirages aléatoires avec contraintes* et la *vérification de propriétés temporelles*. Nous avons dressé un parallèle avec les fonctionnalités de *e/Specman* et pu noter que les possibilités étaient assez proches même si leurs mises en œuvre étaient différentes.
Nous avons évoqué l'importance de la *couverture fonctionnelle*, et mis en évidence la stratégie probable de CADENCE, à savoir imposer *TestBuilder* pour vendre son logiciel *Cockpit*.

La question qui reste ouverte est « quel sera, demain, le langage de vérification standard ». La compétition est ouverte et certaines solutions décrites ici sont probablement amenées à disparaître.

TestBuilder a de réels atouts pour s'imposer. L'utilisation du C++ semble en effet se généraliser dans la modélisation et *TestBuilder* est d'ores et déjà prêt pour cette évolution. Son côté *open-source* est par ailleurs, une garantie d'adhésion aux besoins des utilisateurs.

Les *outils formels*, de leur côté ont permis des résultats parfois spectaculaires. Assez longtemps cantonnés dans les laboratoires, ils apparaissent depuis quelques années sous forme de logiciels commerciaux à travers deux méthodes particulières, le contrôle d'équivalence et le contrôle de modèles. Ils ont gagné en simplicité d'utilisation et les théories sous jacentes permettent de traiter des circuits de plus en plus importants.
De nombreuses recherches sont en cours dans ce domaine et il est probable que les outils formels se généralisent à terme.

À plus court terme, on assistera peut-être à une *fusion* des outils traditionnels que sont les simulateurs avec les méthodes formelles pour aboutir à des outils dits *semi-formels*.
L'idée est d'utiliser des méthodes mathématiques à partir des résultats de la simulation pour augmenter très sensiblement la couverture de vérification.

Glossaire

Ce glossaire n'a pas l'objectif d'être exhaustif et il ne contient pas l'ensemble des termes présentés dans cet exposé. Il présente simplement le vocabulaire permettant d'aborder le sujet.

ASIC (Application Specific Integrated Circuit) : Circuit intégré dédié à une application.

CAD (Computer Aided Design) : Voir CAO

CAO : Conception Assistée par Ordinateur.

Circuit intégré : Un circuit intégré se représente comme un petit boîtier, le plus souvent en plastique, de dimensions centimétriques, d'où sortent quelques dizaines de pattes de raccordement électrique. A l'intérieur du boîtier, une pastille (puce ou chip) de semiconducteur en silicium est reliée à ces pattes par des fils de connexions très fins. C'est cette puce qui assure la fonctionnalité.

Design : Description d'un circuit intégré dans l'une des différentes étapes de sa conception. Nous pouvons parler d'un design pour un layout (dessin des masques), pour une représentation sous forme de blocs (description structurelle) et pour une description comportementale sous la forme d'un programme (VHDL ou Verilog). Ce concept rassemble l'idée de projet, d'ébauche, de dessin [Pel].

DUV (Design Under Verification) : Design en cours de vérification. Par exemple, la description d'un circuit au sein d'un simulateur.

EDA (Electronic Design Automation) : CAO électronique.

FPGA (Field Programmable Gate Array) : Circuit programmable (particulièrement utilisé en émulation).

HDL (Hardware Description Language) : Langage de description de matériel. VHDL et Verilog sont les deux HDL les plus utilisés.

HVL (Hardware Verification Language) : Langage de vérification de matériel.

IP (Intellectual Property) : L'IP permet aux industriels d'avoir accès à des éléments de circuits déjà conçus.

Netlist : Liste des nœuds définissant un schéma logique. Ce qu'on obtient après synthèse d'une description RTL écrite en HDL.

Niveau système : Ce niveau correspond au niveau d'abstraction le plus haut. Il permet de spécifier un système entier en termes de sous-systèmes matériels et logiciels interconnectés.

Niveau comportemental / description comportementale : Une description comportementale vise à décrire seulement la fonctionnalité d'un circuit à l'aide de langages séquentiels ou procéduraux.

Niveau transfert de registres (RTL) / description RTL : Le niveau RTL correspond à une description directement synthétisable. Il s'agit en fait du niveau de la logique synchrone.

Niveau logique / description logique : A ce niveau, le circuit est décrit comme un ensemble de bascules et de portes logiques interconnectées.

Niveau circuit / description layout : Ce niveau correspond au niveau d'abstraction le plus bas de la conception de circuits intégrés. La description de circuits sous forme de masques est directement utilisable par les processus de fabrication.

Synthèse : Passage du niveau RTL au niveau logique, puis layout, grâce à des outils de synthèse.

SoC (System On a Chip) : Système électronique entièrement implanté sur une seule puce de silicium. La compléxité de tels circuits est extrêmement élevée.

Time to market : Temps nécessaire pour concevoir et fabriquer un produit avant de le lancer sur le marché.

Verilog : Un des deux langages de description de hardware (HDL) les plus utilisés.

VHDL : Un des deux langages de description de hardware (HDL) les plus utilisés.

VLSI (Very Large Scale Integration) : circuits intégrés contenant quelques dizaines de milliers de composants au minimum.

Références

[Anc97] François ANCEAU, « ***La vérification formelle de circuits VLSI en milieu industriel*** », Juin 1997, http://lmi17.cnam.fr/~anceau/Documents/ofta.pdf

[Ben] Bob BENTLEY, « ***Validating the Intel® Pentium® 4 Processor*** », http://developer.intel.com/technology/itj/q12001/articles/art_3.htm

[Ber] « ***Le site de Janick BERGERON*** », http://www.janick.bergeron.com/

[Ber00] Janick BERGERON, « ***Writing testbenches, functionnal verification of HDL models*** », Kluwer Academic Publishers, 2000, ISBN : 0-7923-7766-4

[Cad00] BRAHME & AL., « ***The transaction-based verification methodology*** », Cadence Berkeley Labs technical report, Août 2000

[Cal93] J-P CALVEZ, « ***Spécification et conception des ASICs*** », Masson 1993, ISBN : 2–225–84216–7

[Cla] Celia CLAUSE, « ***A brief introduction to PLI (Programming Language Interface)*** », http://home.europa.com/~celiac/pli.html

[Cox01] COX & AL., « ***Creating a C++ library for transaction-based test bench authoring*** », Forum on Design Languages, Septembre 2001

[CW] Edmund M. CLARKE and Jeanette M. WING, « ***Formal Methods : State of the Art and Future Directions*** », http://www-2.cs.cmu.edu/afs/cs/project/calder/papers/acm/acm.ps

[Dee] Le site « ***DeepChip*** », http://www.deepchip.com

[Dil] David L. DILL et Sedar TASIRAN, « ***Simulation meets formal verification*** », http://www.cerc.utexas.edu/~jay/fv_surveys/DillICCAD99.ps

[Eet] Les sites « ***EETimes*** » et « ***EEDesign*** », http://eetimes.com et http://eedesign.com

[Geo01] Philippe GEORGELIN, « ***Vérification formelle de systèmes digitaux synchrones, basée sur la simulation symbolique*** », Thèse soutenue le 18 Octobre 2001, http://philippe.georgelin.free.fr

[Hou00] Dominique HOUZET, « ***Conception de circuits en VHDL*** », Cépadues Editions 2000, ISBN : 2-85428-527-1

[Mar94] John MARKOFF, « ***Circuit flaw causes pentium chip to miscalculate, Intel admits*** », New York Times, 24 Novembre 1994

[NVHPJ] François NAÇABAL, Carlos VALDERRAME, Fabiano HESSEL, Pierre PAULIN, Ahmed JERRAYA, « ***Co-Simulation C-VHDL pour la validation fonctionnelle de logiciel embarqué*** », http://tima.imag.fr/sls/documents/tsi_hessel.pdf

[Ore00] Michael O'REILLY, « ***Uncorking the verification bottleneck*** », New Electronic, Décembre 2000

[Pao01] Christophe PAOLI, « ***Validation de descriptions VHDL fondée sur des techniques issues du domaine du test de logiciels*** », Thèse soutenue le 20 décembre 2001, http://spe.univ-corse.fr/paoliweb/

[Pel] Gilles PELISSIER, « ***Présentation des outils de synthèse d'architecture*** », http://www.mea.isim.univ-montp2/~pravo/Hls/Polycops-pdf/chap6.pdf

[Phi01] Lars PHILIPSON, Lunds TEKNISKA HOGSKOLA « ***Survey compares formal verification tools*** », EE-Design, http://www.eedesign.com/story/OEG20011128S0037

[Rou] Bruno ROUZEYRE, « ***Synthèse architecturale*** », Polycopié de cours – LIRM, http://www.mea.isim.univ-montp2.fr/POLYCOPS/

[Sch99] Ouvrage collectif, coordination de Philippe SCHNOEBELEN, « ***Vérification de logiciels*** », Vuibert 1999, ISBN : 2–7117–8646–3

[Seg] Carl-Johan SEGER, « ***An introduction to Formal Hardware Verification*** », http://www.cerc.utexas.edu/~jay/fv_surveys/Seger.ps

[TB] « ***Manuels de la librairie TestBuilder et code C++*** », http://testbuilder.net

[Vsi] VSI Alliance™, « ***Taxonomy of Functional Verification*** », http://www.vsi.org/library/specs/ver111.pdf